THEORY OF COMPLEXITY CLASSES
VOLUME 1

Chee Keng Yap
Courant Institute of Mathematical Sciences
New York University
251 Mercer Street
New York, NY 10012

September 29, 1998

PREFACE

Complexity Theory is a thriving and growing subject. Its claim to a central position in computer science comes from the fact that algorithms permeate every application of computers. For instance, the term *NP*-completeness is by now in the vocabulary of the general computer science community. Complexity Theory has evolved considerably within the last decade. Yet there are few books that systematically present the broad sweep of the subject to a prospective student from a 'modern' point of view, that is, the subject as a practitioner sees it. The few available treatments are embedded in volumes that tended to treat it as a subfield of related and older subjects such as formal language theory, automata theory, computability and recursive function theory. There are good historical reasons for this, for our legacy lies in these subjects. Today, such an approach would not do justice to the subject – we might say that Complexity Theory has come-of-age. So one goal of this book is to present the subject fully on its own merits.

Complexity Theory can be practiced at several levels, so to speak. At one extreme we have analysis of algorithms and concrete complexity. Complexity Theory at this *low level* is highly model-dependent. At the other end, we encounter axiomatic complexity, recursive function theory and beyond. At this very *high level*, the situation becomes quite abstract (which is also one of its beauties); the main tools here center around diagonalization and the results often concern somewhat pathological properties of complexity classes. So what we mean by Complexity Theory in this book might be termed *medium level*. Here diagonalization remains an effective tool but powerful combinatorial arguments are also available. To distinguish the subject matter of this book from these closely related areas, I have tried to emphasize what I perceive to be our central theme: *Complexity Theory is ultimately about the relationships among complexity classes.* Of course, there are reasonable alternative themes from which to choose. For instance, one can construe Complexity Theory to be the theory of various models of computation – indeed, this seems to be the older point of view. But note that our chosen view-point is more abstract than this: models of computation define complexity classes, not vice-versa.

How Complexity Theory arrives at such an abstract theme may be of interest: hopefully such evidence is scattered throughout the entire book. But for a practitioner in the field, perhaps the most cogent argument is that most of the major open problems in the subject can be posed in the form: "Is J included in K?" where J and K are complexity classes. This certainly includes the P versus NP, the $DLOG$ versus $NLOG$ and the LBA questions. Other questions which prima facie are not about complexity classes (e.g. space versus time, determinism versus nondeterminism) can be translated into the above form. Even investigations about individual

languages (e.g. complete languages) can be viewed as attempts to answer questions about complexity classes or to place the language into a well-known complexity class (e.g., is graph isomorphism in P?). Of course, these by no means exhaust all the work in Complexity Theory but their centrality cannot be denied.

As for the student of complexity, some basic questions are more immediate: for a phenomenon as complex as complexity (no tautology intended) it is no surprise that the theory contains many assumptions, not all of which can be rigorously or even convincingly justified. We can only offer heuristic arguments and evidence. This is the burden of chapter one which presents the *methodological assumptions* of Complexity Theory. It comes from my attempt to distill what practitioners in this field had been (subconsciously or otherwise) cultivating in the past twenty years or more. By making them explicit (though by no means to our total satisfaction), it is hoped that they will speedup the entry of novitiates into the field. Indeed, this is out of self-interest: a field thrives with fresh ideas from each new generation of practitioners. Another possible result of explicating the underlying assumptions is renewed criticisms of them, leading to alternative foundations: again, this is most healthy for the field and certainly welcome. In any case, once these assumptions are accepted, we hope the student will discover a powerful and beautiful theory that offers much insight into the phenomenon of complexity. This is enough to justify the present theory (though not to exclude others).

There are several notable features of this book:

(a) We have avoided traditional entry-points to Complexity Theory, such as automata theory, recursive function theory or formal language theory. Indeed, none of these topics are treated except where they illuminate our immediate concern.

(b) The notion of *computational modes* is introduced early to emphasize its centrality in the modern view of Complexity Theory. For instance, it allows us to formulate a polynomial analogue of Church's thesis. Many of us are brought up on the sequential-deterministic mode of computation, called here the *fundamental mode*. Other computational modes include *nondeterministic*, *probabilistic*, *parallel*, and *alternating*, to name the main ones. These modes tend to be viewed with suspicion by non-practitioners (this is understandable since most actual computers operate in the fundamental mode). However, it is important to wean students from the fundamental mode as early as possible, for several reasons: Not only are the other modes theoretically important, the technological promise of economical and vast quantities of hardware have stirred considerable practical interest in parallel and other modes of computation. The student will also come to appreciate the fact that computational

modes such as represented in proof systems or grammars in formal language theory are valid concepts of computation. On the other hand, these alternative computational modes have distinct complexity properties which is in fact what makes the subject so rich.

(c) Traditionally, the computational resources of time and space have been emphasized (other resources are more or less curiosities). The recent discovery of the importance and generality of the computational resource of reversals is stressed from the beginning of the book. Of course, the number of results we currently have on reversals is meager compared to the number of papers devoted time and space; indeed through the writing of this book I am convinced that this imbalance should be remedied. (Paucity of results of other sorts are also evident: in the chapters on reducibilities, diagonalization and relative classes, we see no results for modes other than the fundamental and nondeterministic modes. It is very likely that the filling of these gaps will require new techniques). It is my strong belief that the new triumvirate of *time-space-reversal* gives us a more complete picture of computational complexity. We also treat simultaneous resource bounds, such as simultaneous time-space. Again this approach gives us a more rounded view of complexity. It is also a topic of increasing importance.

(d) I believe an important contribution of this book is the theory of valuations and choice machines. Besides its unifying and simplifying appeal, it addresses many foundational questions raised by the newer notions of computation such as interactive proofs. Researchers were able to simply ignore some of these questions in the past because they focused on "nice" situations such as polynomial time. But once the proper foundation is introduced, new issues arise on their own right. For instance, the use of interval algebra exposes new subtleties in the concepts of error. We hope that this inquiry is only the beginning.

(e) In attempting to give a coherent treatment of the subject, it is necessary to unify the widely varying notations and definitions found in the literature. Thus, many results on space complexity are proved using a version of Turing machine different from that used in time complexity. Yet a common machine model must be used if we want to study simultaneous time-space complexity. We choose the *off-line multitape Turing machines*. Another case where uniformity is badly needed is something so basic as the definition of "time complexity": to say that a nondeterministic machine accepts within t steps, some definitions require all paths to halt within this time bound; others are satisfied if some accepting path halt within this bound. We distinguish them

as *running time* and *accepting time*, respectively. Generalizing this distinction to other measures, we hence speak of *running complexity* versus *accepting complexity*. How comparable are these results? By and large, we would prefer to stick to accepting complexity because it seems more fundamental and afford simpler proofs. This we manage to do for most of the first half of the book. Unfortunately (or fortunately?), the corpus of known results seems rich enough to defeat any artificial attempt to impose uniformity in this respect. This is most evident in probabilistic computations where running complexity seems to be the more fruitful concept. A final example is the disparate notations for reducibilities. From the diverse possibilities, a unifying choice appears to be $\leq_t^c$ where t indicates the type of reduction (many-one, Turing-reducibility, etc) and c indicates the complexity considerations (polynomial time, log-space, etc). Or again, for a complexity class K, we prefer to say that a language is "K-complete under $\leq$-reducibility" rather than "$\leq$-complete for K": here the choice is driven by the wide currency of the term "NP-complete language". We are aware of the dark side of the grand unification impulse, which can rapidly lead to unwieldy notations: it is hoped that a reasonable compromise has been made within the scope of this book.

(f) In selecting results (many appearing in book-form for the first time) for inclusion in this book, I have tried to avoid those results that are essentially about particular machine models. This is consistent with our book's theme. The main exception is chapter two where it is necessary to dispose of well-known technical results concerning the Turing model of computation.

Finally, this book is seen as a self-contained and complete (though clearly non-exhaustive) introduction to the subject. It is written so as to be usable as a textbook for an advanced undergraduate course or an introductory graduate course. The didactic intent should be evident in the early chapters of this book: for instance, we may offer competing definitions (running complexity versus acceptance complexity, one-way oracle machines versus two-way oracle machines) even when we eventually only need one of them. By exposing students to such definitional undercurrents, we hope they will appreciate better the choices (which most experts make without a fuss) that are actually made. The later part of the book, especially volume two, is intended more as a reference and the treatment is necessarily more condensed.

A quick synopsis of this two-volume book is as follows: There are twenty chapters, with ten in each volume. Chapter 1 attempts to uncover the foundations and presuppositions of the enterprise called Complexity Theory. Chapter 2 establishes the basic machinery for discussing Complexity Theory; we try to confine most model-dependent results to this chapter. Chapter 3 on the class NP is really the

introduction to the heart this book: a large portion of the remainder of the book is either an elaboration of motifs begun here, or can be traced back to attempts to answer questions raised here. Chapters 4 to 6, is a study of the tools that might be termed 'classical' if this is appropriate for such a young field as ours: reducibilities, complete languages, diagonalization and translation techniques for separating complexity classes. Chapters 7 and 8 consider two important computational modes: probabilism and alternation. Although these two modes are seldom viewed as closely related, we choose the unusual approach of introducing them in a common machine model. One advantage is that some results known for one mode can be strengthened to their combination; also, contrasts between the two modes become accented. Chapter 9 is about the polynomial-time hierarchy and its cognates. Chapter 10 introduces circuit complexity: superficially, this is an unlikely topic since circuits describe finite functions. Traditionally, the interest here comes from the hope that combinatorial techniques may yield non-linear lower bounds for circuit complexity which in turn translates into non-trivial lower bounds for machine complexity. This hope has (as yet) not borne out but circuit complexity has yielded other unexpected insights into our main subject. Two other topics which we would have liked in volume 1 to round up what we regard as the core topics of the field are relativized classes and parallel computation. Unfortunately we must defer them to volume 2. The rest of volume 2 covers topics that are somewhat esoteric as well as some directions that are actively being pursued: randomness, structural approaches to complexity, alternative models of computation (storage modification machines, auxilliary Turing machines, etc), alternative complexity theories (such as Kolgomorov complexity, optimization problems, Levin's average complexity), specialized approaches to complexity (such as mathematical logic), and complexity of some classes of languages that have inherent interest (such context-free languages, theory of real addition, etc).

Ideally, Complexity Theory should be taught in a two course sequence; this is probably a luxury that many Computer Science curriculum cannot support. For a one course sequence, I suggest selections from the first 7 chapters and perhaps some advanced topics; preferably such a course should come after a more traditional course on the theory of automata and computability.

The reader is kindly requested to inform the author of any errors of commission as well as of omission. Since much of our material is organized in this manner for the first time, there will be inevitable rough spots; we ask for the readers' indulgence. All suggestions and comments are welcome.

I have variously taught from this material since 1981 at the Courant Institute in New York University, and most recently, at the University of British Columbia.

As expected, this ten-year old manuscript has evolved considerably over time, some parts beyond recognition. Although many individuals, students and colleagues, have given me many thoughtful feedback over the years, it is clear that the most recent readers have the greatest visible impact on the book. Nevertheless, I am very grateful to *all* of the following names, arranged somewhat chronologically: Norman Schulman, Lou Salkind, Jian-er Chen, Jim Cox, Colm Ó'Dúnlaing, Richard Cole, Bud Mishra, Martin Davis, Albert Meyer, Dexter Kozen, Fritz Henglein and Richard Beigel. The extensive comments of Professors Michael Loui and Eric Allender, and most recently, Professors Jim Cox and Kenneth Regan, from their use of these notes in classes are specially appreciated. Finally, I am grateful to Professor David Kirkpatrick's hospitality and for the facilities at the University of British Columbia while completing the final portions of the book.

C.K.Y.

New York, New York
March, 1991

Contents

Chapter 1

Initiation to Complexity Theory

January 16, 2001

This book presumes no background in Complexity Theory. However, "general mathematical maturity" and rudiments of automata and computability theory would be useful. This introductory chapter explores the assumptions of Complexity Theory: here, we occasionally refer to familiar ideas from the theory of computability in order to show the rationale for some critical decisions. Even without such a background the reader will be able to understand the essential thrusts of this informal chapter.

The rest of the book does not depend on this chapter except for the asymptotic notations of section 3.

This chapter has an appendix that establishes the largely standard notation and terminology of naive set theory and formal language theory. It should serve as a general reference.

1.1 Central Questions

Most disciplines center around some basic phenomenon, the understanding of which is either intrinsically interesting or could lead to practical benefits. For us, the phenomenon is the intuitive notion of *complexity of computational problems* as it arises in Computer Science. The understanding of what makes a problem (computationally) complex is one cornerstone of the art and science of algorithm design. The stress in 'complexity of computational problems' is on 'complexity'; the concept of 'computational problem', is generally relegated to the background.[1]

To set the frame of mind, we examine some rather natural questions. A main motivation of our subject is to provide satisfactory answers to questions such as:

(1) Is multiplication harder than addition?

The appeal of this question, first asked by Cobham [5], is that it relates to what are probably the two most widely known non-trivial algorithms in the world: the so-called *high school algorithms* for addition and multiplication. To add (resp. multiply) two n-digit numbers using the high school algorithm takes linear (resp. quadratic) time. More precisely, the addition (resp. multiplication) algorithm takes at most $c_1 n$ (resp. $c_2 n^2$) steps, for some positive constants c_1 and c_2. Here a 'step' is a basic arithmetic operation $(+, -, \times, \div)$ on single digit numbers. So a simple but unsatisfactory answer to (1) is 'yes' because $c_2 n^2$ dominates $c_1 n$ when n gets large enough. It is unsatisfactory because the answer only says something

[1] There is no general theory of computational problems except in special cases. Such a theory should study the logical structure of problems, their taxonomy and inter-relationships. Instead, complexity theory obliterates all natural structures in computational problems by certain sweeping assumptions we will come to.

about *particular* methods of adding and multiplying. To provide a more satisfactory answer, we probe deeper.

It is intuitively clear that one cannot hope to do additions in less than n steps. This is because any algorithm must at least read all the n input digits. So the high school method for addition is optimal – note that here and throughout the book, *optimality is taken up to some multiplicative constant factor*. The situation is less clear with multiplication: Is there any algorithm for multiplication that is asymptotically faster than the high school method? The answer turns out to be 'yes'; in 1971 (culminating a series of developments) Schönhage and Strassen [30] discovered what is today asymptotically the fastest known multiplication algorithm. Their algorithm takes $c_3 n \log n \log\log n$ steps[2] on a *Turing machine* (to be introduced in Chapter 2). Since a Turing machine turns out to be more primitive than any real-world computer, this means the same time bound is achievable on actual computers. However, the large constant c_3 in the Schönhage-Strassen algorithm renders it slower than other methods for practical values of n. For now, let us just accept that Turing machines are indeed fundamental and thus statements about computations by Turing machine are intrinsically important. But is the Schönhage-Strassen algorithm the best possible? More precisely,

(2) Must every Turing machine that multiplies, in worst-case, use at least $cn \log n \log\log n$ steps, for some $c > 0$ and for infinitely many values of n? Here c may depend on the Turing machine.

This is an important open problem in Complexity Theory. A negative answer to question (2) typically means that we explicitly show an algorithm that is faster than that of Schönhage-Strassen. For instance, an algorithm with running time of $n \log n$ will do. We say such an algorithm shows an *upper bound* of $n \log n$ on the complexity of multiplication. (It is also conceivable that the negative answer comes from showing the existence of a faster algorithm, but no algorithm is exhibited in the proof[3].) On the other hand, answering question (2) in the affirmative means showing a *lower bound* of $cn \log n \log\log n$ on the complexity of multiplication. Such a result would evidently be very deep: it says something about *all* possible Turing machines that multiply! Combining such a result with the result of Schönhage-Strassen, we would then say that the *intrinsic complexity* of multiplication is $n \log n \log\log n$. In general, when the upper and lower bounds on the complexity of any problem P meet (up to a constant multiplicative factor) we have obtained a bound which is intrinsic[4] to P. We may now (satisfactorily) interpret question (1) as asking whether the intrinsic complexity of multiplication is greater than the intrinsic complexity of addition. Since the intrinsic complexity of addition is easily seen to be linear, Cobham's question amounts to asking whether multiplication is intrinsically non-linear in complexity. (Most practitioners in the field believe it is.) Generally, for any problem P, we may ask

(3) What is the intrinsic complexity of P?

It turns out that there is another very natural model of computers called *Storage Modification Machines* (see §5) which can multiply in linear time. This shows that complexity is relative to a given model of computation. The student may recall that as far as the theory of computability goes, all reasonable models of computation are equivalent: this is known as *Church's Thesis*. But simplistic attempts to formulate analogous statements in Complexity Theory would fail. For instance, there are problems which can be solved in linear time in one model but provably take cn^2 (for some $c > 0$, for infinitely many n) in another model. So a fundamental question is:

(4) Which model of computation is appropriate for Complexity Theory?

[2]Unless otherwise indicated, the reader may always take logarithms to the base 2. We shall see that the choice of the base is inconsequential.

[3]Such proofs are known but are rare. For instance, the recent work of Robertson and Seymour on graph minors leads to precisely such conclusions (see for example [18]). Indeed the situation could be more complicated because there are degrees of explicitness.

[4]See section 8 for more discussion of this.

Perhaps there is no uniquely appropriate model of computation for Complexity Theory. This suggests that we examine various models and discover their inter-relations. We shall see in section 7 that we can recover some version of Church's Thesis in Complexity Theory if we take some care in selecting models and classifying them according to their 'computational mode'.

This question about intrinsic complexity of a problem P begs another question:

(5) How should the input and output values of problems be encoded?

For instance, we have implicitly assumed that the multiplication problem has its inputs and outputs as binary numbers. But other representations of numbers are conceivable and this choice can be shown to affect the answer to (3). In section 4, we discuss this in more detail (it turns out that without a proper theory of problems, few precise statements can be made about choice of encodings and we are stuck with assumptions such as in the case of multiplication: each problem P comes equipped with a definite encoding).

The discussion of the complexity of multiplication so far centers around speed of computation or *time*. Time is an instance[5] of a *computational resource*. We measure complexity in terms of the amount of computational resources used up during a computation. Another natural resource is memory size or *space*. (The high-school algorithm is seen to take quadratic space with usual hand-calculation but it can easily be reduced to linear space.) We can imagine composite resources that combine time and space, for instance.

(6) What are other natural computational resources and their inter-relations?

It is seen that time and space are related: assuming each step can access one new unit of space, the amount of space used by an algorithm is no more than the amount of time consumed. However, there is a fundamental difference between time and space: space, but not time, is reusable. There can be trade-offs between time and space in the sense that an algorithm can run in less time only at the (unavoidable) cost of using more space. Up to a degree, resources depends on the computational model.

The high-school multiplication algorithm exhibits an interesting feature not generally true of algorithms: for all inputs of a given size, the algorithm executes the same number of steps. This is because it has no 'decision' steps (e.g. 'if $x = 0$ then $\cdots$ else $\cdots$').[6] The variety of well-known sorting algorithms are examples of decision-making algorithms. For such algorithms the running time may vary for different inputs of the same size. Then one has the choice of measuring the *worst-case* or the *average-case* time. In other words, the method of *aggregating* the total resources used by the algorithm is an issue to be decided in measuring complexity.

Another issue arises when we consider probabilistic computations where the algorithm makes random decisions by 'tossing a coin'. For a fixed input, different runs of the algorithm may produce different outputs. We are then faced with defining a suitable notion of the 'correct output': this is not formidable once we realize that the output can be a random variable.

We summarize the above discussion by saying that the *complexity measure* is a function of (a) the model of computation, (b) the computational resources of interest, (c) the method of aggregating the resource usage and (d) the definition of the algorithm's output (not to be confused with encoding the output of a computation problem). Thus question (6) translates into a question about complexity measures.

The theory in this book will address these and other related questions. In the rest of this chapter, we lay the foundations by discussing some issues of methodology. The assumptions of methodology are embodied in conventions, definitions and notations. Whenever possible, we will isolate them with a letter label ((A), (B), etc.).

[5] No pun intended.

[6] One can argue that the multiplication algorithm has decisions in the form of checking for carry bits. However, such decisions are unessential for integer multiplication since we may assume that carry bits (possibly 0) are present at each step. Nevertheless we can modify the algorithm to incorporate decision steps that are not so easily dismissed, such as to check if one of its arguments is zero.

1.2 What is a Computational Problem?

The concept of a *computational problem* (or simply, *problem*) in informal language can take many meanings and shades. Even restricting the concept to its use in Computer Science is too wide. The theory we propose to develop takes the prototypical and simplest aspects only. This is a natural choice to start with, and it is already enough to give us a rich and meaningful theory.

There are three main aspects of a problem in its natural context: (i) The problem statement, P. (ii) The methods available for its solution, M. (iii) The criteria for evaluating and comparing different solutions, C. The triple $\langle P, M, C\rangle$ may be called a *problem-in-context*. We will identify M with the *model of computation* and C with the *complexity measure*. A *solution* to $\langle P, M, C\rangle$ is simply an algorithm in M that satisfies the specifications of P and that is optimal or correct in the sense of C. It is not very meaningful to discuss P without this context which may be implicit. In practice, it is not always easy to separate a problem-in-context into the three components. Roughly speaking, P specifies the input-output behavior (i.e., the relationship between any given input x with the output y of any purported algorithm for this problem). Generally, M is dependent on P, and likewise C depends on both M and P. If M' (respectively, C') is different from M (respectively, C) then the two problems-in-context $\langle P, M, C\rangle$ and $\langle P, M', C'\rangle$ should really be considered different problems, even though they share a common P.

Let us illustrate the preceding discussion using the well-known example of *sorting*. Let P be the sorting problem where, given a set X of integers, the problem is to 'arrange the set into an ascending sequence'. Three very different models of computation M_i $(i = 1, 2, 3)$ have been extensively studied in the literature.

M_1**:** The *comparison tree model* where algorithms are finite binary trees whose nodes correspond to comparisons.

M_2**:** The *tape merging model* where algorithms wind and rewind tapes based on outcomes of comparing the data at the front-ends of the tapes.

M_3**:** The *comparison network model* where algorithms are directed acyclic graphs whose nodes (called comparators) have in-degrees and out-degrees of 2.

The complexity measure studied is dependent upon the model: In M_1, the goal is to minimize the height of the binary trees. Depending on whether we consider 'worst-case height' or 'average height', we get different measures. In M_2, the criterion is to minimize tape motion (rather than the number of comparisons, as in M_1). In M_3, the criterion is to minimize the number of comparators or to minimize the length of the longest path.

Intuitively, P is the most fundamental of the three parameters $\langle P, M, C\rangle$, and we shall try to restrict the use of the term 'problem' to mean P alone. This is not entirely possible but we shall largely get by with the (mild) myth that our problems P do not imply any particular M or C. So the task that confronts us after the preceding clarifications, is to formalize P, this notion of 'problem-in-the-restricted-sense'. Let us begin by listing some typical problems in Computer Science:

(i) (Rubik's puzzle) For any given configuration of the Rubik's Cube, find the least number of moves required to get to the configuration which is monochromatic on each face. This is a *finite problem* since there are a finite number of possible input configurations.

(ii) (Fibonacci numbers) Let $f(n)$ be the nth Fibonacci number. The sequence of Fibonacci numbers is $0, 1, 1, 2, 3, 5, 8, \ldots$. For each input n, the problem is to compute $f(n)$. This is an instance of computing a number theoretic function[7]. In fact, recursive function theory uses number theoretic functions as the basic objects of study.

[7] A *number theoretic function* f is a partial function from the natural numbers $0, 1, 2, \ldots$ to natural numbers. Note that the ancients do not regard zero as a natural number, although mathematicians have more and more come to regard it as natural. We take the modern viewpoint in this book.

(iii) (Planarity Testing) For each graph G, the problem is to decide if G is planar (i.e., embeddable in the plane). This is an example of a *decision problem* where the algorithm has to decide between a 'yes' and a 'no' answer. Alternatively, this is called a *recognition problem* where we interpret the algorithm as trying to recognize those inputs representing planar graphs.

(iv) (Linear Programming) For each m by n matrix A and an n-vector c, find any n-vector x such that $Ax \geq 0$ and $c \cdot x$ is maximized. If there are no x satisfying $Ax \geq 0$, or if $c \cdot x$ is unbounded, indicate so. (We assume that m and n can vary and are part of the input.) This exemplifies the class of *optimization problems* which typically arise in the field of Operations Research.

(v) (Element identification) For each input set X of integers, $|X| < \infty$, construct a data-structure $D(X)$ such that queries Q of the form: 'Is x in X?' can be rapidly answered using $D(X)$. This is an instance of a *preprocessing problem* where the inputs are given in two stages and separate algorithms are involved in each stage. In the first stage we have an algorithm A to 'preprocess' the input X. In the second stage, we have an algorithm B which uses the preprocessed structure $D(X)$ to answer queries about X. If X comes from some linear order, then a typical solution to this 'element identification problem' builds a binary search tree to serve as $D(X)$ and uses a binary search tree algorithm for the second stage. Many preprocessing problems arise in Computational Geometry. For example: Given a set X of points in the plane, construct $D(X)$. We want to use $D(X)$ to quickly answer queries of the form: "Retrieve the subset of points in X that are contained in a given half-plane H" (It turns out that we can build a $D(X)$ that uses linear space and answer queries on $D(X)$ in logarithmic time.)

How can we hope to build a theory that encompasses such a wide variety of problems? The key is that any problem that can be solved mechanically by a computer is ultimately encodable into a finite sequence of symbols, where the symbols come from some arbitrary but fixed finite set Σ. This is essentially the argument used by Turing in his celebrated paper [32] in which the machines now bearing his name were introduced.

A finite set Σ of symbols is called an *alphabet*. A finite sequence of symbols from Σ is called a *word* over Σ. The set of all words over Σ is denoted by Σ^*. A subset L of Σ^* is called a *language* over Σ. The empty sequence, denoted ϵ, containing no symbols is also a word. The *length* of a word w is denoted $|w|$.[8] Thus $|\epsilon| = 0$. We shall embody (part of) the Turing analysis in the following convention.[9]

(A) The input and output objects of a problem are words over some arbitrary but fixed alphabet.

We could have allowed different alphabets for the input and output objects but this is not essential. For each problem P in (i-v) above, we chose some encoding of the input and output domains of P into Σ^*. In (i), we may encode each Rubik cube configuration by specifying for each face (in some predetermined order) the colors of each square on the face. The answer encodes a sequence of moves, where we must choose a systematic representation for the cube faces and their rotations. In (ii), we may use binary numbers to encode natural numbers. Note that each of problems (i) and (ii), after the choice of a suitable encoding, becomes a function $f : \Sigma^* \to \Sigma^*$. In (iii), a graph G is represented by the list (in any order) of its edges. The problem of testing for planar graphs may now be represented as a language L over Σ where a word w is in L iff w represents a graph G which is planar. In (iv), we have to be more careful. In mathematics, the problem usually assumes that the entries of the matrix A, vectors c and x are arbitrary real numbers. For computational purposes, we will assume that these are rational numbers of arbitrary precision. (In general, replacing real numbers by rationals affects the nature of the problem.) The linear

[8] In general, there ought to be no confusion over the fact that we also use $|\cdot|$ to denote the absolute value of a real number as well as the cardinality of a set: it will be clear from context whether we are discussing strings, numbers or sets. One case where this overloading of notation might be confusing will be noted when it arises.

[9] This is analogous to the situation in computability theory where we have the arithmetization or Gödelization of machines. Thus (A) amounts to the 'arithmetization of problems'.

programming problem can be encoded as a binary relation $R \subseteq \Sigma^* \times \Sigma^*$, where $(w, v) \in R$ iff w encodes an input (A, c) and v encodes an output vector x such that $c \cdot x$ is maximized, subject to $Ax \geq 0$. In (v), we may represent X by a sequence of binary numbers. We then encode the element identification problem as a 3-ary relation $R \subseteq \Sigma^* \times \Sigma^* \times \Sigma^*$ where $(u, v, w) \in R$ iff u encodes a set X, v encodes a query on X, and w encodes the answer to the query.

Finite problems such as (i) are usually of little interest to us. Once the encoding is fixed, we find that different problems can have different forms: language in (iii), functions in (i) and (ii), and relations in (iv) and (v). (Note that this is in order of increasing generality.) Which is the appropriate paradigm? It turns out that we choose the simplest form:

(B) A problem is a language.

This answers the question posed as the title of this section.[10] However, it is far from clear that (B) will be satisfactory for studying problems that are functions or relations. The sense in which (B) is reasonable would hopefully be clearer by the end of this chapter. Before we see some deeper reasons (in particular in §8.2, and also §2 of chapter 3) for why (B) is a reasonable choice, it is instructive to first note a simple connection between recognition problems and functional problems.

Consider the following recognition problem: given a triple (x, y, z) of binary numbers, we want to recognize if $xy = z$. This problem is obtained by a transformation of the functional problem of multiplication, and it is easy to see that the transformation is completely general. Clearly an algorithm to solve the functional multiplication problem leads to a solution of this problem. It is not obvious if there is a converse, that is, a solution to the recognition problem will lead to a solution of the functional problem. Fortunately, a slight modification of the recognition problem does have the desired converse (Exercises). Part of the justification of (B) hinges upon such an ability to convert any functional problem of interest to a corresponding recognition problem, such that their complexity are closely related.

1.3 Complexity Functions and Asymptotics

Definition 1 Let f be a partial function from $\mathbb{R}$ real numbers $\mathbb{R} \cup \{\infty\}$ (the extended reals). If $f(x)$ is undefined, we write $f(x) \uparrow$, otherwise write $f(x) \downarrow$. Note that $f(x) \uparrow$ is distinguished from $f(x) = \infty$. We call f a *complexity function* if f is defined for all sufficiently large natural numbers. The complexity function is *finite* if $f(x) < \infty$ whenever $f(x)$ is defined. ■

Let us motivate some decisions implicit in this definition. For each algorithm A and computational resource R, we can associate a function $f_{A,R}(n)$ (or, simply $f_A(n)$ if R is understood) such that for inputs of size n, the algorithm uses no more than $f_A(n)$ units of R and for some input of size n, the algorithm uses exactly $f_A(n)$ units of R. By definition, $f_A(n) = \infty$ if the algorithm uses an unbounded amount of R on some input of size n. We also specify that $f_A(x)$ is undefined for x not a natural number.

The definition of complexity functions is basically intended to capture such functions as f_A. Usually, the domain and range of f_A is the set of natural numbers. So why do we extend these functions to real numbers (and artificially allow them to be undefined at non-natural numbers). First, range of f_A need not be natural numbers if $f_A(n)$ measures the average use of resource R over all inputs of length n. Next, the function f_A may be quite bizarre or difficult to determine exactly; for many purposes, it suffices to get an upper bound on f_A using some nicer or more familiar functions. For instance, it may be enough to know $f_A(n) \leq g(n)$ where $g(x) = x^2$, even though $f_A(n)$ is really $n^2 - 3\log n + 4.5e^{-n}$. These nicer functions are usually defined over the reals: for instance, $g(x) = \log_2 x$ or $g(x) = x^{1/2}$. Many of these functions (logarithms, division) are partial functions. We could artificially convert a function such as $g(x) = \log_2 x$ into a number theoretic function by restricting its domain x to be natural numbers and redefining

[10] Earlier we said that the study of computational problems is relegated to the background in Complexity theory: we see this embodied in assumptions (A) and (B) which, together, conspire to strip bare any structure we see in natural problems. Of course this loss is also the theory's gain in simplicity.

$g(n) := \lceil \log_2 n \rceil$ (this is sufficient for upper bounds). But this destroys natural smoothness properties, and becomes awkward when forming the functional composition of two such functions, $(g_1 \circ g_2)(x) = g_1(g_2(x))$. (Composition of complexity functions is important because it corresponds to certain subroutine calls between algorithms.) Our solution to these requirements is to let the domain of g be reals, and for g to be defined at as many values as convenient. In practice, complexity functions tend to be non-negative and monotonically non-decreasing, for sufficiently large values of its arguments.

The complexity function f_A appears to have properties that are unessential or incidental to the problem for which A is a solution. There are three reasons for this: (a) First, the behavior of f_A for initial values of n seems unimportant compared to the eventual or asymptotic behavior of $f_A(n)$. To see this, suppose that $f_A(n)$ is "unusually large" for initial values of n. For instance, $f_A(n) = 10^{1000}$ for $n < 20$ and $f_A(n) = n$ otherwise. We could replace A by another B which does a 'table look-up' for the first 100 answers but otherwise B behaves like A. Under reasonable assumptions, B uses almost no resources for $n \leq 100$, so $f_B(n)$ is small for $n \leq 100$. Since B is essentially A, this illustrates the incidental nature of initial values of f_A. (b) Second, most algorithms have some degree of model independence which we would like to see reflected in their complexity. For instance, the essential idea of the "mergesort algorithm" (or substitute your favorite algorithm) is generally preserved, whether you implement it in Pascal or C/C++ or some more abstract computational model. This fact shows when you analyze the complexity of the algorithm under each of these models: they are all related by a constant factor. (c) Third, even when we fix a particular model of computation, it turns out that one can often "speed-up" an algorithm A by desired any constant factor, without essential change to the algorithmic ideas. We will see this in Chapter 2 for the Turing machine model, where any algorithm A can be modified to another B such that each step of B imitates about 1/c steps of A. We would like f_A and f_B to be regarded as as equivalent.

These considerations lead us to the next two definitions.

Definition 2 Let f, g be complexity functions. We say f *dominates* g if there exists a positive constant n_0 such that for all $n \geq n_0$, $g(n) \leq f(n)$ whenever both sides are defined. We have a notation for this:

$$f \geq g(\text{ev.})$$

(and read "$f \geq g$ eventually"). If F and G are two sets of complexity functions, we say F *dominates* G if for each $g \in G$ there is an $f \in F$ that dominates it. ■

The eventually notation is quite general: if $R(x)$ is a predicate on real numbers x in some domain $D \subseteq \mathbb{R}$, then we say "$R(x)$ eventually" if there is some x_0 such that for all $x \in D$, $x \geq x_0$ implies $R(x)$ holds.

The next definition/notation is of fundamental importance in complexity theory. There are several alternative formulations of this notation which dates back to Du Bois-Reymond (1871) and Bachmann (1894); and Knuth's[21]. We chose a variant [7] that seems particularly useful:

Definition 3 (The O-notation) For any complexity function f, $O(f)$ denotes the set of all complexity functions g such that for some positive constant $C = C(g)$,

$$0 \leq g \leq C \cdot f(\text{ev.}).$$

Note that $O(f)$ is empty unless $f \geq 0(\text{ev.})$. If F is a set of functions, $O(F)$ denotes the union over all $O(f)$ for $f \in F$. We read '$O(F)$' as 'big-Oh of F' or 'order of F'. ■

We extend this definition to recursively defined expressions.

Syntax and semantics of O-expressions. The set of *O-expressions* is defined by the following recursive rule. Let n be a real variable[11], f be any symbol denoting a complexity function, and c any symbol

[11]The literature uses 'n' for the variable of (univariate) complexity functions, in part because domains of complexity functions are often restricted to natural numbers. We continue to use 'n' with complexity functions even though we intend to let n range over all real numbers.

that denotes a real constant. The following are O-expressions:

$$\begin{array}{rl} \text{Basis:} & f, \quad n, \quad c \\ \text{Induction:} & O(E), \quad E+F, \quad E-F, \quad E \cdot F, \quad E^F, \quad E \circ F. \end{array}$$

where E, F are recursively defined O-expressions. If E is the symbol f, we may write '$f(F)$' instead of '$f \circ F$' ($\circ$ denotes functional composition). We may freely introduce matching parentheses to improve readability.

Note an O-expression need not contain any occurrences of the symbol 'O'. An O-expression that contains some occurrence 'O' is called an *explicit* O-expression; otherwise it is an *implicit* O-expression.

The following are some O-expressions.

$$3 + 2^n, \quad n + O(n^{-1} + \log(n) + 5), \quad O(n^2) + \log^{O(1)} n, \quad f^{O(g)}, \quad (f(g))^{h^2}.$$

Here '1' denotes the constant function, $f_1(n) = 1$ for all n; 'n' denotes the identity function $\iota(n) = n$, etc.

Each O-expression E denotes a set $[E]$ of complexity functions. We define $[E]$ recursively: (Basis) If $E = f$ is a function symbol, then $[f]$ is the singleton set comprising the function denoted by f. Similarly if $E = n$ or $E = c$, $[n] = \{\iota\}$ and $[c] = \{f_c\}$ where $f_c(n) = c$. If $E = O(F)$ then $[E]$ is defined to be $O([F])$, using definition 3 above. The semantics of the other O-expressions are the obvious ones:

$$[E \circ F] := \{e \circ f : e \in [E], f \in [F]\}, \quad [E + F] := \{e + f : e \in [E], f \in [F]\}, \quad \text{etc.}$$

Henceforth, we intend to commit the usual linguistic abuse, by letting symbols (syntactical objects) such as f and c stand for the functions (semantical objects) that they denote, when the context demand a function rather than a symbol. This abuse should not lead to any confusion.

The main use of O-expressions is this: If F is an O-expression and E is an explicit O-expression, we may say

$$\text{'}F \text{ is } E\text{' and write '}F = E\text{'}.$$

This simply means that $[F] \subseteq [E]$. An important special case is where $F = f$ and $E = O(g)$ where f, g are function symbols. For instance, if $t(n)$ denotes the time complexity of the high school multiplication algorithm then we may write '$t(n) = O(n^2)$'. Note that we restrict E to be an explicit O-expression since there is potential for confusion otherwise.

The use of the equality symbol in this context is clearly an abuse of our usual understanding of equality: we do not regard '$E = F$' and '$F = E$' as interchangeable. Thus $O(n^2) = O(n^3)$ is true but $O(n^3) = O(n^2)$ is not. For this reason, the equality symbol here is called the *one-way equality*.

A corollary of the *inclusion interpretation* of the one-way equalities is this: an *inequality* involving O-expressions is to be interpreted as *non-inclusion* '$\not\subseteq$'. Thus '$n \log n \neq O(n)$' is true but '$n \neq O(n \log n)$' is false.

Some authors avoid the one-way equality symbol by writing the more accurate form

$$\text{'}F \subseteq E\text{'}$$

or, in the case F is a function symbol f, '$f \in E$'. We shall not use this alternative form.

Example 1 Further examples of usage:

(i) The O-expressions $O(1), O(n)$ and $n^{O(1)}$ denote, respectively, the set of all functions that are dominated by constants, by linear functions and by polynomial functions. For instance, $\frac{1}{1+n^2} = O(1)$ but $\frac{1}{n} \neq O(1)$. Strictly speaking, $\frac{1}{1+n^2}$ and $\frac{1}{n}$ are not O-expressions since we do not allow division.

(ii) Depending on the context, the appearance of an O-expression may sometimes denote a set of functions, rather than as denote some unspecified member in this set. For instance, when we write $NP = NTIME(n^{O(1)})$, we intend to use the full set of functions denoted by the expression "$O(1)$". This is a potential pitfall for students.

(iii) By the non-inclusion interpretation, $f \neq O(g)$ means that for all $c > 0$, there are infinitely many n such that $f(n) > cg(n)$. There is, however, a useful intermediate situation between $f = O(g)$ and $f \neq O(g)$: namely, there exists $c > 0$ such that for infinitely many n, $f(n) > cg(n)$. Below, we will introduce a notation to express this.

(iv) The O-notation factors out unessential features of complexity functions: thus we say '$f(n) = O(\log n)$' without bothering to specify the base of the logarithm.

(v) The notation saves words: we simply write '$f(n) = n^{O(1)}$' instead of saying that $f(n)$ is dominated by some polynomial. It also conserves symbols: we say '$f(n) = O(1)^n$' instead of saying that there exists $n_0, c > 0$ such that $f(n) \leq c^n$ for all $n \geq n_0$. Thus we avoid introducing the symbols c and n_0.

(vi) It is handy in long derivations. For example,

$$n + O(n^{-1}) = n + O(1) = O(n) = O(n^2).$$

■

A subscripting convention.[12] We augment the O-notation with another useful convention:

(i) It often occurs that we want to pick out some *fixed but non-specific* function f' in the set $O(f)$, depending on some parameter α. Then we write $O_\alpha(f)$ to refer to this f'. If f' depends on several parameters $\alpha, \beta, \ldots$ then we write $O_{\alpha,\beta,\ldots}(f)$. An O-expression is *fully subscripted* if each occurrence of 'O' is subscripted. So a fully subscripted O-expression refer to a single function.

(ii) As an extension of the one-way equality, the 'equality symbol' between two fully subscripted O-expressions denotes domination between two functions. For instance, we write '$g = O_\alpha(f)$' to indicate that g is dominated by some $f' \in O(f)$ whose choice depends on α. There is much room for syntactic ambiguity and we assume common sense will avoid such usage (for instance, an implicit O-expression E is, by definition, fully-subscripted and it would be confusing to write '$g = E$').

(iii) Sometimes we want to annotate the fact that among the various occurrences of an O-expression, some refers to the same function. We may use subscripts such as O_1, O_2, etc., as an 'in-line' device for showing this distinction.

Example 2 We illustrate the subscripting convention.

(i) We say an algorithm A runs in time $O_A(n)$ to mean that the running time is dominated by kn for some k depending on A.

(ii) We could write $n2^k = O_k(n)$ as well as $n2^k = O_n(2^k)$, depending on the context.

(iii) Suppose $g(n) = O_1(n^2 \log n)$ and $f(n) = O_2(n) + 3O_1(n^2 \log n)$. The two occurrences of 'O_1' here refer to the same functions since they subscript identical O-expressions. Then we may write their sum as $f(n) + g(n) = O_2(n) + 4O_1(n^2 \log n) = O_3(n^2 \log n)$.

(iv) We emphasize a fully-subscripted O-notation no longer denotes a set of functions. Hence it is meaningful to write: '$O_1(f) \in O(f)$'.

(v) We illustrate a more extensive calculation. The following recurrence arises in analyzing the running time $T(n)$ of certain list manipulation algorithms (e.g., the 'finger tree' representation of lists in which we admit operations to split off prefixes of a list): $T(1) = 1$ and for $n \geq 2$,

$$T(n) = \max_{1 \leq i < n} \{T(i) + T(n-i) + \log \min\{i, n-i\}\}$$

[12] This convention is peculiar to this book.

To see the subscripting convention in action, we now prove that $T(n) = O_1(n) - O_2(\log(2n))$. If $n = 1$, this is immediate. Otherwise,

$$\begin{aligned} T(n) &= \max_{1\le i<n}\{O_1(i) - O_2(\log(2i)) + O_1(n-i) - O_2(\log(2(n-i))) \\ &\qquad + \log\min\{i, n-i\}\} \\ &= \max_{1\le i<n}\{O_1(n) - O_2(\log(2i)) - O_2(\log(2(n-i))) + \log\min\{i, n-i\}\} \\ &\le \max_{1\le i\le n/2}\{O_1(n) - O_2(\log(2i)) - O_2(\log(2(n-i))) + \log i\} \\ &\quad \text{(since the expression is symmetric in } i \text{ and } n-i) \\ &= O_1(n) - \min_{1\le i\le n/2}\{O_2(\log(2(n-i)))\} \\ &\quad \text{(we may assume } O_2(\log(2i)) \ge \log i) \\ &= O_1(n) - O_2(\log n) \end{aligned}$$

■

Note that we have used "$\le$" in the above derivation. This is something we would not write without the accompanying subscripting convention.

We shall have occasion to use four other related asymptotic notations, collected here for reference:

Definition 4 (Other asymptotic notations)
(i) $\Omega(f)$ denotes the set of functions g such that there exists a positive constant $C = C(g)$ such that $C \cdot g \ge f \ge 0$(ev.).

(ii) $\Theta(f)$ denotes the set $O(f) \cap \Omega(f)$.

(iii) $o(f)$ denotes the set of all g such that for all $C > 0$, $C \cdot f \ge g \ge 0$(ev.). This implies that ratio $g(n)/f(n)$ goes to zero as n goes to infinity. Also $o(f) \subseteq O(f)$.

(iv) $\omega(f)$ denotes the set of all g such that $C \cdot g \ge f \ge 0$(ev.). This implies the ratio $g(n)/f(n)$ goes to infinity as n goes to infinity. ■

These notations are used in ways analogous to that in O-notations; in particular, we use one-way equalities such as '$f = o(g)$'. The subscripting convention extends to the these notations. The O-notation and the Ω-notation are inverses in the sense that

$$f = O(g) \iff g = \Omega(f).$$

Similarly,

$$f = o(g) \iff g = \omega(f).$$

To verbally distinguish the O-notations from the o-notations, we also call them the "Big-Oh" and the "Small-Oh" (or, "Little-Oh") notations, respectively. Clearly, $f = o(g)$ implies $f = O(g)$. Also $f = \Theta(g)$ iff $g = \Theta(f)$ iff $f = O(g)$ and $g = O(f)$.

We sometimes call the set $O(f)$ the *big-Oh order of* f, and if $g = O(f)$, we say g and f *have the same big-Oh order*. Similarly, $\Theta(f)$ is the *big-Theta order of* f, etc.

We warn that incorrect usage of these less-frequently used asymptotic notations is common. Furthermore, they are sometimes given rather different definitions. For instance, in [1] '$\Omega(f)$' denote the set of functions g such that for some c, $g(n) \ge cf(n)$ for infinitely many n. (This is the intermediate situation between $g = O(f)$ and $g \ne O(f)$ mentioned above.) But notice that '$g = \Omega(f)$' under the definition of [1] is recaptured as our notation

$$g \ne o(f).$$

Although we seldom use this concept in this book, it is a typical situation that obtains when one proves "lower bounds" in concrete complexity.

Three final notes on usage:
(a) Clearly we could introduce Ω-expressions, etc. The meaning of *mixed* asymptotic expressions such as $O(n^2) + \Omega(n \log n)$ can also be defined naturally. A proper calculus of such expressions needs to be worked out. (Exercises) Fortunately, there seems to be little need of them.
(b) The reader must have noticed the similarities between the expression $f = O(g)$ and the inequality $x \leq y$.[13]. In fact, the similarities extend to the other notations:

$$f = O(g), \quad f = \Theta(g), \quad f = \Omega(g), \quad f = o(g), \quad f = \omega(g)$$

are analogous (respectively) to the inequalities

$$x \leq y, \quad x = y, \quad x \geq y, \quad x << y, \quad x >> y$$

on real numbers x, y. Such analogies has has led to expressions such as

$$f \leq O(g), \quad f \geq \Omega(g).$$

This could lead to trouble[14] since one is next tempted to manipulate these expressions under the usual rules of inequalities. On the other hand, a fully subscripted O-expression (say) refers to a particular function and inequalities involving such an expression can be interpreted as domination and manipulated confidently: for instance, we do not consider the expression $n^2 \geq O_1(f(n))$ to be inappropriate. In short, we never use inequality symbols $\leq$ or $\geq$ with expressions that contain unsubscripted occurrences of the O-notation.
(c) One could extend the definition of complexity functions and all the associated asymptotic notation to admit multiple parameters such as $f(m, n)$. This becomes necessary when discussing complexity classes defined by several simultaneous resource bounds.

Additional Notes: See Hardy and Wright [14] for the treatment of similar notations. Knuth's original definition of the O-notation goes as follows: $g = O(f)$ if there are constants n_0 and $c > 0$ such that for all $n > n_0$, $|g(n)| \leq c \cdot f(n)$. Our definition departed from Knuth (who is closer to the classical precedents) in that we use $g(n)$ instead of the absolute value of $g(n)$ in the above inequality. Our choice seems to be sufficient for most applications and easier to use. Like Knuth, however, we do not take the absolute value in the Ω-notation; Knuth explains the asymmetry by thinking of 'O' as referring to the neighborhood of zero and 'Ω' as referring to a neighborhood of infinity. For an updated discussion of these notations, see [34] [3] [12].

1.4 Size, Encodings and Representations

Complexity is a function of the size of the input. In the natural setting of certain problems, a correct choice for the size parameter is often not obvious. For example, let D be the set of square matrices with rational entries. An $n \times n$ matrix M in D has three candidates for its size: its dimension n, the number of entries n^2, and the total number of bits to represent all the entries. The convention (A) above removes this problem: if the matrix M is encoded as a string w over Σ, then we define the size of M to be $|w|$. In general, let D be any (mathematical) domain (such as matrices, integers, finite graphs, etc). An *encoding* of D is a function $e : D \to \Sigma^*$ (for some alphabet Σ) that is one-one. Relative to e, the *size* of $x \in D$ is simply defined as $|e(x)|$.

The use of encoding solves the problem of defining size but it introduces other issues. In particular, the complexity of the problem depends on the choice of encoding. We may be willing to accept two different encodings of the same mathematical problem as really two distinct computational problems, but that is perhaps too easy a way out. This is because two encodings may be different for rather trivial reasons:

[13]For instance, viewing "$f = O(g)$" as a binary relation, then reflexivity ($f = O(f)$) and transitivity ($f = O(g), g = O(h)$ imples $f = O(h)$) holds. Even anti-symmetry holds: $f = O(g), g = O(f)$ imples $f = \Theta(g)$.
[14]As indeed has happened in the literature.

for instance, suppose each input x to a problem P is encoded by a word $e(x) \in \Sigma^*$, and the complexity of P under the encoding e is $f(n)$. For any constant $k > 0$, we can choose another encoding e' such that for each input x, $|e'(x)| \leq \frac{|e(x)|}{k}$ (how?). With reasonable assumptions, we see that the complexity of the problem under e' is $g(n) = f(n/k)$. In this sense $g(n)$ and $f(n)$ are essentially the same.[15]

The next problem with encodings is that $e : D \to \Sigma^*$ may not be an onto function so a certain word $w \in \Sigma^*$ may not encode any element of D. Let us call w *well-formed* if it does encode some x in D; otherwise w is *ill-formed*. Should we assume that the algorithm need only restrict its attention to well-formed inputs? If so, the complexity function becomes undefined at those values of n where there are no well-formed inputs of length n. (For example, if D is the set of square boolean matrices and e encodes a boolean matrix in row-major order i.e. listing the rows where successive rows are separated by a marker symbol, then all well-formed inputs have length $n^2 + n - 1$.) To simplify things somewhat, we make the following decision:

(C) All words in Σ^*, well-formed or not, are possible inputs.

This decision is inconsequential if the set of well-formed words can easily be recognized: in many problems, this is indeed the case. Otherwise, given an algorithm that works on only well-formed words, we modify it to work on all inputs simply by attaching on the front-end a 'parsing phase' to screen the inputs, rejecting those that are not well-formed. If this parsing phase is expensive relative to the actual computation phase, then the complexity of this problem may turn out to be an artifact of the encoding. In section 5.3 we shall see examples where the parsing complexity is intrinsically high. In any case, the abandonment of (C) should lead to interesting variants of the theory.

One way to avoid an intrinsically hard parsing problem is to generalize the notion of encodings by allowing several words to represent the same object. More precisely, a *representation* r of a domain D over Σ is an onto partial function

$$r : \Sigma^* \to D.$$

We call $w \in \Sigma^*$ an *r-representative* of x if $r(w) = x$. The fact that r is onto means that every object has at least one representative. Since r is a partial function, let us say $w \in \Sigma^*$ is *well-formed* or *ill-formed* (relative to r) depending on whether $r(w)$ is defined or not.

If r is also 1-1, then in fact the inverse of r is an encoding. We have two computational problems associated with any representation r:

1. The *r-parsing problem* is the problem of recognizing well-formed words.

2. The *r-isomorphism problem* is the problem of deciding when two words w, w' are r-representatives of the same element: $r(w) = r(w')$.

If r is in fact an encoding then the isomorphism problem is trivial. Thus, the use of representations has simply shifted the parsing complexity to the isomorphism complexity. This suggests that there is an inherent complexity associated with certain domains D in the sense that every representation of D has a hard parsing problem or a hard isomorphism problem (Exercises).

It is clear that a representation r cannot be arbitrary if the notion of complexity is to be meaningful: consider the following encoding of the domain D of finite graphs, and let $P \subseteq D$ be the set of planar graphs. If for $G \in D$, the first symbol in the encoding $e(G)$ is '1' iff $G \in P$, then clearly the encoding e is rather contrived and particularly ill-suited for encoding the planarity testing problem. We have no theoretical guidelines as to how representations must be restricted. Fortunately, it is not as sorry a state of affairs as we might imagine. Despite the lack of a general theory, for most problems that arise, either

[15] This seems to argue for a notation for the set of all g such that $g(n) = O(f(O(n)))$: denote this set by $\odot(f)$. Thus $5^n = \odot(2^n)$ but not $5^n = O(2^n)$. The O- and $\odot$-notations coincide when restricted to polynomials. In Complexity Theory, we should perhaps only distinguish complexity functions up to their $\odot$-order; after all complexity practitioners normally do not distinguish between 5^n and 2^n.

there is a consensus as to the natural representation to use, or else the different choices of representations are equivalent in some sense. In the remainder of this section we show how this is so.

We give an initial informal criterion for choosing between various natural representations (without actually resolving the issue of which representations are 'natural'):

(D) Let r and r' be two natural representations for a problem P. Then r is to be preferred over r' if the following holds:

(D1) The parsing problem for r is easier.
(D2) The isomorphism problem for r is easier.
(D3) The intrinsic complexity of P under r is more than that under r'.

We would apply the criteria (D1-3) in the indicated order: for instance, if the parsing problem for r and r' are equivalent, but the isomorphism problem for r is easier that for r', we would prefer r over r'. If criteria (D1-3) cannot distinguish between two representations, then it turns out that, in practice, there is no reason to differentiate between them anyway.

We have seen the reasons for (D1, D2). Observe that (D2) implies that we should prefer encodings over general representations. The intuitive reasoning for (D3) begins with the observation that some representations are more compact (or, more succinct or efficient) than others. The intuitive notion of the most compact representation of a problem seems to be meaningful, but the notion of least compact representation does not. For, it is easy to imagine representations that introduce an arbitrary amount of irrelevant data ('padding') or that are arbitrarily redundant. If a problem is encoded more compactly than another, then the complexity of the more efficiently encoded one tends to be greater (not less!). Hence (D3) rejects redundant or padded representations.

Example 3 The contrast between compact and non-compact representations is illustrated by the following results. Suppose we want to choose between two encodings of positive integers: the usual unary encoding (a positive integer n is represented by a string of n zeroes) and the 'exponential unary encoding' (n is representated by a string of 2^n zeroes). A well-known result of Minsky[23] says that every partial recursive function can be computed by a 2-counter machine (see exercises in chapter 2 for a definition of counter machines), assuming the exponential unary encoding of numbers. A less well-known result, independently obtained by Schroeppel[31] and by Frances Yao[35] says that no 2-counter machine can compute the function $f(n) = 2^n$, assuming the unary encoding of numbers. Now by criteria (D1) and (D2), there is no distinction between these two encodings. But (D3) says the usual unary encoding is preferable. ■

Despite the ad hoc nature of (D), we are able to use it in some common examples illustrated next.

1.4.1 Representation of Sets

Suppose D' is the domain of finite subsets of another domain D, and $e : D \to \Sigma^*$ is an encoding of D. We can extend e to a representation r of D': let $\#$ be a symbol not in Σ. The word

$$e(x_1)\#e(x_2)\#\cdots\#e(x_k) \tag{1}$$

is called a *standard representative* of $X = \{x_1, \ldots, x_k\} \in D'$, provided $x_1, \ldots, x_k$ are distinct. Since the order of the x_i's are arbitrary, X has $k!$ (k factorial) standard representatives. If $e(x_1), \ldots, e(x_k)$ are in ascending lexicographical order (assuming an arbitrary but fixed ordering on Σ) then (1.1) is unique and is called the *canonical encoding* of X. If e were a representation to begin with, then the standard representation of D' is still well-defined but the canonical encoding is not well-defined.

Consider the problem of recognizing membership in P where $P \subseteq D'$. Let

$$L_c = \{e'(x) : x \in P\}$$

where e' is the canonical encoding, and

$$L_s = \{w : w \text{ is a standard representative of some } x \in P\}.$$

Both L_c and L_s appear to be natural, so we attempt to distinguish them using the criteria (D). Let f_c and f_s be the complexity of L_c and L_s (respectively). Under reasonable assumptions, it is seen (Exercise) that f_c and f_s are related as follows:

$$f_c \leq f_s + f_0 \tag{2}$$

$$f_s \leq f_c + f_1 \tag{3}$$

where f_0 is the complexity of deciding if a word of the form (1.1) is in ascending order, and f_1 is the complexity of sorting a sequence of words $e(x_1), \ldots, e(x_k)$ given in the form (1.1) into ascending order. Now in typical models of computation and complexity measures, we have

$$f_0(n) = O(n) \text{ and } f_1(n) = O(n \log n).$$

If f_c and f_s are $\Omega(n \log n)$, then (1.2) and (1.3) implies that $f_c = \Theta(f_s)$, i.e., they are indistinguishable in the sense of (D). This is a reassuring conclusion: it does not matter which representation of sets is used, provided L_c and L_s have large enough complexity. We will adopt the convention:

(E) Sets are given by their canonical or standard representation.

1.4.2 Representation of Numbers

The usual choice for representing natural numbers $1, 2, 3, \cdots$ is the k-ary ($k \geq 1$) notation over the alphabet $\Sigma = \{0, 1, 2, \ldots, k-1\}$. Clearly the unary ($k = 1$) notation is an encoding but for $k > 1$, we only have a representation since a prefix string of zeroes does not affect the value of a k-ary number. In this latter case, we can get an encoding by restricting the k-ary numbers to those that begin with a non-zero symbol. Note that criteria (D1-2) do not distinguish the various choices of k, so we need to test with (D3).

Consider the problem of recognizing a set P of natural numbers (such as the set of primes: $2, 3, 5, 7, 11, 13, \cdots$) Let $L_k \subseteq \{0, 1, \ldots, k-1\}^*$ be the set of k-ary numbers in P and f_k be the complexity of L_k. Consider the k- and l-ary encodings for any fixed $k, l > 1$. Then

$$f_k(n) = O(f_l(n)) + O(n^2). \tag{4}$$

This comes from the fact that (using reasonable computational models) there are algorithms converting between k-ary and l-ary notations that run in $O(n^2)$ time. Furthermore, for all natural numbers m, if v and w are k-ary and l-ary numbers (respectively) encoding the same number m, then their lengths are related by $|v| = \Theta(|w|)(= \Theta(\log m))$. (Exercise) We conclude that all k-ary notations ($k > 1$) are indistinguishable in the sense of (D) for problems with complexity $\Omega(n^2)$.

Unfortunately the above result does not hold for the unary notation. We note an exponential discrepancy between unary and k-ary ($k \geq 2$) notations: if u (in unary notation) and v (in k-ary notation, $k > 1$) represent the same integer, then $|u| = \Theta(k^{|v|})$. Therefore to convert a k-ary number v to a unary number takes $\Omega(k^{|v|})$ time since this is the time just to write each output symbol. Therefore, if f_k is subexponential (i.e., $f_k(n) = \omega(c^n)$ for any $c > 0$) then f_1 is a slower (*sic*) growing function than f_k. (What if f_k is at least exponential?) Criterion (D) then says we must reject the unary notation. In conclusion, the following convention will be used.

(F) Integers are encoded in k-ary for some $k > 1$.

There are other representations of numbers besides k-ary notations. For example, from basic number theory we know that each number greater than 1 has a unique decomposition into a product of powers of distinct prime numbers. We may encode such a number by a sequence of non-negative integers (say in binary notation) $p_1\#x_1\#p_2\#\cdots\#p_k\#x_k$ where $p_1 < p_2 < \cdots < p_k$ are primes and each $x_i \geq 1$. This sequence represents the number $p_1^{x_1}p_2^{x_2}\cdots p_k^{x_2}$. Multiplication under this encoding is linear time, but addition seems hard. It is not easy to dismiss this notation as 'unnatural' (a number theorist may not think so). We may then ask if addition is intrinsically harder than multiplication under this encoding, a curious reversal of Cobham's question in Section 1. This points out that the choice (F) at this point of our understanding is somewhat ad hoc.

An equally reasonable alternative to (F) would be to assume the *k-adic* notation: for each $k \geq 1$, we have an isomorphism between the strings over $\Sigma = \{1, \ldots, k\}$ and the non-negative numbers where the correspondence is given by

$$a_0a_1\cdots a_n \in \Sigma^* \leftrightarrow \sum_{i=0}^{n} a_i k^i (n \geq 0).$$

In other words[16], k-adic notation differs from k-ary notation only in its use of the integers $\{1, \ldots, k\}$ instead of $\{0, \ldots, k-1\}$. The k-adic notation avoids the well-known problem of non-uniqueness of k-ary notation.

1.4.3 Representation of Graphs

In graph theory, two different graphs g and g' are usually identified if they are isomorphic, i.e., the actual identity of the nodes is considered unimportant. In Computer Science, we normally distinguish between g and g'. To emphasize this distinction we say that graph theory treats *unlabeled graphs* while Computer Science treats *labeled graphs*. In this book, the word *graph* always denotes an undirected, labeled graph; we permit self-loops but not multiple edges in our graphs. There are three well-accepted methods of representing graphs: (1) Adjacency matrices. (2) Adjacency lists. (3) Edge lists (i.e., lists of unordered pairs of vertices). It is not hard to see that these three representations are interconvertible in time $O(n^2)$ in most reasonable models, where n is the number of nodes in the graph. This amounts to a linear time conversion for dense graphs (i.e., with $\Omega(n^2)$ edges) but in any case at most quadratic time. Thus criterion (D) justifies the following.

(G) Graphs are represented using any of the three representations of adjacency matrix, adjacency list or edge list.

Now consider the problem of representing unlabeled graphs. We can use a representation where each encoding $e(g)$ of a labeled graph g (using any of the above 3 methods) is said to be a representative of its unlabeled counterpart G. Thus the unlabeled graph G has $n!$ representations. The isomorphism problem for this representation is the well-known problem of *graph isomorphism* for which no polynomial time algorithm is known. So this is somewhat unsatisfactory. On the other hand, we can define an encoding of G by using a *canonical labeling* of G: let g be the labeled version of G such that $e(g)$ is lexicographically minimal among all other labelings of G. There is theoretical evidence that the parsing problem in this case is hard, and using criteria (D1), this canonical encoding should be rejected in favor of the previous representations. Perhaps it is not our inability to find representations with easy isomorphism problems or encodings with easy parsing problems, but that such representations simply do not exist. This suggests the problem of classifying mathematical domains (using the tools of mathematical logic) according to 'the inherent complexity of their representations'. Other domains whose representations have apparently large complexity include many groups and rings. Specifically, the representation of multivariate polynomials (a basic example of rings) is an important one with repercussions in the attempts to classify the complexity of algebraic problems such as polynomial factorization.

[16]The term 'unary' as it is normally used is really a misnomer: it should be 'unadic'. With this warning, we will perpetrate the abuse. It seems that Quine [26] is one of the first to use this notation.

In summary, in this section we made a critical decision (C) to allow all inputs, and we gave some informal criteria for how to use our theory (namely, how to represent a problem so that the notions of size has some natural meaning). As for decision (C), we are well-aware of its shortcomings. We will see instances where we would like to analyze the complexity of an algorithm A relative to some proper subset of possible inputs. For instance, if A is used to operate on the outputs of another algorithm A' (this happens when we study reducibilities, where we use A' reduce another problem to the problem solved by A). As for the questions of representing problems, we have seen that for many problems with sufficiently large complexity (in the cases examined, complexity of $\Omega(n^2)$ is enough) any of a number of natural representation are equivalent (relative to criterion (D)). *In the rest of this book, when we discuss natural problems involving numbers, sets or graphs, we normally will not be explicit about their representation because of the preceding conclusions* (*assumptions* (E),(F) *and* (G)). For the interested reader, the following is a very incomplete list of additional references on the relationship between representation and complexity: [33], [22], [19].

1.5 Models of Computation

The theory of computability [28] begins with the fundamental question:

(1) What is the concept of computability?

For definiteness, assume we ask this question of number theoretic functions. Naturally, 'computable" must be defined in the context of some suitable formal system F of computing devices. It must be emphasized that we require F to be "sufficiently general" in the sense that any function that we all intuitively understand to be computable should be computable within the formalism of F. For instance, we intuitively believe that the multiplication of two numbers, the function $f(n)$ that produces the sum of the first n primes, the function $f(n) = \lfloor \int_{a=0}^{n} x^3 dx \rfloor$ should be computable. Logicians interested in the foundations of mathematics introduced different formalizations of F. The *equational calculus* of *Herbrand* and *Hilbert*, the *λ-definable functions* of *Church*, the *μ-recursive functions* of *Kleene* and *Gödel*, and the machines of *Turing* are among these formalisms. For each F, we now have the concept of F-computable functions. It turns out to be natural to extend the concept to *partially F-computable functions*. The pleasant surprise is that for any two of these formalisms F and F', one can prove that a function is (partially) F-computable if and only if it is (partially) F'-computable. These discoveries are elevated into a general law called *Church's thesis*[17] [4]: *for all general computational formalisms, the concepts of computable and partially computable are invariant.* We have now provided a satisfactory answer to (1).

Furthermore, we conclude that all partially computable functions can effectively be enumerated (since we can clearly list, say, all Turing machines and conclude from Church's thesis that all partially computable functions are thereby named). It is not hard to use an older diagonalization argument due to *Cantor* to conclude that there exists an uncomputable function. In 1931, Gödel[10] showed that the validity of statements in number theory is undecidable. This landmark discovery demonstrates that uncomputable functions not only exist, but occur naturally.[18] Such discoveries give rise to a central theme of computability theory:

(2) Which functions are computable and which are not?

Note that this question makes sense because of Church's thesis. Extending this fundamental distinction, logicians went on to classify the uncomputable problems into 'degrees of uncomputability'.

[17] Also called the Church-Turing Thesis. Note that, as "discoveries", we have a collection of descriptive statements. But as a law, it carries a prescriptive weight – henceforth any new formalism F' that diminishes or enlarges the class of computable functions is inadmissible.

[18] It seems that the existence of such functions per se seems uninteresting unless one can show 'natural' examples. There is an analogous situation in Complexity Theory: although we already know from the work of Hartmanis that there are arbitrarily hard-to-recognize languages, greater interest was generated when Meyer and Stockmeyer showed that such languages occur naturally. See chapters 5 and 6. Another recent example in logic is the work *Paris-Harrington* in combinatorial number theory.

Unfortunately, they rarely try to classify the computable problems. In a sense, computability theory and Complexity Theory are really one subject: the latter does for the computable problems what the former does for the uncomputable ones. Complexity theory draws many of its methods (e.g., diagonal arguments) and concepts (e.g., reducibilities) from computability theory. Many phenomena at the uncomputable levels are mirrored at the lower levels. However, many of these phenomena become considerably more subtle at the lower levels. For instance, many questions that have been solved at the higher level remain open at the lower level. In this and the next section, we shall examine the complexity theoretic versions of (1) and (2).

Our view must necessarily be richer because researchers noticed that not all *forms of computation* are equivalent. The word 'computational form' is advisedly chosen in this context; it is intended to capture the concrete versions of computational models, before any attempt to put them into equivalence classes.[19] And yet, researchers also noticed that many forms of computation *are* equivalent. Here, "equivalence of computational form" is somewhat vague but in each concrete situation, we can make correspondences across forms. For example, 'time' resource can be identified in two computational forms and some suitable notion of equivalence (mutual simulation in polynomial time) defined. This not-all-forms-are-equivalent and many-forms-are-equivalent phenomena at first appear confusing. The former says that we have lost Church's thesis; the latter suggests that there might be some hope. Before pointing a way out of this paradise-lost, we need to sharpen our concept of computational form. Let us begin with examples of computational forms (this list is incomplete and includes computational forms that are not "general" in the sense demanded of F-formalisms above):

> Turing machines, storage modification machines, arrays of finite automata, pushdown automata, finite state automata, random access machines, vector machines, aggregates, formal grammars, various proof systems, lambda calculus of Church, μ-recursive functions of Kleene and Gödel, Herbrand and Hilbert's equational calculus, Post's canonical systems, Markov algorithms, Shepherdson and Sturgis' register machines, random access stored programs of Elgot and Robinson, Elementary Formal Systems of Smullyan.

The reader is not expected to know any in this list. We propose to characterize computational forms along two orthogonal directions:

(i) *Model of computation*: this identifies the basic computational structures (control, instructions and data structures). Several computational forms will be collapsed into a common model corresponding to our intent that members of the same model use similar computational structures: thus finite state automata and Turing machines will be regarded as falling under the same model (the Turing model).

(ii) *Mode of computation*: this refers to the method of using the computational structures to define computation. For example, the Turing model can compute in the deterministic, probabilistic or nondeterministic mode (see next section).

A computational form is defined by specifying the model and mode. Computational models and modes are essentially independent notions in the sense that within each model we can define machines operating in any given mode.[20] However, this must be qualified because some computational models were specifically designed for particular modes and do not naturally embrace other modes. For example, grammars or production rules in formal language theory essentially compute in nondeterministic modes,

[19] Taken literally, any perceptible difference between two models should lead to distinct computational forms. But this cannot be taken too seriously either. Anticipating our "model-mode" view of the world to be discussed shortly, a computational form is a mixture of model and mode.

[20] Up till now, we have used the word 'algorithm' to describe the computing agent. When we discuss particular forms of computation, we traditionally refer to 'machines' of that form. A useful distinction (which we try to adhere to) is that 'algorithms' are abstract objects while 'machines' are their concrete representations within a form. Thus, we have 'the' high school multiplication *algorithm*, but various Turing *machines* or PASCAL *programs* can be regarded as implementing the same high-school algorithm. Another term which we regard as essentially synonymous with 'machine' is 'program'.

and do not to easily embrace some other computational modes. Or again, the Boolean circuit model[21] one single is essentially a parallel model and it is not very natural to define sequential modes for circuits. We will say that a model is *general* if it does embrace all modes. For instance, the Turing model and the pointer machine model will be seen as general. Since the concept of computational modes post-dates the definitions of these models, we sometimes take the liberty of modifying original definitions of some models to make them general. The remainder of this section is a brief exposition of some important computational models; computational modes will be treated in the next section.

We now examine the computational structures that characterize some important computational models.

(i) *Turing model*. This will be covered in chapter 2 and is the most important one for this book. Essentially each computing agent in this model has finitely many states and symbols, and a finite number 'heads' each of which scans some memory location. The memory locations have some fixed neighborhood structure (which is specified by a bounded degree undirected graph with locations as vertices). Each location can store a symbol. The simplest example of a fixed neighborhood structure is that of a total linear relation, giving rise to what is usually known as a Turing machine tape. The computing agent has a finite set of instructions specifying what to write into scanned locations and how to move its heads when in a given state scanning certain symbols.

(ii) *Pointer machine model*. This model, also known as the *storage modification machine model*, was introduced by Schönhage in 1970 [29]. Similar models were earlier proposed by *Kolmogorov and Uspenskiĭ*, and by Barzdin and Kalnin'sh. Like Turing's model, this uses finite state control to determine the step-by-step execution of instructions. The essential difference comes from having a varying neighborhood structure that is represented by a fixed out-degree (arbitrary in-degree) directed graph G. The edges (called *pointers*) of G are labeled by symbols from a fixed set Δ. Every node has outdegree $|\Delta|$ and the labeling establishes a bijection between Δ and the set of edges going out from any node. One of the nodes is designated the *center*. Then each word $w \in \Delta^\star$ is said to *accesses* the node obtained by following the sequence of pointers labeled by symbols in w, starting from the center. In Figure 1.1, the center is indicated by an arrow with no label.

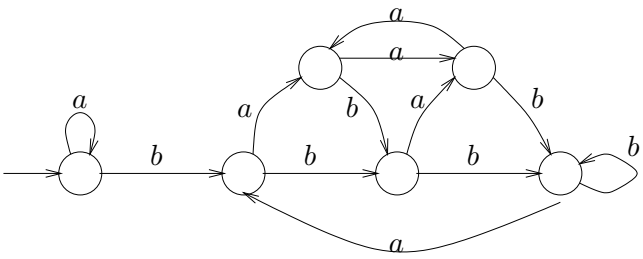


Figure 1.1: Pointer machine Δ-structure ($\Delta = \{a, b\}$). The center is indicated by the arrow without any label.

For instance the empty word ϵ accesses the center. Now a *pointer machine* (for any set Δ) is a finite sequence of instructions of the following types:

(i) $w := w'$ (assignment statement)

(ii) $w :=$ **new** (node creation statement)

(iii) **if** $w' = w$ **goto** L (branch statement)

[21] Actually, circuits raises another more annoying issue for our attempt to classify models – it is what is known as a 'non-uniform' model. There are many easy (though not necessarily natural) ways to make it uniform like Turing machines. We will assume this step has been taken if necessary.

(iv) Choose (L, L') (choice statement)

Here w, w' are words in $\Delta^\star$ and L, L' are natural numbers viewed as labels of instructions. Initially G comprises only of the center, with each pointer from the center pointing back to itself. Let us now give the semantics of these instructions. Suppose G is the graph before executing an instruction and G' is the graph after.

(i) If w' accesses the node v in G then both w and w' access v in G'. This is achieved by modifying a single pointer in G.

(ii) We add a new node v to G to form G', and w now accesses v. Furthermore, each pointer from v points back to itself.

(iii) If both w' and w access the same node, then we branch to the Lth statement; otherwise we execute the next instruction in the normal fashion. Here $G = G'$.

(iv) The machine can go to the Lth or the L'th instruction. This corresponds to computational choice.

Although no explicit information is stored in nodes, we can simulate the same effect as follows: designate certain special nodes $a_1, \ldots, a_k$ as representing k distinct symbols, and one of the labels $\sigma \in \Delta$ is designated the 'content' label. By convention, we say that a node v 'contains' the symbol a_i $(i = 1, \ldots, k)$ if the σ-pointer from v points to a_i. Using this, we can easily represent a Turing machine tape whose cells contain the symbols $a_1, \ldots, a_k$. We now make up some convention for the input and output words. We leave as an exercise the efficient simulation of Turing machines by pointer machines and vice-versa. Just as the Turing model, this class is capable of many interesting variations: first of all, we should probably expand the repetoire of instructions in non-essential ways (e.g., allowing the ability to branch on a general Boolean combination of equality tests). But non-trivial extensions include allowing each node to store an arbitrary integer, and providing basic arithmetic or bitwise operations on a single instruction.

(iii) *Random access machines (RAM).* This is perhaps the closest model to real world computers. Again, each computing agent here consists of a finite set of instructions executed under a finite state control. The difference here is the use of *registers* to store an arbitrarily large integer. The contents of registers can be tested for zero, compared to each other, have arithmetic operations performed on them, all in one step. Registers have unique integer addresses, and registers are accessed in instructions by (i) specifying its address explicitly in the instruction or (ii) indirectly by specifying the address of a register which contains its address.

The reader will appreciate our earlier remark that a computational model is usually characterized by its data structures: Turing machines have fixed topology data structures (called tapes) where each location stores a finite symbol. The data structures in pointer machines have variable topology while random access machines have fixed topology but non-atomic memory locations. Wagner and Strong is an early effort to characterize abstract computational models,

1.6 Modes of Computation: Choice and Parallelism

Despite the multiplicity of computational forms, it is possible to isolate a small number of equivalence classes among them: each equivalence class corresponds to basic and intuitive concepts of computation. We identify each equivalence class with a particular *mode of computation.*[22] As we shall see, *within each equivalence class, we can formulate a version of Church's Thesis.*

[22]Our notion of modes is partly inspired by the 'computational types' by Hong Jia-wei. The latter concept appears to be a mixture of computational models and computational modes.

We distinguish two dimensions of modes. In the first dimension, the computation is either *deterministic* or it may have *choice*. In the second dimension, the computation is either *sequential* or *parallel*. So the four basic classifications of modes are: deterministic-sequential, deterministic-parallel, choice-sequential and choice-parallel. We can have varying degrees of choice (resp. parallelism), with determinism (resp. sequentialism) being the limiting case of *no* choice (resp. *no* parallelism). Therefore, in the above four classes of modes, only the deterministic-sequential mode is uniquely determined: the other three classifications can be further refined. We next indicate characteristics of such modes.

1.6.1 The Fundamental Mode of Computation

The prototypical computation is in the deterministic-sequential mode. Indeed, this mode is usually the only one recognized in theory of computability; of course, the other modes turn out to be no more general in the sense of Church's Thesis. We shall therefore call this the *fundamental mode* of computation. A computation (by a machine M) operating in this mode has the following typical features:

(a) There is a bounded number of internal states, of which one is designated the 'current state'.

(b) There is a finite but unbounded number of memory locations (chosen from an infinite reserve) that are used at any moment. By this we mean that each location at any moment contains a symbol from an alphabet Σ and a location is considered *unused* if the symbol it contains is a certain predesignated 'blank' symbol. Each location has a bounded number of other locations which are its 'neighbors'.

(c) A bounded number of the memory locations are 'scanned' at any instant.

(d) At each time moment, a unique instruction (which only depends on the symbols scanned in (c) and the current internal state of (a)) is executed. The instruction can change the contents of the scanned locations, the current state, and the set of scanned locations. The new set of scanned locations must be neighbors of the previously scanned ones.

(e) There are fixed conventions for starting, halting, error handling, inputs, and outputs.

It is interesting to compare (a)-(e) with the well-known characterization given in chapter 1 of Rogers [28]. The reader is encouraged to identify our abstract description with the execution of a program on a real computer – most of the above can have a reasonable interpretation (the assumption of an unbounded number of memory locations in (b) is most problematic in this comparison). All the 'bounded numbers' in (a)-(c) are bounded by constants that solely depend on M . At each instant, the *instantaneous description* (ID) of M consists of the current state, the set of scanned locations and the contents of all the used memory locations. Thus we can regard the instructions described in (d) as transforming ID's. If I and I' are ID's such that I is transformed in one step to I', we write $I \vdash I'$, and say I *directly derives* I'. The characterization of M above is deterministic because once the computation is started, the succession of ID's is uniquely determined. The sequence of ID's (possibly infinite) is called a *computation path*. M is sequential because at each time unit, the contents of only a bounded number of locations can be scanned and modified.

Assumption. We have tacitly assumed time and space are discrete above. More generally, *all resources are discrete*. When we discuss non-fundamental modes with more than one computing agent, we will further assume time is *synchronized or global*. This is assumed throughout this book.

To facilitate discussion of the next mode of computation, we elaborate on (e) somewhat. Let us assume that M is a machine for recognizing its inputs. There are two designated internal states designated as *rejecting* and *accepting*. The computation stops iff the machine reaches either of these states. If π is any computation path we define the predicate $\text{ACCEPT}_0(\pi)$ to equal 1 if π is finite and the last ID in π contains the accepting state. Otherwise $\text{ACCEPT}_0(\pi) = 0$. Then M is said to *accept* an input x if $\text{ACCEPT}_0(\pi) = 1$, where π is the computation path of M on input x.

1.6.2 Choice Modes

Let us now turn to computational modes that replace 'determinism' with 'choice'. This involves replacing (d) above and (e) above.

(d)* At each time unit, a set (which only depends on the symbol scanned and the current internal state) of next instructions is 'executable'. The machine can 'choose' to execute any one of these instructions. As in (d), each instruction can change contents of locations, current state and set of scanned locations.

(e)* A suitable 'choice aggregation' function is used to determine the output, as illustrated next.

For simplicity, assume that there is only a choice of two next instructions to execute. On input x, we now have a binary *computation tree* $T = T(x)$ whose nodes are ID's defined as follows. The root of T is the initial ID (this is a function of x and M). If I is a node of T, and I_1 and I_2 are the ID's which result from I by executing the two executable instructions, then in T the node I has I_1 and I_2 as its two children. A maximal path in T from the root (terminating in a leaf if the path is finite) corresponds to a computation path.

We want to define the analogue of the ACCEPT_0 function above, and it should be a function of the computation tree T. We describe two main ways to do this. In the first way, we define the predicate ACCEPT_1 by

$$\text{ACCEPT}_1(T) = 1 \text{ iff there is a path } \pi \text{ in } T \text{ such that } \text{ACCEPT}_0(\pi) = 1.$$

The choice mode machine M which accepts its input x iff $\text{ACCEPT}_1(T(x)) = 1$ is said to compute in the *existential-sequential* mode (or more commonly known as the *nondeterministic* mode).

Another way to define acceptance is to assign probabilities to each node in T. Each node I at level k has probability $\Pr(I) = 2^{-k}$, and in particular the root has probability 1. We define the predicate ACCEPT_2 by

$$\text{ACCEPT}_2(T) = 1 \text{ iff } \sum_I \Pr(I) > 1/2,$$

where I ranges over those leaf nodes in T with the 'accept' state. The choice mode machine M which accepts its input x iff $\text{ACCEPT}_2(T(x)) = 1$ is said to compute in the *probabilistic-sequential* mode (or more commonly known as the *probabilistic* mode). We shall see other choice modes in the book.

The existential mode of computation may appear strange at first. Yet it turns out to be an accurate model of the notion of a formal axiomatic system. With suitable conventions, a nondeterministic machine M corresponds exactly to a formal axiomatic system S: the machine M accepts an input x iff x encodes a theorem of S. The reader familiar with grammars in formal language theory will also be able to make connections with nondeterminism. Perhaps the main reason for the importance of the existential-sequential mode is its use in defining a class of problems denoted NP, which has great empirical importance. We shall study NP in chapter 3.

1.6.3 Parallel Modes

Now we consider the deterministic-parallel modes. Such modes have begun to assume increasing importance with the advent of mass-produced very large scale integrated (VLSI) circuits as well as the rapidly maturing technology of multiprocessor computers. In parallel computation, we will call the entire set of machines which can execute in parallel the *ensemble*. We shall always assume that the ensemble works *synchronously*.[23] Each computational model described in the previous section (Turing machines, pointer

[23]The study of asynchronous or distributed ensembles in discrete computations is an emerging subject that seems to give rise to a rather different theory. The emphasis there is the complexity of communication (message passing) between independent agents in the ensemble.

machines, random access machines) amounts to specifying individual computing units – this is true in general. Hence to describe ensembles, we need to specify how these units are put together. This comes from four additional design parameters:

(f) (Computing Units) The ensemble is made up of an infinite number of *computing units*, each operating in the fundamental mode (a)-(e). These units may be finite in nature (e.g. finite state automata) or infinite (e.g. general Turing machines).

(g) (Topology) The units have a connectivity (or neighborhood) structure which may be represented by a graph of bounded degree. The graph may be directed or undirected. Units can only communicate with their neighbors. The connectivity may be variable or fixed depending on the model.

(h) (Sharing and Communication) Each pair of machines which share an edge in the connectivity graph has access to a common bounded set of memory locations: simultaneous reads of this shared memory are allowed, but write conflicts cause the ensemble to halt in error. Note that this shared memory is distinct from the infinite local memory which individual machines have by virtue of (b).

(i) (Activation, Input and Output) There is a designated 'input unit' and another 'output unit'. Input and output conventions are relative to these two units. These two units are not necessarily distinct. Every unit is either 'active' or 'inactive' (quiescent). Initially, all but the input unit is inactive. An active machine can activate its quiescent neighbors. Thus at any moment, a finite but unbounded number of machines are active. The ensemble halts when the output unit enters certain special states (accept or reject).

It is easy to suggest further variations of each of the aspects (f)-(i) above; the literature contains many examples. For instance, unlike the suggestion in (f), not all the individual machines need to be identical. Or again, the sharing need not be local as suggested by (h) if we postulate a global memory accessible by all machines. This is sometimes called the *ultra-computer* model, a term popularized by Jack Schwartz. Another very important practical issue arising from shared memory is the resolution of *reading* and *writing conflicts*. Instead of our stringent condition on disallowing writing conflicts in (h), there are three more liberal approaches:

> We may allow simultaneous writing to a location provided all processors write the same value, or we may have no restrictions on simultaneous writes but say that some arbitrarily chosen processor is successful in writing, or we may say that the smallest numbered processor will be successful.

Another extreme case of (h) is where the machines may have almost no common memory except for a flag associated with each channel or port. The channel or port of a machine may be specific (connected to a particular machine) or non-specific (any machine may try to communicate through that port).

In view of the large number of variations in parallel models, it is all the more surprising that anything significant or unifying can be said about computations in this mode. It is evidence that we are correct in designating essentially one computational mode to all these.

We remark that the literature sometimes regards choice modes as 'parallel' modes since we can think of the various choices of executable instructions as being simultaneously executed by distinct copies of the original machine (reminiscent of 'fork' in UNIX$_{TM}$). However we will not use 'parallel' in this sense. In true parallelism (unlike choice) processes will be able to communicate during their simultaneous computations.[24] Consequently, the aspects (f)-(i) of parallel machines are irrelevant for the choice-sequential mode. More importantly, choice computation is not to be confused with parallelism because acceptance by choice computation is done post-computation (by a fixed evaluation mechanism that is independent of the machine).

[24]The reader will appreciate this distinction if she attempts to show 'in an obvious manner' that the class *NP* (to be defined in chapter 2) is closed under complementation and faces up to the inability for different branches of a nondeterministic computation to communicate.

The final class of modes, choice-parallel ones, can easily be conceived after the preceding development. For instance, if there are m active computing units at some moment in a choice-parallel computation, each unit having a branching factor of 2, then the entire ensemble has a branching factor of 2^m. We shall not elaborate on this mode of computation.

In conclusion, we see that the computer scientists' answer to the fundamental question (1) of section 5 is quite different from the one the logicians obtained. This is captured in the notion of computational modes. We next turn to the computer scientists' view of question (2) in section 5.

1.7 Tractability and some Computational Theses

It has been observed in the computing milieu that many problems such as the *Perfect Matching Problem* in graph theory have polynomial time algorithms (even though the initial mathematical characterization of 'perfect matchings' only suggested exponential algorithms). In contrast, certain computational problems such as the *Traveling Salesman Problem* (TSP) defy persistent attempts to design efficient algorithms for them. All known algorithms and suggested improvements for these algorithms still have, in the worst case, super-polynomial[25] running time. Typically, this means an exponential running time such as 2^n. The obvious question is whether a problem such as TSP is intrinsically super-polynomial. The gap between an exponentially growing function (say) 2^n and a polynomial function (say) n^2 is quite staggering: on a computer executing 10^6 instructions per second, an input of size 200 would require 2^{200} (resp., 200^2) steps or more than 10^{46} years (resp., less than a second) of CPU time. Of course, the difference increases dramatically with increasing n. This unbridgeable gap between polynomial functions and exponential functions translates into a clear distinction between problems with polynomial complexity and those with super-polynomial complexity. Super-polynomial problems are 'practically uncomputable' (except for small values of n) even though they may be computable in the sense of computability theory.[26] The phenomenon appears very fundamental and researchers have coined the term 'infeasible' or 'intractable' to describe these difficult problems:

(H) A problem is *tractable* if it has polynomial complexity; otherwise the problem is *intractable*.

Cobham [5] and Edmonds[8] were among the first harbingers of the tractable-intractable dichotomy.[27] Let us note an obvious criticism of (H). While we may agree that super-polynomial problems are intractable, we may not want to admit all polynomial problems as tractable: a problem with complexity of n^{37} hardly deserves to be called tractable! In practice, we seldom encounter such high degree polynomials. On the other hand, it is difficult to decide on theoretical grounds when polynomials become intractable, and hence (H).

Note that (H) is really a general principle in the sense that we have not restricted it to any particular model, mode of computation or computational resource. For instance, if we consider the Turing model operating in the deterministic mode, and use space as our resource, then this principle tells us that the tractable problems are what is normally called *PSPACE*. (If we operate in the nondeterministic mode, a well-known result of Savitch which we will encounter in chapter 2 says that precisely the same class of problems are tractable.) One reason why (H) is attractive is that it is a very robust concept: tractability is easily seen to be invariant under minor modifications of the computational models. A far-reaching generalization of this observation is the tractability thesis below which basically says that the concept is robust enough to withstand comparisons across totally different computational models, *provided we compare these models using the same computational mode and using corresponding resources.* There

[25] A complexity function f is *super-polynomial* if $f(n)$ is not $O(n^k)$ for any fixed value of k. A complexity function f is said to be (at most) *exponential* if $f(n) = O(1)^n$, *double exponential* if $f(n) = 2^{O(1)^n}$, etc.

[26] This remark must be balanced against the fact that very often, naturally occurring instances of an intractable problem can be solved by efficient special methods.

[27] Edmonds called an algorithm 'good' if it runs in polynomial time in the fundamental mode. In [9] he informally introduced the class *NP* by describing such problems as having 'good characterization'.

is a problem with the last assertion: what do we mean by 'corresponding resources'? For this to be non-vacuous, we must be sure that there are certain computational resources which are common to every conceivable model of computation. In the actual models that have been studied in the literature, this is not problematic. In any case, we postulate three fundamental resources in every computational model:

(I) For every computational model in a given computational mode, there is a natural notion of *time, space* and *reversal* resources.

Time and space have intuitive meaning; reversal on Turing machines is intuitively clear but it can be formulated for all other models as well. For postulate (I) to be meaningful, there ought to be axioms that these measures obey (otherwise, how do we know if 'time' on one model-mode is really to be equated with 'time' on another model-mode?). For example, we expect that "space is at least as powerful as time". We said that the Boolean circuit model[28] does not naturally yield a notion of sequential mode. One possible approach is to use what is known as a 'straight-line program' representation of a circuit (that is, we list the gates in a linear sequence, where each gate must have its inputs coming from earlier gates or from the external inputs). Then (sequential) time is simply the length of this straight-line program. More generally, Hong has suggested that sequential time is simply the total number of individual actions that take place inside a computation (although the circuit example blurs the distinction between the computing device and the computation itself). We leave such issues as matters of further research. In any case, when we say 'resources' below, they could refer to composite resources such as simultaneous time and space.

We are now ready to assert what may be called the "polynomial analogue of Church's Thesis".

(J) (*Tractability Thesis*) For each computational mode and each computational resource, the notion of tractability is invariant over all models of computation.

For comparison, we also formulate a version of Church's Thesis here.[29]

(K) (*Church's Thesis*) The notion of computability is invariant over all models of computation.

When restricted to the time resource in the fundamental mode, thesis (J) is sometimes called Cobham's thesis after Cobham who made the first observations about its invariance properties. Although the thesis (J) has not been tested in its full generality (especially, in some of the newer modes), it is useful as a working hypothesis or as a research program.

In all our theses, there is an implicit restriction to 'reasonable' models of computation. Or at least, we intend to call a model 'unreasonable' if it violates such principles. Thus we might appeal to the tractability theses to reject certain models of computations as unreasonable. For instance, it is clear that if we severely restrict our computational model we could violate (J). Likewise, it is not hard to imagine a model too powerful to respect the tractability thesis. This is not surprising, but simply points out that our theses need *a priori* conceptual analysis in order to establish them firmly. Subject to this qualification, and provided that we accept (H) and (I), we see that (J) has mathematical content in the sense that we may verify it for each proposed model of computation. On the other hand, (J) is not a mathematical theorem, and (as Emil Post remarked, concerning Church's thesis) "it requires continual verification". In summary, we say that the thesis serves both a normative as well as descriptive function.[30]

Thesis (J) is the consequence of another more general *polynomial smearing phenomenon* that says, with respect to a fixed resource, all reasonable computational models computing in a fixed mode are equally

[28] suitably made 'uniform', see chapter 10.

[29] As Albert Meyer points out to us, it is possible to interpret Church's thesis in a much wider philosophical sense. See (for example) [16] for such discussions. The mathematical content of this wider thesis is unclear and our formulation here is, for our purposes, the generally accepted mathematical interpretation of the thesis.

[30] The same can be said of Church's thesis – we would use its normative role to dismiss someone who insists that finite automata be admitted as general computing devices. Taking Lakatos' critique of mathematical truth as a clue, we might modify Post's position and say that these theses require continual refinement.

powerful up to a polynomial factor. More precisely: say that a computational model M *polynomially simulates* another model M' *in resource* R if for every machine A' in M' which uses $f(n)$ units of resource R, there is a machine A in M which uses $O((f(n))^k)$ units of R and which solves the same problem as A'. The constant k here may depend on A', although we find that in all known models, k may be uniformly chosen (e.g. $k = 2$). We say M and M' are *polynomially equivalent* in resource R if they polynomially simulate each other.

(L) (*Polynomial Simulation Thesis*) Within each computational mode, and with respect to a given computation resource, all models of computation are polynomially equivalent.

Clearly (H) and (L) imply (J). Hong[17] has provided strong evidence for this thesis (his so-called 'similarity theorems'). We now state two conjectures and one more thesis. These relate across computational modes and resources:

(M) (*The $P \neq NP$ Conjecture*) With respect to time measure, tractability in the fundamental mode and tractability in the nondeterministic mode are different.

(N) (*Parallel Computation Thesis*) Time measure in the deterministic-parallel mode is polynomially related to space measure in the fundamental mode.

(O) (*Duality Conjecture*) The resources of Space and Reversal are duals (see text below) of each other when the simultaneous bounds on space and reversals are polynomially related.

Conjecture (M) is perhaps the most famous open problem in the subject, and we shall return to it in chapter three. Thesis (N) (proposed by Pratt and Stockmeyer [25] and Goldschlager [11]) equates the tractable problems in the deterministic-parallel mode under time resource bounds with the tractable problems for the fundamental mode under space resource bounds. See also [24]. The duality conjecture was formulated by Jia-Wei Hong. It is more technical and involves simultaneously bounding the space and reversal resources. Roughly, it says that if a complexity class is defined by Turing machines bounded by $O(f)$ space and $O(g)$ reversal simultaneously then the same class is described by $O(f)$ reversal and $O(g)$ space simultaneous bounds.

In this book we shall provide evidence for the above theses. We note the tremendous unifying power of (H)-(O): First, (H) gives us a 'world-view' by pointing out a fundamental distinction (just as the computable-uncomputable distinction in computability theory); this in turn guides the basic directions of research. Second, these theses lead us to the general principle (or metathesis):

> *The truly fundamental phenomena of Complexity Theory are invariant across computational models.*

Third, the theses (M), (N) and (O) suggest that the concepts of modes and resources are not arbitrary. Rather, their interrelation reflects an internal structure of the concept of computation that awaits discovery. This could be the most exciting direction in the next stage of Complexity Theory. It is fair to add that the mode-resource-model structure we have imposed on the corpus of the known complexity substratum may not bear careful scrutiny, or at least requires new clarifications, especially as new computational concepts proliferate.[31]

[31]Such proliferation seems as inevitable as in subatomic Physics. The challenge is to provide an adequate unified field theory of complexity theory as the world of complexity unfolds. It is also useful to remember that although Newtonion Physics is superseded, it is hardly extinct, thanks to its compact embodiment of non-extremal physical phenomena.

1.8 What is Complexity Theory?

We have given an impressionistic tour of Complexity theory in this chapter and outlined some of the methodological framework. This framework is plausible but not necessarily convincing. In any case, we sometimes refer to these assumptions (if only to hint at alternative assumptions) as *Standard Complexity Theory*. We are ready to round up with three perspectives of what the subject is about, plus a very brief historical perspective. By *complexity classes* below we mean any set of languages. In practice, these classes are defined by some well-defined and general mechanism which admits some concept of complexity.

a) The study of computational complexity is usually taken in a wider sense than that taken by this book. In particular, it usually incorporates a very large and important literature on the analysis of algorithms and data-structures. Complexity Theory there is sometimes called *concrete complexity* (dealing in 'concrete' as opposed to 'abstract' problems). However, complexity in concrete complexity is often more a property of the particular *algorithm* being analyzed, rather than a property of the *problem* that the algorithm solves. In contrast:

Complexity Theory is concerned with the intrinsic complexity of problems.

Specifically, we are interested in classifying problems according to their intrinsic complexity.

Let us probe deeper this intuitive concept of intrinsic complexity. The notion turns out to be rather subtle: one phenomenon that may arise is that there may be no fastest algorithm – for each algorithm there exists a faster one. We will see examples of such 'speed-up' phenomena. If we consider two or more computational resources, there may be inherent tradeoffs between the complexities with respect to these resources. In the face of these possibilities, we can still attempt to classify the complexity of a problem P relative to a given *hierarchy* of complexity classes: more precisely, if

$$K_1 \subseteq K_2 \subseteq K_3 \subseteq \cdots \tag{5}$$

is an non-decreasing (not necessarily strict) sequence of complexity classes, we say that P is in the ith level of this hierarchy if $P \in K_i - K_{i-1}$ ($i = 1, 2, \ldots$ and $K_0 = \emptyset$). Note that this abstract method of classification is a really profound departure from the original idea of comparing growth rates of complexity functions. The hierarchy $\{K_1, K_2, \ldots\}$ that replaces complexity functions need not have any clear connection to complexity functions. Let us call any hierarchy (1.5) used for this purpose a *complexity ruler*. Once we agree to classify problems relative to complexity rulers, some very fruitful directions in complexity theory become possible. Such classifications are studied in chapter 9.

One complexity ruler that is used commonly is the *canonical ruler* formed from the canonical list (see chapter 2, section 3), but omitting the class $PLOG$:

$$\begin{aligned} DLOG \subseteq NLOG \subseteq P \quad &\subseteq \quad NP \subseteq PSPACE \\ &\subseteq \quad DEXPTIME \subseteq NEXPTIME \subseteq EXPSPACE \end{aligned}$$

One of the first things a complexity theorist does on encountering a computational problem is to find out where it fits in this canonical ruler. Another ruler is this: a logarithmico-exponential function (for short, L-function) $f(x)$ is a real function that is defined for values $x \geq x_0$ (for some x_0 depending on f) and is either the identity or constant functions, or else obtained as a finite composition with the functions

$$A(x), \quad \log(x), \quad e^x$$

where $A(x)$ denotes a real (branch of an)[32] algebraical function. A classical result of Hardy[13] says that if f, g are L-functions then either $f = \Theta(g)$, or f dominates g, or g dominates f. Hence the family R_0 consisting of the classes $DTIME(\Theta(f))$ where f is an L-function forms a complexity ruler. (Of course,

[32]For instance, the polynomial $p(x) = x^2$ defines two branches corresponding to the positive and negative square-root functions. Generally speaking, a polynomial $p(x)$ of degree d defines d functions corresponding to the d roots of $p(x)$ as x varies. These functions are called 'branches' of $p(x)$.

the ruler is doubly infinite in both directions, dense and all that.) In practice, when complexity theorists discuss the time complexity of concrete problems (like matching, multiplication, etc), they are implicitly using the ruler R_0. For instance, although multiplication has time complexity $t(n)$ that satisfies the inequalities $n \leq t(n) = O(n \log n \log\log n)$, there may in fact be no single function $f(n) \in R_0$ such that $t(n) = \Theta(f(n))$; nevertheless it makes sense to talk about the sharpest upper (or lower) bound on multiplication relative to the ruler R_0.

Actually, the use of rulers to measure complexity can be further generalized by introducing the concept of reducibility among languages (chapter 4). The inclusion relation can be replaced by the reducibility relation.

b) The discussion of encodings and models of computations leads us to conclude that our theory, necessarily becomes quite distorted for problems with low-level complexity (say $o(n^2)$ time). On the other hand, the concept of computational modes arises because we are also not very concerned with high-level complexity (otherwise they all become equivalent by Church's thesis (K)). In short:

Complexity Theory is concerned with medium-level complexity.

Here (if one has to be so precise) one may identify as medium-level those problems which are elementary or primitive recursive.[33] It is this concern for medium level complexity that makes the fundamental dichotomy (H) a meaningful one. Indeed a central concern of the theory is to classify problems as tractable or otherwise (with respect to any mode or resource). Our interest in medium level complexity partly justifies some of our assumptions. For instance, even though assumption (B) captures recognition problems only, many functional or optimization problems are polynomially equivalent to suitable recognition problems (chapter 3, section 2); studying complexity up to polynomial equivalence seems justifiable for medium level complexity.

c) Finally, we contend that

Complexity Theory is essentially a theory of classes of problems

as opposed to a theory of *individual* problems. No doubt, the complexity of certain important problems (graph isomorphism, multiplication, primality testing, etc) in particular models of computation has abiding interest. But the exact complexity of an individual problem is to some degree an artifact of the details of the computational model. It is only when we examine an entire class of (suitably chosen) problems that we manage to have truly invariant properties. Moreover, even when we study the complexities of individual problems, we usually aim at placing the problem in a well-known complexity class or in some level of a hierarchy. Another case we may adduce to support our claim is the existence of the tractability thesis: the class of interest there are those problems with polynomial complexity. Other examples can be provided (see also the Preface). Indeed, we regard this emphasis on complexity classes as the chief distinguishing mark of the Complexity Theory represented by this book, and hence the book title.

d) The preceding three perspectives on Complexity Theory are from an intrinsic or genetic standpoint. A historical perspective would also be greatly rewarding especially for the student of the history and philosophy of science. Unfortunately, only a few remarks can be offered here. The theory of computational complexity (which we shorten to Complexity Theory in this book) took its name from the seminal paper of Hartmanis and Stearns [15] in 1965. The earlier papers of Rabin [27] and Blum [2] are generally counted among the most influential works associated with the founding of this subject. We refer the interested reader to the volume [6] for further references as well as personal accounts from that seminal period. The work of Cook and Karp in the early 1970s profoundly changed the field by raising certain important questions; these questions though still open, play a paradigmic role that has influenced directly or indirectly much of subsequent research programs.

[33] 'Elementary recursive' and 'primitive recursive' are technical terms in recursive function theory. Section 6 of chapter 5 has a definition of elementary recursiveness.

Exercises

[1.1] Verify the assertion in the text: $g \neq o(f)$ means that there is some $c > 0$ such that for infinitely many n, $g(n) \geq cf(n)$.

[1.2] What is the distinction between $f(n) = \Omega(g(n))$ and $g(n) \neq O(f(n))$, and between $f(n) = \omega(g(n))$ and $g(n) \neq o(f(n))$?

[1.3] (i) A useful notation is $f(n) \sim g(n)$, defined here to mean

$$f(n) = (1 \pm o(1))g(n).$$

Show that if $f(n) \sim g(n)$ then $g(n) \sim f(n)$.
(ii) Suppose $x2^x = n$ holds for all n. We want to solve for $x = x(n)$. Show that $x(n) = \log n - \log\log n + O(\frac{\log\log n}{\log n})$
(iii) Conclude that $\log n - x(n) \sim \log\log n$.

[1.4] ** (i) Are there useful applications of mixed asymptotic expressions (big-Oh, big-omega, small-oh, etc)? Does it make sense to say that the running time of a machine is $\Omega(n^{O(n)})$?
(ii) More generally, work out a calculus of these mixed notations.

[1.5] Extend the asymptotic notations to multi-parameter complexity functions.

[1.6] Let $g(n) = \exp\exp(\lfloor \log_2 \log_2 n \rfloor))$ where $\exp(m) = 2^m$. Clearly $g(n) = \Omega(n^\alpha)$ and $g(n) = O(n^\beta)$ for some constants $\alpha, \beta > 0$. Give the values for α and β that are sharpest in the sense that for any $\epsilon > 0$, both the Ω- and the O-bound would not be true if $\alpha + \epsilon$ and $\beta - \epsilon$ (respectively) were used.

[1.7] Show:
(i) $H_n = \Theta(\log n)$ where H_n (the harmonic series) is defined as $\sum_{i=1}^n \frac{1}{i}$.
(ii) $\sum_{i=1}^{\log n} 2^i \log\left(\frac{n}{2^i}\right) = \Theta(n)$. [Obtain the best constants you can in the upper and lower bounds on the sums in (i) and (ii).]
(iii) $(x + a)\log(x + b) = x\log x + a\log x + b + o(1)$.

[1.8] Describe an $O(n^2)$ algorithm for converting k-ary numbers to k'-ary numbers for $k, k' > 1$. Can you do better than $O(n^2)$? *Hint:* Regard the k-ary number $a_n a_{n-1} \cdots a_0$ $(a_i = 0, \ldots, k-1)$ as the polynomial $p(x) = \sum_{i=0}^n a_i x^i$ and evaluate $p(x)$ at $x = k$ in k'-ary notation.

[1.9] Demonstrate the inequalities (2-4) in Sections 4.1, 4.2. Make explicit any reasonable assumptions about the computation model and complexity measure.

[1.10] Referring to section 4.2, show that for $k > 1$,

$$f_k(n) = O(f_1(\Theta(k^n))) + O(k^n).$$

$$f_1(n) = O(f_k(\Theta(\log n))) + O(n\log n).$$

What can you say about the relative growth rates of f_1 and f_k if f_k is exponential?

[1.11] List some properties that distinguish k-adic from k-ary notations.

[1.12] * Give a systematic treatment of the possible variations of parallel mode of computation. Extend this to the parallel-choice modes of computation.

[1.13] Verify that multiplication is indeed linear time under the suggested prime number encoding of numbers in Section 4.2. What is the complexity of addition?

[1.14] Recall the recognition problem in Section 2 for triples (x, y, z) of binary numbers such that $xy = z$. We modify this so that the relation is now $xy \geq z$. Show that the ordinary multiplication problem can be reduced to this one. If this recognition problem can be solved in $T(n)$ time on inputs of length n, give an upper bound for the multiplication problem.

[1.15] Construct a pointer machine to add two numbers. Assume that numbers are encoded in binary, and that a binary string is encoded as a linked list of nodes. Each node in this list points to one of two designated nodes called 0 or 1. Your input and output are in this format.

[1.16] Let M be a deterministic Turing machine that accepts in time-space (t, s). Show how a pointer machine can simulate M in $O(t, s)$ time and space.

[1.17] Let P be a deterministic Pointer machine that accepts in time-space (t, s). Fix a convention for input, and assume that P cannot change this input, to make it compatible with our convention for Turing machines. Show how a Turing machine can simulate M in $O(t \log s, s \log s)$ time and space.

[1.18] ** Axiomatize or give abstract characterizations of the computational models in the literature. Some work in the past have done precisely this for some models. Describe the various computational modes for in each of these models. With our new distinction between modes and models, this task requires new care.

[1.19] ** Give proper foundations for the computational theses expounded in section 7. This presumes a suitable solution of the preceding question.

[1.20] * There are many ways to represent a real algebraic number α (i.e. α is a root of some polynomial $p(x)$ with integer coefficients). (i) We can exploit the fact that real numbers are ordered and represent α as $\langle p(x), i\rangle$ if α the ith real root of $p(x)$. (ii) We can isolate the root of interest by an interval, so use the representaion $\langle p(x), I\rangle$ where I is an interval with rational endpoints containing α but no other roots of $p(x)$. (iii) We can exploit Thom's lemma that asserts that if we assign a sign $\sigma(D_i(p)) \in \{-1, 0, +1\})$ to the ith derivative $D_i(p)$ of $p(x)$ for each $i \geq 0$ then the set of real numbers a such $D_i(p)(a)$ has sign $\sigma(D_i(p))$ forms an interval (possibly empty) of the real line. In particular, if we choose $\sigma(D_0(p)) = \sigma(p) = 0$ then either the set of such a is empty or else we have identified a unique root. Hence α can be represented by $\langle p(x), \sigma\rangle$ for a suitable σ. Compare the three representations using criteria (D).

[1.21] * Consider the problem of representing multivariate polynomials whose coefficients are integers. What are the various methods of encoding? Discuss the consequences of these on the complexity of various algorithms on polynomials (a) GCD, (b) factorization. In this connection, the reader is referred to the work of Erich Kaltofen on the straight-line program representation of polynomials (e.g., [20]).

[1.22] ** Investigate the theory of computational problems (see footnote 1). In this regard, compare the 'theory of first order problems' for linear orderings in [36] (which is taken from [37]).

[1.23] ** Show that some mathematical domains D (such as unlabeled graphs) have an inherently high 'representaional complexity' in the sense that fore any representation $r : \Sigma^* \to D$, either the r-parsing or r-isomorphism problem is complex (compare the next exercise).

[1.24] ** Investigate the effects of encodings on complexity. On a methodological level, we could say that the issue of encodings is outside Complexity Theory, that Complexity Theory begins only after a problem is suitably encoded. E.g., Megiddo [22] points out that the the known polynomial time algorithms for the linear programming problem (originally shown by Khacian) assumes a certain encoding for the problem. Yet it may be possible to bring this encoding into our theory if we resort to meta-mathematical tools.

Bibliography

[1] A. Aho, J. Hopcroft, and J. Ullman. *Data structures and algorithms*. Addison-Wesley, 1983.

[2] Manuel Blum. A machine-independent theory of the complexity of recursive functions. *Journal of Algorithms*, 14(2):322–336, 1967.

[3] Giles Brassard. Crusade for a better notation. *SIGACT News*, 17:1:60–64, 1985.

[4] Alonzo Church. An unsolvable problem of elementary number theory. *Amer. J. Math. 58*, pages 345–363, 1936.

[5] Alan Cobham. The intrinsic computational difficulty of functions. *Proc. 1964 International Congress for Logic, Methodology and Philosophy of Science*, pages 24–30, 1964.

[6] The Computer Society of the IEEE. *Proceedings, Third Annual Conference on Structure in Complexity Theory*. Computer Society Press of the IEEE, June 14-17, 1988. (held at Georgetown University, Washington, D.C.).

[7] T. H. Corman, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press and McGraw-Hill Book Company, Cambridge, Massachusetts and New York, 1990.

[8] Jack Edmonds. Paths, trees, and flowers. *Canadian J. Math.*, 17:449–467, 1967.

[9] Jack Edmonds. Matroid partition. In G.B. Dantzig and Jr. A.F. Veinott, editors, *Mathematics of the decision sciences*. Amer. Math. Soc., Providence, R.I., 1968.

[10] Kurt Gödel. Uber formal unentscheidbare Satze der Principia Mathematica und verwandter System I. *Monatschefte Math. Phys.*, 38:173–98, 1931. (English Translation in [Dav65]).

[11] L. M. Goldschlager. A universal interconnection pattern for parallel computers. *Journal of Algorithms*, 29:1073–1087, 1982.

[12] Yuri Gurevich. What does $O(n)$ mean? *SIGACT News*, 17:4:61–63, 1986.

[13] G. H. Hardy. Properties of logarithmico-exponential functions. *Proc. London Math. Soc.*, 2:54–90, 1912.

[14] G. H. Hardy and E. M. Wright. *An Introduction to the Theory of Numbers*. Oxford University Press, London, 1938.

[15] J. Hartmanis and R. E. Stearns. On the computational complexity of algorithms. *Trans. Amer. Math. Soc.*, 117:285–306, 1965.

[16] Douglas R. Hofstadter. *Godel, Escher, Bach: an eternal golden braid*. Vantage, New York, N.Y., 1980.

[17] Jia-wei Hong. *Computation: Computability, Similarity and Duality*. Research notices in theoretical Computer Science. Pitman Publishing Ltd., London, 1986. (available from John Wiley & Sons, New York).

[18] David S. Johnson. The *NP*-completeness column: an ongoing guide. the many faces of polynomial time. *Journal of Algorithms*, 8:285–303, 1987.

[19] S. Jukna. Succinct data representation and the complexity of computations. In L. Lovász & E. Szemerédi, editor, *Theory of algorithms*, volume 44, pages 271–282. Elsevier Science Pub. Co., 1985.

[20] E. Kaltofen. Greatest common divisors of polynomials given by straight-line programs. *Journal of Algorithms*, 35:231–264, 1988.

[21] Donald E. Knuth. Big omicron and big omega and big theta. *SIGACT News*, Vol.8, No.2:18–24, 1976.

[22] N. Megiddo. Is binary encoding appropriate for the problem-language relationship? *Theoretical Computer Science*, 19:337–341, 1982.

[23] Marvin Minsky. *Computation: Finite and Infinite Machines.* Prentice-Hall, Inc., Englewood Cliffs, N.J., 1967.

[24] Ian Parberry. Parallel speedup of sequential machines: a defense of the parallel computation thesis. *SIGACT News*, 18:1:54–67, 1986.

[25] Vaughn R. Pratt and Larry Stockmeyer. A characterization of the power of vector machines. *Journal of Computers and Systems Sciences*, 12:198–221, 1976.

[26] W. V. Quine. Concatenation as a basis for arithmetic. *Journal of Symbolic Logic*, 11 or 17?:105–114, 1946 or 1952?

[27] Michael Rabin. Degree of difficulty of computing a function. Technical Report Tech. Report 2, Hebrew Univ., 1960.

[28] Hartley Jr. Rogers. *Theory of Recursive Functions and Effective Computability.* McGraw-Hill, 1967.

[29] A. Schönhage. Storage modification machines. *SIAM Journal Computing*, 9(3):490–508, 1980.

[30] A. Schönhage and V. Strassen. Schnelle Multiplikation Grosser Zahlen. *Computing*, Vol. 7:281–292, 1971.

[31] Richard Schroeppel. A two counter machine cannot calculate 2^n. Technical Report AI Memo 257, M.I.T., 1973.

[32] Alan M. Turing. On computable number, with an application to the Entscheidungs problem. *Proc. London Math. Soc., Ser.2-42*, pages 230–265, 1936.

[33] R. Verbeek and K. Weihrauch. Data representation and computational complexity. *Theoretical Computer Science*, 7:99–116, 1978.

[34] P. M. B. Vitányi and L. Meertens. Big omega versus the wild functions. *SIGACT News*, 16(4):56–59, 1985.

[35] Frances F. Yao. Computation by 2-counter machines. Unpublished term paper, M.I.T., 1973.

[36] Chee-Keng Yap. Space-time tradeoffs and first order problems in a model of programs. *12th ACM Symposium on Theory of Computing*, pages 318–325, 1980.

[37] Chee-Keng Yap. *Three studies on computational problems.* PhD thesis, Yale University, 1980. PhD Thesis, Computer Science Department.

Appendix A

Basic Vocabulary

This appendix establishes some general terminology and notation used throughout the book.

We assume the usual set theoretic notations. We write $S \subseteq S'$ for set inclusion, and $S \subset S'$ for proper set inclusion. We let $|S|$ denote the cardinality of a set S.

Instead of the equality sign, we use ':=' to indicate a definitional equality. For example, we write '$S := \ldots$' if '$\ldots$' serves as a definition of the set S.

Let D, R be sets. By a *partial function* f with *domain* D and *range* R we mean a rule that associates certain elements x of D with an element $f(x) \in R$. So D and R are assumed to be specified when f is given. The function is said to be *undefined* at x if the rule does not associate x to any element of R, and we denote this by $f(x)\uparrow$; otherwise, we say f is *defined* at x and write $f(x)\downarrow$. If f is defined at all elements of its domain, we say f is *total*. Composition of functions is denoted $f \circ g$ where $(f \circ g)(x) := f(g(x))$.

The set of *integers* and the set of *real numbers* are denoted $\mathbb{Z}$ and $\mathbb{R}$, respectively. The set of *extended reals* refers to $\mathbb{R} \cup \{\infty\}$. The set $\mathbb{N}$ of *natural numbers* refers to the non-negative integers, $\mathbb{N} = \{0, 1, 2, \ldots\}$. The *floor function* takes any real number x to the largest integer $\lfloor x \rfloor$ no larger than x. The *ceiling function* takes any real number x to the smallest integer $\lceil x \rceil$ no smaller than x. So $\lfloor 0.5 \rfloor = 0 = \lceil -0.9 \rceil$.

An *alphabet* Σ is a non-empty finite set of markings. We call each element of Σ a *letter* or *symbol*. The set of finite sequences of letters (called *words* or *strings*) over an alphabet Σ is denoted Σ^*. The *empty word* is denoted ϵ. A *language* is a pair (Σ, L) where Σ is an alphabet and L is a subset of Σ^*. Usually, the alphabet Σ of a language (Σ, L) is either understood or immaterial, and we shall loosely refer to L as the language. The language L is *trivial* if $L = \Sigma^*$ or $L = \emptyset$ where $\emptyset$ denotes the empty set. The *complement* of L, denoted *co-L*, is the language $(\Sigma, \Sigma^* - L)$. Note that the complementation operation is an instance of a language operator whose proper definition requires that the alphabet of the language be explicitly given. A collection K of languages is usually called a *class*. For any class K, *co-K* is defined to be the class $\{\textit{co-}L : L \in K\}$.

The concatenation of two words v and w is written $v \cdot w$ or simply vw. This notation extends naturally to sets of words: if S and T are sets of words then $S \cdot T := \{vw : v \in S, w \in T\}$. The length of a word w is denoted $|w|$. The unique word in Σ^* of length zero is denoted ϵ, and Σ^+ is defined as $\Sigma^* - \{\epsilon\}$. (We may assume that the word ϵ is common to all Σ^*.) For any non-negative integer n and word w, we let w^n denote the n-fold self-concatenation of w. More precisely, $w^0 := \epsilon$ and for $n \geq 1$, $w^n := w \cdot w^{n-1}$.

A *language operator* ω is a partial d-ary function, $d \geq 0$, taking a d-tuple $(L_1, \ldots, L_d)$ of languages to a language $\omega(L_1, \ldots, L_d)$. Here we assume that the L_i have a common alphabet which is also the alphabet of $\omega(L_1, \ldots, L_d)$. Some simple language operators are now introduced. Other important operators, to be introduced in the course of this book, are usually defined using machines; this is in contrast with the following set-theoretic definitions.

Let $(\Sigma, L), (\Sigma, L')$ be languages. The *complement* of L, denoted *co-L*, is the language $\Sigma^* - L$. The *union*, *intersection* and *difference* of (Σ, L) and (Σ', L') are (resp.) the languages $(\Sigma \cup \Sigma', L \cup L')$, $(\Sigma \cup \Sigma', L \cap L')$ and $(\Sigma, L - L')$. (The preceding are the *Boolean operators*.) The *concatenation* of L and L', denoted $L \cdot L'$, is the language $\{ww' : w \in L, w' \in L'\}$ over the alphabet $\Sigma \cup \Sigma'$. For any non-negative integer n, we define the language L^n inductively as follows: L^0 consists of just the empty

word. $L^{n+1} := L^n \cdot L$. The *Kleene-star* of L, denoted L^*, is the language $\bigcup_{n\geq 0} L^n$. (Note that the Σ^* and L^* notations are compatible.) A related notation is L^+ defined to be $L \cdot L^*$. The *reverse* of L, denoted L^R, is $\{w^R : w \in L\}$ where w^R denotes the reverse of w.

A language L is said to be *finite* (resp. *co-finite*) if $|L|$ (resp. $|co\text{-}L|$) is finite.

Let Σ and Γ be alphabets. A *substitution* (from Σ to Γ) is a function h that assigns to each $x \in \Sigma$ a subset of Γ^*. h is naturally extended to a function (still denoted by) h that assigns to each word in Σ^* a set of words in Γ^*: $h(a_1a_2\cdots a_n) := h(a_1)h(a_2)\cdots h(a_n)$. We say h is *non-erasing* if $\epsilon \notin h(x)$ for all $x \in \Sigma$. A *homomorphism* is a substitution h where each set $h(x)$ has exactly one element (we may thus regard $h(x)$ as an element of Γ^*). A *letter homomorphism* is a homomorphism where $h(x)$ is a word of length 1 for all $x \in \Sigma$ (we may thus regard $h(x)$ as an element of Γ). An *isomorphism* is a letter homomorphism such that h is a bijection from Σ to Γ . An isomorphism is therefore only a 'renaming' of the alphabet.

For every substitution h from Σ to Γ, we may define the language operator (again denoted by h) that takes a language (Σ, L) to the language $(\Gamma, h(L))$ where $h(L)$ is the union of the sets $h(w)$ over all $w \in L$. We also define the *inverse substitution* operator h^{-1} that takes a language (Γ, L') to $(\Sigma, h^{-1}(L'))$ where $h^{-1}(L')$ is the set $\{w \in \Sigma^* : h(w) \subseteq L'\}$.

A *(language) class* K is a collection of languages that is closed under isomorphism. We emphasized in Chapter 1 that complexity theory is primarily the study of language classes, not of individual languages. The classes which interest us are usually defined using machines that use a limited amount of computing resources. In this case, we call K a *complexity class* although this is only an informal distinction.

Operators are important tools for analyzing the structure of complexity classes. For instance, many important questions are of the form "Are two complexity classes K and K' equal?". If Ω is any set of operators, let

$$\Omega(K) := \{\omega(L_1, \ldots, L_d) : L_i \in K, \omega \text{ is a } d\text{-ary operator in } \Omega\}.$$

The *closure* of K under Ω is

$$\Omega^*(K) := \bigcup_{n\geq 0} \Omega^n(K)$$

where $\Omega^0(K) := K$ and $\Omega^{n+1}(K) := \Omega(\Omega^n(K))$. One possible approach to showing that K is not equal to K' is to show that, for a suitable class of operators Ω, K is closed under Ω (i.e., $\Omega^*(K) = K$) but K' is not. An important simple case of Ω is where Ω consists of just the Boolean complement operator; here $\Omega(K) = \{co\text{-}L : L \in K\}$ is simply written as $co\text{-}K$. A branch of formal language theory called AFL theory investigates closure questions of this sort and certain complexity theory questions can be resolved with this approach.

Contents

Chapter 2

The Turing Model: Basic Results

February 5, 1999

2.1 Introduction

We take the Turing model of computation as the *canonical* one in Complexity Theory. In this we are simply following the usual practice but other more logical reasons can be given: the fundamental analysis by which Turing arrives at his model is still one of the most cogent arguments in support of Church's thesis.[1] The simplicity of Turing's basic model is appealing. Despite the fact that Turing predated our computer age, there is a striking resemblance between his machines and modern notions of computation. Henceforth, any new model that we introduce shall (perhaps only implicitly) be compared with this canonical choice. Of course, Turing considered only the fundamental mode of computation but we can naturally adapt it to the other computational modes. Furthermore, we find it convenient to consider variants of the original Turing machine. Recall from chapter 1 that all these model-specific details turn out to be unimportant for the major conclusions of our theory. It is somewhat paradoxical that some form of these model-dependent details cannot be avoided in order to attain our model-independent conclusions.[2]

In this chapter we will study some basic complexity results in the Turing model. For the time being, we restrict ourselves to the fundamental and the nondeterministic modes of computation. A Turing machine basically consists of a finite-state 'black-box' operating on one or more tapes where each tape is divided into an infinite linear

[1] According to Martin Davis [8] Gödel did not believe in Church's thesis until he heard of Turing's results. Gödel's skepticism arose from his insistence on an *á priori* analysis of the concept of computation (which is what Turing provided). The article contains an authoritative account of the origins of Church's thesis.

[2] One is reminded of Sir Arthur Eddington's elephants in *Nature of the Physical World*.

sequence of squares. Each square is capable of storing a single symbol (chosen from a finite set depending on the particular machine). Each tape has a reading head scanning a square. Under the control of the black-box, in one step the heads can change the contents of the square being scanned and can move left or right to the adjacent squares. The following computational resources have been studied with respect to Turing machines:

time, space, reversals, ink, tapes, symbols, states.

Time and space are intuitively clear, so we briefly explain the other resources. 'Reversal' is the number of times some tape head changes directions. 'Ink' measures the number of times the machine has to write on a tape square (a tape square may be rewritten many times). 'Tapes' ('symbols, ' 'states, ' respectively) are the number of tapes (symbols, states, respectively) used in the definition of the machine. The latter three resources are called *static resources* because they are a function of the machine description only. The others are *dynamic resources* since the amount of (say) time or space used also depends on the particular computation. In Complexity Theory, we primarily study dynamic resources. Time, space and reversals are considered basic and will be studied in this chapter. Reversals (in contrast to time and space) may initially appear artificial as a measure of complexity. It is a comparatively recent realization that reversal is an essential ingredient for gluing together space and time.

Reading Guide. This is a long chapter containing basic technical results about the Turing model. It is important that the student understand the subtle points in the definitions of complexity in section 3. Sections 6, 7 and 8 form a group of results about simulations that use time, space and reversals (respectively) as efficiently as possible: these techniques are fundamental and should be mastered. Sections 10 and 11 may be safely omitted since later chapters do not rely on them.

Two conventions for languages and machines

The reader should be familiar with the vocabulary of formal language theory; the appendix in chapter one is given for convenient reference.

We will be introducing mathematical "machines" in this chapter, and as above, these machines have *states* and they operate on *symbols*. These machines will be used to define languages. In order to simplify their description, we make the following universal convention for the entire book:

> **Convention** (α)**.** We fix Σ_∞ to be any infinite set of markings that are called *symbols*. Σ_∞ is the *universal set of symbols*, assumed to contain every symbol (such as $0, 1, a, b, \$, \#$, etc) that we will ever use in defining machines or languages. It contains a distinguished symbol $\Box$ called the *blank symbol*. No alphabet of a language contains this blank symbol.

Convention (β)**.** We fix Q_∞ to be any infinite set of markings that are called *states*. We assume Q_∞ is disjoint from Σ_∞. It is called the *universal set of states*, assumed to contain every state that we will ever use in defining machines. It contains three distinguished states q_0, q_a and q_r called the **start state**, the **accept state**, and the **reject state**, respectively. The explicit use of reject states, however, can be avoided until chapter 7.

2.2 Turing Machines

The **multitape Turing machine** is now introduced. Turing machines are used in two capacities: to define languages and to define functions over words.[3] In the former capacity, we call them **acceptors** and in the latter, **transducers**. For now, we focus on acceptors. Transducers will be used in chapter 4 and in §9.

It is convenient to regard a Turing acceptor as having two parts: a transition table δ (defining the set of machine instructions) together with an 'acceptance rule' (specifying when a computation of δ accepts its input). The advantage of this approach is that, just by specifying alternative acceptance rules, we obtain different choice modes. This separation is also a recognition of the distinction between the Turing model of computation and computational modes (which are basically model independent).

Informally, a multitape Turing acceptor may be viewed as a physical machine or system consisting of a finite state automaton equipped with $k+1$ (paper) tapes for some $k \geq 0$: an *input tape* (tape 0) and k *work tapes* (tapes $1, \ldots, k$). Each tape consists of a doubly infinite sequence of *cells* (or *tape squares*) indexed by the integers. Each tape square contains a symbol in Σ_∞ . On tape i ($i = 0, \ldots, k$), there is a *head* (head i) that *scans* some tape cell. The *input head* refers to head 0 and the rest are called *work heads*. To begin a computation, we are given an input word $x = a_1 a_2 \cdots a_n$ on the input tape; initially every cell of the each tape contains the blank symbol except for cells $1, 2, \ldots, n$ of the input tape containing the input string x. The head on each tape initially scans cell 1. Depending on the current state and the $k+1$ symbols scanned under the heads, the machine has certain instructions that are *executable*. Executing one of these instructions results in a new state, in new symbols under each head, and in the movement of each head at most one cell to the right or left of the current cell. The machine halts when it has no executable instruction. By definition, an input head is restricted so that it may not change the contents of its tape. In general, a tape with such a restriction is termed *read-only*. The machine may or may not eventually halt. See Figure 2.1

The machine just described is also known as an *off-line k-tape Turing machine.*

[3] A third use, as 'generators' of languages, is not needed in our book. Generators corresponds to grammars in formal language theory. Also, languages are just unary relations. We could have extended Turing machines to accept any k-ary relation.

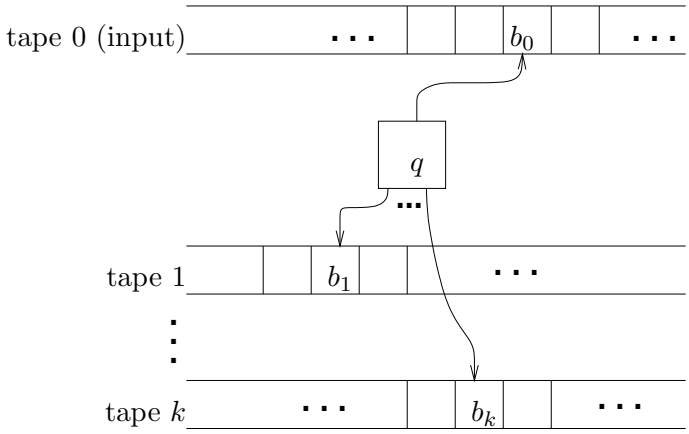


Figure 2.1: An off-line k-tape Turing machine.

If we restrict the input head to move only from left to right, we obtain an *on-line* version. In general, a tape is called *one-way* if its head is constrained to move in only one direction (and for emphasis, we call ordinary tapes *two-way*). (In the literature, off-line and on-line machines are also called 2-way or 1-way machines, respectively.) Sometimes we have a k-tape ($k \geq 1$) Turing machine *without* an input tape; then the input is conventionally placed on one of the work-tapes. The special case of this where $k = 1$ is called a *simple Turing machine* (this is essentially the original Turing model and is often useful because of its extreme simplicity). In this book, we will mainly use the off-line multitape version (and simply call them 'Turing machines' without qualifications), and we occasionally use the simple Turing machine. The main reason for the use of an input tape is so that we can discuss sublinear space complexity (otherwise, the space is at least linear just to represent the input).

Remark: Many other variants of Turing machines have been studied: we may allow more than one head on each tape, consider tapes that are multi-dimensional (the above tapes being one-dimensional), allow tapes that have a binary tree structure, tape heads that can remember a fixed number of previously visited positions and can 'reset' to these positions in one step, etc. For reasons discussed in the previous chapter, these variations are not inherently interesting unless they give rise to interesting new complexity classes.

Recall our conventions (α) and (β) in the Section 1 concerning the universal sets Σ_∞ and Q_∞ of symbols and states. We now use these to formalize the physical machines described above.

Definition 1 *Let $k \geq 0$. A k-*tape transition table *δ is a finite subset of*

$$\Sigma_\infty^{k+1} \times Q_\infty \times \Sigma_\infty^k \times Q_\infty \times \{+1, 0, -1\}^{k+1}$$

■

Each $(3k+4)$-tuple in δ has the form

$$\langle b_0, b_1, \ldots, b_k, q_1, c_1, \ldots, c_k, q_2, d_0, \ldots, d_k \rangle \tag{2.1}$$

and represents a machine instruction which is interpreted as follows:

> "if b_i (for each $i = 0, \ldots, k$) is the symbol being scanned on tape i and q_1 is the current state, then change b_j (for each $j = 1, \ldots, k$) to c_j (leaving b_0 unchanged), make q_2 the next state, and the ith tape head (for $i = 0, \ldots, k$) should move left, stay in place or move right according as $d_i = -1, 0$ or $+1$"

To simplify proofs, it is convenient to make the following restrictions on transition tables:

(1) The input head never scans more than one blank cell past the input word in either direction.

(2) There are no state transitions *from* the accept or reject states q_a, q_r or *into* the initial state q_0. (Recall that q_a, q_r, q_0 are distinguished states in Q_∞.)

(3) The blank symbol $\square$ cannot be written on any of the work tapes (thus, after a blank cell is visited it turns non-blank). Formally, this means that in (2.1), $c_j \neq \square$ for each j.

A transition table satisfying the above restrictions is said to be in *standard form*. It is easy to make syntactic restrictions on transition tables to ensure that they are in standard form. From now on, unless otherwise stated, this assumption is made. The restrictions on δ to obtain on-line or simple Turing machines is also a simple exercise.

The *tape alphabet* of δ is the set of symbols (excluding the blank symbol) that occur in some tuple of δ. The *input alphabet* of δ is the set of symbols (excluding the blank) that occur as the first component of a tuple in δ (i.e., as b_0 in (2.1)). Thus the input alphabet is a subset of the tape alphabet. The *state set* of δ is similarly defined.

A *configuration* of δ is a $(2k+3)$-tuple

$$C = \langle q, w_0, n_0, w_1, n_1, \ldots, w_k, n_k \rangle = \langle q, w_i, n_i \rangle_{i=0}^{k} \tag{2.2}$$

where q is in the state set of δ, each $w_j (j = 1, \ldots, k)$ is a word over the tape alphabet of δ, w_0 is a word over the input alphabet of δ, and $0 \leq n_i \leq 1 + |w_i| (i = 0, \ldots, k)$. The string w_i represents the non-blank portion of tape i. The convention that blanks cannot be written ensures that the non-blank portion of each tape is contiguous. The integer n_i indicates that head i is scanning the n_ith symbol in w_i. For a string x, let $x[i]$ denote the ith symbol in x if $1 \leq i \leq |x|$. If i is outside the

indicated range, then by definition, $x[i]$ denotes the blank symbol. Thus if w_i is not the empty string ϵ then $n_i = 1$ (respectively, 0) means the first (respectively, the blank prior to the first) symbol of w_i is scanned. The *initial configuration* on input x is defined to be

$$C_0(x) = \langle q_0, x, 1, \epsilon, 1, \ldots, \epsilon, 1\rangle$$

where q_0 is the start state.

Remarks: The integers n_i are *relative* addresses of cells, namely, relative to the leftmost non-blank cell. Occasionally we use instead the absolute addressing of cells, but there ought to be no confusion. In the literature, configurations are also called *instantaneous descriptions (ID's).*

We define the binary relation $\vdash_\delta$ (or $\vdash$, if δ is understood) on configurations of δ. Informally, $C \vdash_\delta C'$ means the configuration C' is obtained by modifying C according to some instruction of δ. More precisely, suppose C is given by (2.2) and δ contains the instruction given in (2.1). We say the instruction (2.1) is *applicable* to C if $q = q_1$ and $w_i[n_i] = b_i$ for each $i = 0, \ldots, k$. Then *applying* (2.1) to C we get

$$C' = \langle q', w'_0, n'_0, \ldots, w'_k, n'_k\rangle$$

where $q' = q_2$, $n'_i = d_i + max\{1, n_i\}$ $(i = 0, \ldots, k)$, [4] $w'_0 = w_0$ and for $j = 1, \ldots, k$:

$$w'_j = \begin{cases} c_j w_j & \text{if} \quad n_j = 0 \\ w_j c_j & \text{if} \quad n_j = |w_j| + 1 \\ u_j c_j v_j & \text{if} \quad w_j = u_j b_j v_j, \text{and} |u_j b_j| = n_j \end{cases}$$

We write $C \vdash_\delta C'$ in this case; C' is called a *successor* of C and the sequence $C \vdash_\delta C'$ is called a *transition* or a *step.* We write $C \vdash_\delta^k C'$ if there is a k-step sequence $C_0 \vdash C_1 \vdash \cdots \vdash C_k$ such that $C = C_0$ and $C' = C_k$. The reflexive, transitive closure of the binary relation $\vdash_\delta$ is denoted $\vdash_\delta^*$ (or just $\vdash^*$).

Observe that the instruction (if any) of δ applicable to C is generally not unique. We say δ is *deterministic* if each configuration C of δ has at most one instruction applicable to it. Otherwise δ *has choice.* We say the Turing machine with a deterministic δ operates in the *deterministic mode*, otherwise it operates in the *choice mode.*

A configuration is *accepting* if its state is the accept state q_a. Similarly for *rejecting* if its state is the accept state q_r. It is *terminal* if it has no successor. For transition tables in standard form, accepting or rejecting configurations are terminal. A *computation path* (of δ) is either a finite sequence of configurations

$$\overline{C} = (C_0, C_1, \ldots, C_m) = (C_i)_{i=0}^m$$

or an infinite one

$$\overline{C} = (C_0, C_1, C_2, \ldots) = (C_i)_{i \geq 0}$$

[4] This unintuitive formula takes care of the case $n_i = 0$ in which case $n'_i = 1$ even though $d_i = 0$.

such that C_0 is an initial configuration, C_m (when the path is finite) is terminal, and for each $i \geq 0$, $C_i \vdash C_{i+1}$. We may denote $\overline{C}$ by

$$\overline{C} = C_0 \vdash C_1 \vdash C_2 \vdash \cdots \vdash C_m$$

or

$$\overline{C} = C_0 \vdash C_1 \vdash C_2 \vdash \cdots.$$

Any contiguous subsequence

$$C_i \vdash C_{i+1} \vdash C_{i+2} \vdash \cdots \vdash C_j$$

$(0 \leq i \leq j \leq m)$ is called a *sub-computation path.*

We call $\overline{C}$ an *accepting computation path on input* x if, in addition, $C_0 = C_0(x)$ is an initial configuration on x, and the path terminates in an accepting configuration C_m. *Non-accepting* computation paths come in two flavors: they either do not terminate or terminate in non-accepting configurations. Although it is often unnecessary to distinguish between these two situations, this distinction is crucial sometimes (see §9; also chapters 7 and 8).

The next definition captures an important form of the choice mode:

Definition 2 *A* nondeterministic Turing acceptor *M is given by a transition table* $\delta = \delta(\mathrm{M})$*, together with the following acceptance rule.*

> **Nondeterministic Acceptance Rule.** *A word* x *is* accepted *by M iff* x *is over the input alphabet of M and there is an accepting computation path of* δ *for* x*.*

■

M is a *deterministic Turing acceptor* if δ is deterministic. By convention, a Turing acceptor is assumed to be nondeterministic unless otherwise specified. The *language accepted* by M is given by (Σ, L) where Σ is the input alphabet of $\delta(\mathrm{M})$ and L consists of those words that are accepted by M. We write $L(\mathrm{M})$ for the language accepted by M.

Example 1 We describe informally a 1-tape deterministic Turing acceptor which accepts the palindrome language $L_{pal} = \{w \in \{0,1\}^* : w = w^R\}$ where w^R is the reversal of the word w. The reader should feel comfortable in translating this into a formal description (i.e. in terms of δ) because, from now on, we shall describe Turing machines in such informal terms. The acceptor works in three stages:

(i) Copy the input w onto tape 1.

(ii) Move the input head back to the start of the input, but leave the head on tape 1 at the right end.

(iii) Move the input head right and head 1 left, in synchrony, comparing the symbols under each head – they should agree or else we reject at once. Accept iff all the symbols agree.

■

Example 2 (Guessing, verifying and nondeterminism) We give a nondeterministic 1-tape acceptor for the complement *co-*L_{pal} of the palindrome language. Note that $x \in \{0,1\}^*$ is in *co-*L_{pal} iff for some i:

$$1 \leq i \leq n = |x| \text{ and } x[i] \neq x[n-i+1]. \tag{2.3}$$

Using nondeterminism we can 'guess' such an i and 'verify' that it has the properties in (2.3). By 'guessing' an i we mean that the machine initially enters a state q_1 such that in this state it has two choices:

(i) Write down a '1' in its single work-tape, move head 1 one position to the right and remain in state q_1;

(ii) Write a '1' in the work-tape, keeping head 1 stationary and enter a new state q_2.

During the guessing stage (state q_1), the input head does not move. When we enter state q_2, there is a unary word on work-tape. This is the unary representation of the guessed i. Let x be the input of length n. To 'verify' that i has the right properties, we can determine $x[i]$ by moving the input head to the right while moving head 1 to the left, in synchrony. When head 1 reaches the first blank past the beginning of i, the input head would be scanning the symbol $x[i]$. We can 'remember' the symbol $x[i]$ in the finite state control. Notice that the guessed i may be greater than n, and this could be detected at this point. If $i > n$ then we reject at once (by this we mean that we enter some non-accepting state from which there are no transitions). Assuming $i \leq n$, we can similarly determine $x[n-i+1]$ by moving the input head to the end of x and moving i steps to the left, using tape 1 as counter. We accept if $x[i] \neq x[n-i+1]$, rejecting otherwise.

To check that the above machine accepts the complement of L_{pal}, observe that if x is in L_{pal} then every computation path will be non-accepting; and if x is not in L_{pal} then the path corresponding to the correct guess of i will lead to acceptance. This example shows how nondeterminism allows us to check if property (2.3) for *some* values of i, by testing for all values of i simultaneously. This ability is called "guessing", and generally, it simplifies the logic of our machines. ■

Transformations and Turing Transducers

This subsection is placed here for easy reference – the concepts of transformations and transducers is only used in section 9.

Definition 3 *A* multivalued transformation *is a total function*

$$t : \Sigma^* \to 2^{\Gamma^*}$$

where Σ, Γ *are alphabets and* 2^S *denotes the power set of* S*. If* $t(x)$ *is a singleton set for all* $x \in \Sigma^*$ *then we call* t *a* transformation *and think of* t *as a function from* Σ^* *to* Γ^**.* ■

Definition 4 *A* nondeterministic k-tape transducer T *is a nondeterministic* $(k+1)$*-tape* ($k \geq 0$) *Turing acceptor such that tape 1 is constrained to be one-way (i.e., the head may not move left). Tape 1 is called the* output tape *and the non-blank word on this tape at the end of any accepting computation path is called the* output word *of that path. The work tapes of* T *now refer to tapes 2 to* $k+1$*.* T *is said to compute a multivalued function* t *in the natural way: for any input* x*,* $t(x)$ *is the set of all output words* y *in accepting computation paths.* ■

The set $t(x)$ can be empty. In case T computes a transformation, then T has the following properties: (a) For each input x, there exists at least one accepting path. (b) On a given x, all accepting paths lead to the same output word. We call such a transducer *univalent*. The usual example of a univalent transducer is a deterministic transducer that accepts all its inputs.

Complexity of transducers. In the next section we will define computational complexity of acceptors for various resources (time, space, reversal). The computational complexity of a transducer T is identical to the complexity when T is regarded as an acceptor, with the provision that space on tape 1 is no longer counted.

2.3 Complexity

In this section, we see the first concepts of computational complexity. We consider the three main computational resources of time, space and reversal.

The *time* of a computation path $\overline{C} = (C_0, C_1, \cdots)$ is one less than the length of the sequence $\overline{C}$; the time is infinite if the length of the sequence is infinite. The *space used by* a configuration $C = \langle q, w_i, n_i \rangle_{i=0}^{k}$ is defined as

$$space(C) = \sum_{i=1}^{k} |w_i|$$

Observe that the space in the input tape is not counted. The *space used by* the computation path $\overline{C} = (C_i)_{i \geq 0}$ is $\sup\{space(C_i) : i \geq 0\}$; again the space could be infinite. It is possible for the time to be infinite while the space remains finite.

The definition of reversal is a little subtle. Let $\overline{C} = (C_i)_{i \geq 0}$ be a computation path of a k-head machine. We say that head h ($h = 0, \ldots, k$) *tends in direction* d (for $d \in \{-1, +1\}$) in C_i if the last transition $C_{j-1} \vdash C_j$ (for some $j \leq i$) preceding C_i

in which head h moves is in the direction d. This means that the head h "paused" in the time from $j+1$ to i. If head h has been stationary from the start of the computation until C_i we say head h tends in the direction $d=0$ in C_i. We also say that head h has *tendency* d if it tends in direction d. It is important to note that the head tendencies of a configuration C are relative to the computation path in which C is embedded. We say head h makes a *reversal* in the transition $C_{j-1} \vdash C_j$ in $\overline{C}$ if the tendency of head h in C_{j-1} is *opposite* to its tendency in C_j. We say $C_{j-1} \vdash C_j$ is a *reversal transition*. The *reversal* in a reversal transition $C_{j-1} \vdash C_j$ is the number of heads (including the input head) that makes a reversal. So this number is between 1 and $1+k$. The *reversal* of $\overline{C}$ is the total number of reversals summed over all reversal transitions in the entire computation path.

Observe that changing the tendency from 0 to ± 1 is not regarded as a reversal. Each computation path $\overline{C}$ can be uniquely divided into a sequence of disjoint subcomputation paths $P_0, P_1, \ldots,$ where every configuration in P_i $(i = 0, 1, , \ldots,)$ has the same head tendencies. Each P_i is called a *phase*. The transition from one phase P_i to the next P_{i+1} is caused by the reversal of at least one head. Clearly the reversal of $\overline{C}$ bounded by $k+1$ times the the number of phases in $\overline{C}$.

Remarks: An alternative to our definition of reversal complexity is to discount reversals caused by the input head. Hong [20] (see section 8.3) uses this alternative. This would be consistent with our decision not to count the space used on the input tape. However, the real concern there was to admit sublinear space usage to distinguish problems with low space complexity. Our decision here allows possible complexity distinctions which might be otherwise be lost (see remark at the end of §7).

We give two ways to define the usage of resources.

Definition 5 *(Acceptance complexity) If x is an input for* M*, define $AcceptTime_{\mathrm{M}}(x)$ to be the least time of an accepting computation path for x. If* M *does not accept x, $AcceptTime_{\mathrm{M}}(x) = \infty$. If $n \in \mathrm{N}$, define*

$$AcceptTime_{\mathrm{M}}(n) := \max_x \{AcceptTime_{\mathrm{M}}(x)\}$$

where x ranges over words in $L(\mathrm{M})$ of length n; if $L(\mathrm{M})$ has no words of length n then $AcceptTime_{\mathrm{M}}(n) = \infty$. Let f be a complexity function. M accepts in time f *if $f(n)$ dominates $AcceptTime_{\mathrm{M}}(n)$.* ■

Note that $AcceptTime_{\mathrm{M}}(x)$ is a complexity function; it is a partial function since it is defined iff x is a natural number. Although this definition is stated for the time resource, it extends directly to the space or reversal resources. Indeed, all the definitions we give in this section for the time resource naturally extend to space and reversal.

One consequence of our definition is that any finite or co-finite language is accepted by some Turing acceptor in time (respectively, space, reversal) $f(n) = O(1)$

(respectively, $f(n) = 0, f(n) = 0$). Another technical consequence is that a k-tape acceptor uses at least k tape cells. Thus to achieve space 0, we must use 0-tape acceptors.

The fact that M accepts x in time r does not preclude some (whether accepting or not) computation on input x from taking more than r time. Furthermore, if x is not in $L(\mathrm{M})$, it is immaterial how much time is used in any computation on x! This stands in contrast to the next definition:

Definition 6 *(Running complexity) For any input x for* M*, let $RunTime_{\mathrm{M}}(x)$ be the maximum over the time of* all *computation paths of* M *on input x. For $n \in \mathrm{N}$, let*

$$RunTime_{\mathrm{M}}(n) := \max_x RunTime_{\mathrm{M}}(x)$$

where x ranges over all words (whether accepted by M *or not) of length n. For any complexity function f, we say* M runs in time f *if $f(n)$ dominates $RunTime_{\mathrm{M}}(n)$.* ■

Note that if M runs in time f then $f(n) \downarrow$ for all $n \in \mathrm{N}$. If $f(n) < \infty$ then M must halt in every computation path on inputs of length n. If M is deterministic and $x \in L(\mathrm{M})$ then $AcceptTime_{\mathrm{M}}(x) = RunTime_{\mathrm{M}}(x)$.

Example 3 Refer to examples 1 and 2 in the last section. The space and time of the deterministic acceptor for palindromes are each linear; the reversal is 3. Similarly, acceptance space and time of the nondeterministic acceptor for the complement of palindromes are each linear, with reversal 3. However the running space and time of the nondeterministic acceptor is infinite for all n since the guessing phase can be arbitrarily long. We leave as exercises for the reader to modify the machine to satisfy the following respective complexity bounds: (a) the running space and time are each linear; (b) the time is linear and space is logarithmic. In (a), you can actually use a 1-tape machine and make only 1 reversal. What is the reversal in (b)? ■

The following notion is sometimes useful: we call f *time-constructible* function if there is a deterministic M such that for *all* inputs x of sufficiently large length, $RunTime_{\mathrm{M}}(x) = f(|x|)$.
M is said to *time-construct* f.

The definitions of *space-constructible* or *reversal-constructible* are similar: systematically replace the word 'time' by 'space' or 'reversal' above. We have defined constructible functions with respect to deterministic machines only, and we use running complexity. Such functions have technical applications later. The reader may

[4]In the literature, a function that $f(n)$ of the form $RunTime_M(n)$ is sometimes said to be 'time-constructible'. What we call time-constructible is also described as 'fully time-constructible'. Other related terms include: measurable (Hopcroft-Ullman), real-time computable (Yamada) or self-computable (Book), honest (Meyer-McCreight).

try this: describe a 1-tape Turing machine that time-constructs the function n^2. It may be easier to first "approximately" time-construct n^2. See the Exercises for more such constructions.

The Turing acceptor for the palindrome language in the example of the last section is seen to run in linear time and linear space. But see section 9 for a more space-efficient machine: in particular, logarithmic space is sufficient (but at an inevitable blow-up in time requirement).

We shall use acceptance complexity as our basic definition of complexity. The older literature appears to prefer running complexity. Although running complexity has some desirable properties, in proofs, it sometimes takes additional effort to ensure that every computation path terminates within the desired complexity bound. Consequently, proofs for accepting complexity are often (but not always) slightly shorter than for running complexity. A more compelling reason for acceptance complexity is that it is more basic: we could define 'rejection complexity' and view running complexity as a combination of acceptance and rejection complexity. The problem of termination can be overcome by assuming some 'niceness' conditions on the complexity function. The exact-time complexities and time-constructible complexities are typical notions of niceness. Indeed, for nice functions, complexity classes defined using running complexity and acceptance complexity are identical. Since most of the important complexity classes (such as P, NP, etc, defined next) are bounded by nice functions, the use of running or acceptance complexity lead to the same classes in these cases. It is important to realize that most common functions are nice (see Exercises). The general use of running complexity is largely avoided until chapter 8 when we discuss stochastic computations where it seems that running complexity is the more fruitful concept.

Resource bounded complexity classes. We introduce some uniform notations for complexity classes defined by bounds on computational resources used by acceptors.[5] Let F be a family of complexity functions. The class $NTIME(F)$ is defined to consist of those languages accepted by nondeterministic Turing acceptors in time f, for some f in F. If $F = \{f\}$, we simply write $NTIME(f)$. The notation extends to languages accepted by deterministic Turing acceptors ($DTIME(F)$), and to space and reversal complexity ($XSPACE(F)$ and $XREVERSAL(F)$ where $X = N$ or D indicates nondeterministic or deterministic classes). When *running complexity* (time, space, reversal, etc) is used instead of acceptance complexity, a subscript 'r' is appended to the usual notations for complexity classes. For instance, $NTIME_r(F)$, $NSPACE_r(F)$, $DREVERSAL_r(F)$, etc. It is clear that $DTIME_r(F) \subseteq DTIME(F)$ and $NTIME_r(F) \subseteq NTIME(F)$, and similarly for the other resources.

[5] We appear to be following Ronald Book in many of these notations. He attributes some of these suggestions to Patrick Fischer.

Canonical Classes. Most common families of complexity functions can be obtained by iterating the following operations $lin, poly, expo$ on an initial family F: $lin(F) = O(F)$, $poly(F) = F^{O(1)}$, $expo(F) = O(1)^F$. For instance, the following families will be used often:

$$\{\log n\}, \log^{O(1)} n, \{n+1\}, O(n), n^{O(1)}, O(1)^n, O(1)^{n^{O(1)}}.$$

Each family in this sequence 'dominates' the preceding family in a natural sense. For future reference, we collect in the following table the special notation for the most important complexity classes:

Canonical Classes

Special Symbol	Standard Notation	Name
$DLOG$	$DSPACE(\log n)$	deterministic log-space
$NLOG$	$NSPACE(\log n)$	nondeterministic log-space
$PLOG$	$DSPACE(\log^{O(1)} n)$	polynomial log-space
P	$DTIME(n^{O(1)})$	deterministic poly-time
NP	$NTIME(n^{O(1)})$	nondeterministic poly-time
$PSPACE$	$DSPACE(n^{O(1)})$	polynomial space
$DEXPT$	$DTIME(O(1)^n)$	deterministic simply-exponential time
$NEXPT$	$NTIME(O(1)^n)$	nondeterministic simply-exponential time
$DEXPTIME$	$DTIME(2^{n^{O(1)}})$	deterministic exponential time
$NEXPTIME$	$NTIME(2^{n^{O(1)}})$	nondeterministic exponential time
$EXPS$	$DSPACE(O(1)^n)$	simply-exponential space
$EXPSPACE$	$DSPACE(2^{n^{O(1)}})$	exponential space

These classes are among the most important in this theory and for convenience, we shall refer to the list here as the *canonical list.*[6] Note that our distinction between "exponential" and "simply-exponential" is not standard terminology. It will follow from results later in this chapter that (after omitting $PLOG$, $DEXPT$ and $NEXPT$) each class in the above list is included in the next one on the list.

In Chapter 1 we said that complexity theory regains a Church-like invariance property provided that we parametrize the complexity classes with (1) the computational resource and (2) the computational mode. Our notation for complexity classes reflects this analysis: each class has the form

$$\mu - \rho(F)$$

where $\mu = D, N$, etc., is the computational mode, and $\rho = TIME, SPACE$, etc., is the computational resource. According to the computational theses in Chapter 1,

[6]Note that P should really be 'DP' but the 'P' notation is so well-accepted that it would be confusing to change it. $DLOG$ is also known as L or LOG in the literature. Similarly, $NLOG$ is also known by NL.

these classes are model-independent in case $F = n^{O(1)}$. In particular, the canonical classes P, NP and $PSPACE$ have this invariance. That is, if we had defined these notions using some other general computational model such as pointer machines instead of Turing machines, we would have ended up with the same classes $(P, NP, PSPACE)$.

Simultaneous resource bounds. In the preceding definitions of complexity, we bound one resource but place no restrictions on the other resources (actually the resources are not completely independent of each other). In 'simultaneous complexity' we impose bounds on two or more resources within the same computation path.

Definition 7 *(Acceptance within simultaneous bounds) Let $t, s \geq 0$ be real numbers and f, g be complexity functions. An acceptor* M *accepts an input x* in time-space bound of (t, s) *if there exists an accepting computation path for x whose time and space are at most t and s, respectively.* M *accepts in time-space (f, g) if there is some n_0 such that for all $x \in L(\mathrm{M})$, if $|x| \geq n_0$ then* M *accepts x in time-space $(f(|x|), g(|x|))$.* ■

For any complexity functions t and s, let $X\text{-}TIME\text{-}SPACE(t, s)$ (where $X = D$ or N) denote the class of languages L that are accepted by some (deterministic or nondeterministic, depending on X) Turing machine M that accepts in time-space (t, s). The complexity classes defined in this way are called *simultaneous space-time classes*. Clearly,

$$X\text{-}TIME\text{-}SPACE(t, s) \subseteq XTIME(t) \cap XSPACE(s)$$

although it seems unlikely that this would be an equality in general. Similarly we will consider simultaneous space-reversal, time-reversal, time-space-reversal classes denoted, respectively,

$$X\text{-}SPACE\text{-}REVERSAL(s, r), X\text{-}TIME\text{-}REVERSAL(t, r)$$

and

$$X\text{-}TIME\text{-}SPACE\text{-}REVERSAL(t, s, r)$$

where t, s and r are time, space and reversal complexity bounds. For a more compact notation, we could use (but do not recommend) the alternative *XTISP, XTIRE, XSPRE* and *XTISPRE* for *X-TIME-SPACE, X-TIME-REVERSAL*, etc.

Finally, we note these notations will be extended later (see chapter 7) when 'X' in these notations may be replaced by symbols for other computational modes.

Recursively enumerable languages. We occasionally refer to the class of languages accepted by Turing acceptors without any complexity bounds. Even in this case, we can distinguish an important subclass of languages accepted by Turing acceptors which does not have any infinite computation paths.

Definition 8 *A language is* recursively enumerable *or, r.e., if it is accepted by some deterministic Turing acceptor. A* **halting Turing machine** *is one that does not have any infinite computation path on any input. A language is* recursive *if it is accepted by some halting Turing acceptor. Let RE and REC denote the classes of r.e. and recursive languages, respectively.* ■

In recursive function theory, the fundamental objects of study are partial number-theoretic functions instead of languages. The recursive function theory analogues of the above definitions are: a partial function from the natural numbers to natural numbers is *partial recursive* if it is computed by some Turing transducer (assume that natural numbers are encoded in binary) where the function is undefined at the input values for which the transducer does not halt. If the function is total, then it is called a *(total) recursive function.*

2.4 Linear Reduction of Complexity

The results to be shown are of the form: if a language can be accepted in (time, space, or reversal) resource f then it can be accepted in resource cf, for any $c > 0$. The idea is that, by using a transition table with more states and a larger alphabet, we can trade-off dynamic complexity for static complexity. These technical results are very useful: they justify the use of the 'big-oh' notation for complexity functions. For instance, we can talk of a language being accepted in 'quadratic time' or 'logarithmic space' without ambiguity. It also illustrates our discussion in chapter 1 where we concluded that complexity functions must not be taken in an absolute sense, but only up to O-order.

Our first theorem, taken from Stearns, Hartmanis and Lewis [13], says that the space complexity of a problem can be reduced by any constant factor and it is sufficient to use a 1-tape acceptor.

Theorem 1 (Space Compression) *Let $c > 0$ be a constant and $s = s(\cdot)$ a complexity function. For any multitape* M *that accepts in space s, there is a 1-tape* N *that accepts L(M) in space cs. If* M *is deterministic, so is* N.

Proof. As will be the rule in such proofs, we only informally describe N since the formal description of N is tedious although straightforward. It is sufficient to show this for $c = 1/2$ since we can make c arbitrarily small with repeated applications of the construction. The single work tape of N contains k 'tracks', one track for each work tape of M. See Figure 2.2.

Each cell of N can be regarded as containing a 'composite symbol' that is essentially a k by 4 matrix with the ith row containing 4 tape symbols (possibly blanks) of the ith work-tape of M, and possibly a marker ($\uparrow$) for the head position. There are five forms for each row:

$$[b_1b_2b_3b_4], [\uparrow b_1b_2b_3b_4][b_1 \uparrow b_2b_3b_4][b_1b_2 \uparrow b_3b_4][b_1b_2b_3 \uparrow b_4]$$

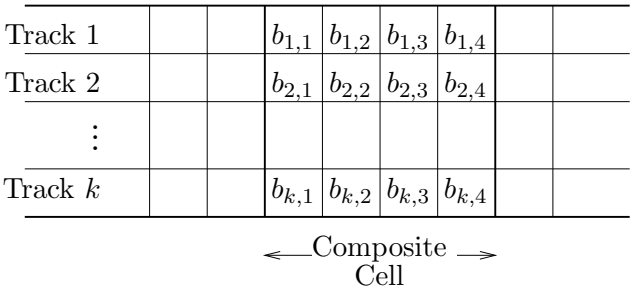


Figure 2.2: Composite cell with k tracks.

where each b_i is either a blank symbol or a tape symbol of δ(M). The marker $\uparrow$ indicates that the immediately succeeding b_i is being scanned. Thus a symbol of N encodes $4k$ symbols of M. Since all the scanned cells have absolute indexing from $-s(n)+1$ to $s(n)-1$ on an input of length n, we see that $s(n)/2 + O(1)$ composite symbols of N suffice to encode the contents of M's tapes. But in fact $\lfloor s(n)/2 \rfloor$ cells suffice if we exploiting the finite state machinery of N to store a constant amount of extra information. N simulates a step of M by making a 'right-sweep' (i.e., starting from the leftmost non-blank cell to the rightmost non-blank) of the non-blank portion of N's tapes, followed by a return 'left-sweep'. The 'current neighborhood' of M consists of those cells of M that either are scanned by some tape head, or are immediately adjacent to such a scanned cell. On the right-sweep, N takes note of (by marking in some appropriate manner) the current neighborhood of M. On the left-sweep, it updates the current neighborhood to reflect the changes M would have made. It is clear that N accepts in space $s(n)/2$. Note that N is deterministic if M is deterministic. **Q.E.D.**

The proof illustrates the reduction of dynamic complexity at the expense of static complexity. In this case, we see space being reduced at the cost of increasing the number of tape symbols and the number of states. The amount of information that can be encoded into the static complexity of an acceptor is clearly finite; we refer to this technique as *storing information in the finite state control* of the acceptor.

Corollary 2 *Let $X = D$ or N. For any complexity function s,*

$$XSPACE(s) = XSPACE(O(s)).$$

The reader may verify that theorem 1 and its corollary hold if we use 'running space' rather than 'acceptance space' complexity. The next result from Hartmanis and Stearns [14] is the time analog of the previous theorem.

Theorem 3 (Linear Speedup) *Given $c > 0$ and a k-tape acceptor* M *that accepts in time $t(n) > n$, there is a $(k+1)$-tape* N *that accepts in time $n + ct(n)$ with L(M) = L(N). If* M *is deterministic so is* N.

Proof. Choose $d > 1$ to be an integer to be specified. Tape j of N, for $j = 1, \ldots, k$, will encode the contents of tape j of M using composite symbols. Similar to the last proof, each composite symbol encodes d tape symbols (including blanks) of M; but unlike that proof, the present composite symbols do not need to encode any head positions of M and do not need multiple tracks. Tape $k+1$ in N will be used to re-code the input string using composite symbols. We describe the operation of N in two phases. Let the input string be x with $|x| = n$.

Set-up Phase. First N copies the input x from tape 0 to tape $k+1$. This takes $n+1$ steps. So now tape $k+1$ contains x in a 'compressed form'. Next N moves head $k+1$ leftward to the first non-blank composite symbol using $\frac{n}{d} + O(1)$ steps. Henceforth, N ignores tape 0 and treats tape $k+1$ as the input tape.

Simulation Phase. N now simulates M by making repetitive sequences of 5 moves, where each sequence simulates d moves of M. These 5 moves have the same form on each tape so it is enough to focus on some generic tape $t = 1, \ldots, k$. Define the 'current neighborhood' of N on tape t to be the currently scanned cell or the two cells immediately adjacent to the scanned cell. First, N makes three moves to determine the contents of its current neighborhood on each tape: if head t of N is scanning the composite cell j, the first 3 moves will visit in turn the composite cells $j+1$, j and $j-1$. Based on these 3 moves, N now "knows" what will be in the current neighborhood after the next d steps of M. Notice that in d steps, each head of M modifies at most two composite cells in each tape, and remains within the current neighborhood. Hence N can make 2 more moves to update the current neighborhood and to leave its head on the appropriate composite cell, in preparation for the next sequence of 5 moves. To illustrate these 2 moves, suppose that the tape head ends up at composite cell $j+1$ after d moves of M. In this case, N can move from composite $j-1$ (where its head is positioned after the first 3 moves) to j and then to $j+1$, updating cells j and $j+1$. The other cases are similarly treated.

If M accepts or reject within d steps, then N accepts or rejects instantly. This completes the simulation phase.

Timing Analysis. We show that N uses $\leq n + ct(n)$ moves in the above simulation of M. In the set-up phase, $n + \frac{n}{d} + O(1)$ moves were made. In the simulation phase, we see that each sequence makes at most 5 steps correspond to d steps of M (the sequence just before termination has only 3 steps). Thus N made $\leq \frac{5t(n)}{d} + O(1)$ moves in this phase. Summing the time for the two phases we get $n + \frac{n}{d} + \frac{5t(n)}{d} + O(1) < n + \frac{6t(n)}{d} + O(1)$. Thus, if we choose $d > 6/c$, the total time will be $\leq n + ct(n)$(ev.). **Q.E.D.**

The following is immediate:

Corollary 4

(i) If $t(n) = \omega(n)$ then $XTIME(t)$=$XTIME(O(t))$, where $X = D$ or N.

(ii) For all $\epsilon > 0, DTIME((1+\epsilon)n) = DTIME(O(n))$.

We do not state the nondeterministic version of Corollary 4 (ii) because a stronger result will be shown next. We show that with $k+3$ (rather than $k+1$) work-tapes we can exploit nondeterminism to strengthen theorem 3. In fact the three additional tapes of N are known as 'checking stacks, ' i.e., the contents of these tapes, once written, are not changed. The basic idea is that the set-up and simulation phases in the proof of theorem 3 can be carried out simultaneously. Nondeterminism is used in an essential way: the reader who is not familiar with nondeterministic computations will find the proof highly instructive. The proof is adapted from Book and Greibach [3]; the same ideas will be exploited later in this chapter when we consider the problem of reducing the number of work-tapes with no time loss.

Theorem 5 (Nondeterministic linear speedup) *For all* $c > 0$*, and for any* k*-tape* M *that accepts in time* $t(n) \geq n$*, there is a nondeterministic* $(k+3)$*-tape* N *that accepts in time* $\max\{n+1, ct(n)\}$ *with* $L(\mathrm{M}) = L(\mathrm{N})$.

Proof. Let d be some integer to be specified later. As in the proof of theorem 3, N 'compresses' the tape contents of M, i.e., uses composite symbols that encode d tape symbols of M at a time. The tapes of N are used as follows. Tape i $(i = 1, \ldots, k)$ of N represents in compressed form the contents of tape i of M. Tape $k+1$ contains in compressed form an initial segment of the (actual) input string. Tapes $k+2$ and $k+3$ each contains a copy of a 'guessed' input string, again in compressed form. In addition, the simulation of M by N uses tape $k+3$ as its 'input' tape. The idea is for N to proceed with the 8-move simulation of theorem 3 using a 'guessed' input string. In the meantime, N verifies that the guess is correct.

We describe the operations of N in two phases; each phase consists of running two simultaneous processes (the reader should verify that this is possible because the simultaneous processes operate on different tapes).

Initial Phase. The following two processes run in parallel.

> *Process 1.* This process copies some initial prefix x_1 of the input x into tape $k+1$, in compressed form. This takes $|x_1|$ steps. The prefix x_1 is nondeterministically chosen (at each step, the transition table of N has two choices: to stop the process instantly or to continue, etc).
> *Process 2.* On tapes $k+2$ and $k+3$, N writes down two copies, *in compressed form*, of a guessed string y in the input alphabet of M. When the writing of y is completed (nondeterministically of course), heads $k+2$ and $k+3$ move synchronously back (ie., left-ward) over the string y. Process 2 halts nondeterministically at some point *before* these heads move past the first blank symbol left of y.

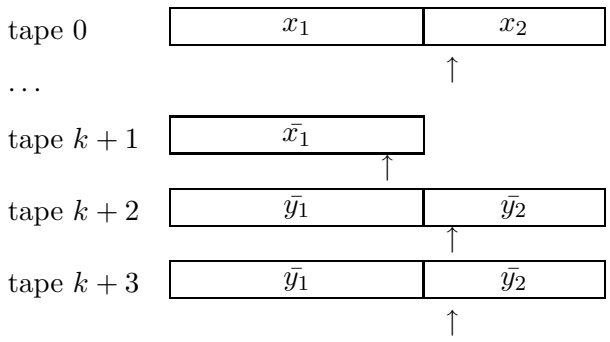


Figure 2.3: At the end of the initial phase

The initial phase ends when both processes halt (if one process halts before the other then it marks time, doing nothing). At this point, on tape 0, we have $x = x_1x_2$ where x_1 is the initial segment that process 1 copied onto tape $k+1$ and head 0 is scanning the first symbol of x_2. Similarly, $y = y_1y_2$ where heads $k+2$ and $k+3$ (on their respective tapes) are scanning the composite symbol that contains the first symbol of y_2. Note that any of x_i or $y_i (i = 1, 2)$ may be empty. The figure 2.3 illustrates the situation (not to scale!) where $\bar{w}$ denotes the compressed version of a string w and $\uparrow$ indicates a head position.

Final Phase. We run the following two processes in parallel.

> *Process 3.* The input head continues to scan the rest of the input (i.e., x_2), comparing it with the compressed copy of y_2 on tape $k+2$: for this purpose, note that heads 0 and $k+2$ are conveniently positioned at the left-most symbol of x_2 and y_2, respectively, at the start of the final phase. If $x_2 \neq y_2$, then N rejects at once. Process 3 halts when the checking confirms that $x_2 = y_2$.
> *Process 4.* First we verify that $x_1 = y_1$, using x_1 on tape $k+1$ and y_1 on tape $k+3$. If $x_1 \neq y_1$, N rejects at once. Otherwise, head $k+3$ will now be positioned at the beginning of y and we can begin the "8-move simulation" of M (as in the proof of Theorem 2). Process 4 halts when this simulation is complete.

We remark that in processes 3 and 4 above, when we say "N rejects at once", we mean that the described computation path halts without acceptance. It does not mean that N is rejecting the input in a global sense. This explanation holds

generally for other similar situations.

The final phase ends when both processes halt. N accepts if neither process rejects: this means $x = y$ and N's simulation of M results in acceptance. To see the time taken in this phase, clearly process 3 takes $\leq |x_2|$ steps. Checking $x_1 = y_1$ by process 4 takes $\leq \frac{|x_1|}{d} + O(1)$ steps since x_1 and y_1 are in compressed form. Since the 8-move simulation takes $8t(|x|)/d$ steps, process 4 takes $\frac{|x_1|}{d} + \frac{8t(|x|)}{d} + O(1)$ steps. Thus the final phase takes $1 + \max\{|x_2|, \frac{|x_1|}{d} + \frac{8t(|x|)}{d} + O(1)\}$ steps. (Note that the "1+" is necessary in making the final decision to accept or not.)

We now prove that $L(\mathrm{M}) = L(\mathrm{N})$. If x is not in $L(\mathrm{M})$ then it is easy to see that every computation path of N is non-accepting x because one of the following will fail to hold:

(a) $y_2 = x_2$,

(b) $y_1 = x_1$,

(c) M accepts x.

Conversely, if x is in $L(\mathrm{M})$ then there is a computation path in which (a)-(c) hold and the final phase accepts.

It remains to bound the acceptance time of N. Consider the computation path in which, in addition to (a)-(c), satisfies the following:

(d) $|x_1| = \lceil 2n/d \rceil$ where $n = |x|$.

Process 2 uses $\lceil \frac{n}{d} \rceil$ steps to guess y and $\lceil \frac{|y_2|}{d} \rceil$ steps to position the heads $k+2$ and $k+3$. When we specify d below, we will ensure that $d \geq 3$. Hence for n large enough, the initial phase takes $\leq \max\{|x_1|, \frac{n+|y_2|}{d} + O(1)\} = |x_1|$ steps. Combining this with the time taken by the final phase analyzed earlier, the total time is

$$\begin{aligned} & |x_1| + 1 + \max\{|x_2|, \frac{|x_1|}{d} + \frac{8t(|x|)}{d} + O(1)\} \\ \leq \;& 1 + \max\{n, \frac{(d+1)|x_1|}{d} + \frac{8t(n)}{d} + O(1)\} \\ \leq \;& 1 + \max\{n, \frac{11t(n)}{d}\} (\text{since } t(n) \geq n). \end{aligned}$$

The last expression is $\leq \max\{n+1, ct(n)\}$ if we choose $d > 11/c$. Hence d can be any integer greater than $\max\{3, 11/c\}$. **Q.E.D.**

An interesting consequence of the last result is:

Corollary 6 $NTIME(n+1) = NTIME(O(n))$.

A Turing acceptor that accepts in time $n+1$ is said to be *real-time.*[7] Thus the real-time nondeterministic languages coincide with those *linear time* nondeterministic languages. This stands in contrast to the fact that $DTIME(n+1)$ is a proper subset of $DTIME(O(n))$.

[7] In the literature $NTIME(n+1)$ is also denoted by Q standing for 'quasi-realtime'.

Before concluding this section, we must make a brief remark about 'linear reduction of reversals'. In some sense, reversal is a more powerful resource than either space or time, since for every $k > 0$ there are non-regular languages that can be accepted using k reversals but that cannot be accepted using $k-1$ reversals [16]. In contrast, only regular languages can be accepted using a constant amount of space or time. Indeed, in section 6, the power of reversals is even more dramatically shown when we prove that all *RE* languages can be accepted with just two reversals when we allow nondeterminism.

For simple Turing machines, reversal complexity seems to behave more to our expectation (based on our experience with time and space): Hartmanis[15] has shown that only regular languages can be accepted with $O(1)$ reversals. Furthermore, Fischer [11] shows there is a linear speedup of reversals in this model. However, it is not clear that a linear speedup for reversals in the multi-tape machine model is possible in general. It is impossible for small complexity functions because of the previous remark about languages accepted in k but not in $k-1$ reversals.

2.5 Tape Reduction

Theorem 1 shows that with respect to space, a Turing acceptor may as well use one work-tape. This section has three similar results on reducing the number of work-tapes. The first is a simple result which reduces k work-tapes to one work-tape, at the cost of increasing time quadratically. The second is a classic simulation of a k-tape machine by a 2-tape machine due to Hennie and Stearns [17], in which the simulation is slower by a logarithmic factor. Although these results hold for nondeterministic as well as deterministic machines, our third result shows that we can do much better with nondeterministic simulation. This is the result of Book, Greibach and Wegbreit [4] showing that a nondeterministic 2-tape machine can simulate a k-tape machine without increasing the time, space or reversals by more than a constant factor. Finally, a tape-reduction result for deterministic reversals is indicated at the end of section 8.

Such tape reduction theorems have important applications in later chapters. Essentially they imply the existence of 'efficient universal' Turing machines (chapter 4), which in turn are used to prove the existence of complete languages (chapter 5) and in obtaining hierarchy theorems (chapter 6).

We now state the first result.

Theorem 7 *If L is accepted by a multitape Turing machine* M *within time t then it is accepted by a 1-tape Turing machine* N *within time $O_{\mathrm{M}}(t^2)$.* N *is deterministic if* M *is deterministic. Moreover, the space (resp., reversal) used by* N *is bounded by a constant times the space (resp., $O(t)$) used by* M.

Proof. Assume M has k work-tapes. We can use the 1-tape machine N described in the proof of the space compression theorem, where each symbol of N is a viewed

as $k \times d$ matrix of M's tape symbols. For the present proof, d may be taken to be 1. Note that the size of the non-blank portion of N's work-tape is $\leq i$ after the ith step. Hence the ith step can be simulated in $O_{\mathrm{M}}(i)$ steps of N. So to simulate the first t steps of M requires $\sum_{i=1}^{t} O_{\mathrm{M}}(i) = O_{\mathrm{M}}(t^2)$ steps of N. The claim about space and reversal usage of N is immediate. **Q.E.D.**

We know that the above simulation is essentially the best possible in the deterministic case: Maass [27] shows that there are languages which require time $\Omega(n^2)$ on a 1-tape machine but can be accepted in real time by a 2-tape machine. See also [25].

Theorem 8 (Hennie-Stearns) *If L is accepted by a k-tape Turing machine within time $t(n)$ then it is accepted by a 2-tape Turing machine within time $O(t(n) \log t(n))$.*

Proof. We use an ingenious encoding of the work-tapes of a k-tape machine M using only tape 1 of a 2-tape machine N. The other work-tape of N is used as a scratch-tape. Tape 1 of N has $2k$ tracks, two tracks for each tape of M. Each cell of N is a composite symbol viewed as a column containing $2k$ symbols (possibly blanks) of M. The cells of N are grouped into 'blocks' labeled by the integers:

$$\cdots, B_{-2}, B_{-1}, B_0, B_1, B_2, B_3, \cdots$$

where B_0 consists of just a single cell (call it cell 0) of N. We will number the individual cells by the integers also, with the initial head position at cell 0. For $j > 0$, B_j (respectively, B_{-j}) consists of cells of N in the range

$$[2^{j-1}, 2^j) = \{2^{j-1}, 2^{j-1} + 1, \ldots, 2^j - 1\}$$

(respectively, $(-2^j, -2^{j-1}]$). Thus blocks B_j and B_{-j} each has 2^{j-1} cells, $j > 0$. We assume the boundaries of these blocks are marked in easily detectable ways as cells are visited, the details being left to the reader. The key idea is that, instead of having the k simulated heads of M at different parts of N's tracks, we constrain them to always be at cell 0. The contents in M's tapes are translated laterally to allow this. Essentially this means that when head i of M moves leftward (say) the entire contents of tape i must be shifted rightward so that the currently scanned symbol of tape i is still in cell 0 of N's tape 1. This may appear to require expensive movement of data, but by a suitable scheme of 'delayed' data movement, we show that there is only a small time penalty.

It is enough for us to explain what occurs in two of the tracks that represent some work-tape T of M. These two tracks are designated *upper* and *lower*, and let B_j^U and B_j^L denote the restriction of B_j to the upper and lower tracks, respectively. Inductively, there is an integer $i_0 \geq 0$ that is non-decreasing with time such that a block B_i is 'active' iff $|i| \leq i_0$. We initialize $i_0 = 0$. The non-blank portion of tape T is contained in the active blocks. We hold the following property to be true of each active block B_i:

(1) Each B_i^X ($X = U$ or L) is either full (*i.e.*, represents 2^{i-1} symbols of M, including blanks) or empty (*i.e.*, does not represent symbols of M, not even blanks). The contents of a full block represent contiguous symbols of T. Note that B_i^X may be filled entirely with blanks of M and still is regarded as full.

(2) Exactly two of B_i^U, B_i^L, B_{-i}^U and B_{-i}^L are full. In particular, both B_0^U and B_0^L are always full (it is easy to initialize this condition at the beginning of the computation). In addition, if B_i^L is empty, so is B_i^U. (We allow B_i^U to be empty while B_i^L is full.)

(3) If $-i_0 \leq i < j \leq i_0$ then B_i represents contents of T that are to the left (in their natural ordering on tape T) of those represented in B_j. Furthermore, if both B_i^L and B_i^U are full then B_i^L represents contents of T that are to the left of those in B_i^U.

Let us now see how to simulate one step of M on T. Suppose that the head of T moves outside the range of the two symbols stored in B_0. Say, the next symbol to be scanned lies to the right and $i \leq i_0$ is the smallest positive integer such that B_i^L is full (recall that if B_i^L is empty then B_i^U is also empty). [If no such i exists, we may increment i_0, fill both tracks of B_{i_0} with blanks of M, and make both tracks of B_{-i_0} empty. Now we may choose i to be i_0.] Clearly, the upper track of B_{-i}^U is empty, and both tracks of B_j for $j = -i+1, -i+2, \ldots, -1, 0$ are full. We copy the contents of these B_j's (there are exactly 2^i symbols) onto the scratch-tape in the correct sequential order (*i.e.*, as they would appear in tape T). There are two cases:

(i) B_{-i}^L is empty. Then transcribe the 2^i symbols in the scratch-tape to the lower tracks of $B_{-i}, B_{-i+1}, \ldots, B_{-1}, B_0$.

(ii) B_{-i}^L is full. Then we transcribe the contents of the scratch tape to the upper track of B_{-i} and to the lower tracks of $B_{-i+1}, \ldots, B_{-1}, B_0$.

Observe that there is exactly enough space in cases (i) and (ii). Now copy the contents of B_i^L to tape 2 and transcribe them onto the lower tracks of $B_1, \ldots, B_{i-1}$ and the upper track of B_0: again there is exactly enough space. (If B_i^U is full we then move it to the lower track.) We call the preceding computations an *order* $|i|$ *operation*. Note that we have now returned to our inductive hypothesis and the scanned symbol of T is in B_0 as desired. Clearly a order $|i|$ operation takes $O(2^{|i|})$ time. Repeating this for each of the k tapes of M, we have completed the simulation of one step of M.

To analyze the cost of this simulation, note that an order $i \geq 1$ operation can only occur if the lower track lying strictly between B_0 and B_{-i} or between B_0 and B_i is empty. Also, immediately following such an operation, the cells in the lower track between B_{-i} and B_i are full. This implies that M must make at least 2^{i-1} moves between two consecutive order i operations. It is also clear that the first order i operation cannot occur until M has made at least 2^{i-1} moves. Suppose M

makes t moves. Then the largest value of i such that some order i operation is made is $m_0 = 1 + \lfloor \log_2 t \rfloor$. Therefore the number of order i operations is $\leq \frac{t}{2^{i-1}}$ and the total number of moves made by N is

$$\sum_{i=1}^{m_0} \frac{t \cdot O(2^i)}{2^{i-1}} = O(t \log t).$$

Q.E.D.

In section 8, we will prove that a deterministic k-tape machine can be simulated by a deterministic 2-tape machine with a quadratic blow-up in the number of reversals.

If we use a 2-tape nondeterministic machines to do the simulation, the result of Book, Greibach and Wegbreit below shows that the above quadratic blow-up in time and reversals can be avoided.

First we need some definitions. Let (Σ, L) be accepted by some nondeterministic k-tape machine M. Let Δ be the tape alphabet of M, Q the states of M, and suppose that the δ(M) contains $p \geq 1$ tuples which we number from 1 to p. Let

$$\Gamma = \Sigma \times Q \times \Delta^k \times \{0, 1, \ldots, p\}$$

be a new alphabet with composite symbols which may be regarded as $(k+3)$-tuples as indicated. Define a *trace* of a configuration C as a symbol of Γ of the form

$$b = \langle a, q, c_1, \ldots, c_k, \beta \rangle \tag{2.4}$$

where a is the currently scanned input symbol in C, q is the state in C, each c_j is currently scanned symbol in the work tape j, and β is the number of any tuple in δ(M) which is applicable to C. If there are no applicable tuples then $\beta = 0$. The trace of a computation path $\bar{C} = (C_0, \ldots, C_m)$ is the word $b_0 b_1 \cdots b_m$ where $b_i \in \Gamma$ is a trace of C_i, and for each $j < m$, the tuple number in b_j (*i.e.*, the last component of b_j) identifies the instruction of M causing the transition $C_j \vdash C_{j+1}$. The trace of $\bar{C}$ is unique. We are now ready for the proof.

Theorem 9 (Tape Reduction for Nondeterministic Machines) *Let $t(n) \geq n$ and* M *be a nondeterministic machine accepting in simultaneous time-reversal bound of (t, r). Then there is a 2-tape nondeterministic machine* N *accepting L*(M) *in simultaneous time-reversal bound of $O(t, r)$.*

Proof. Let M be as in the theorem. With Δ and Γ as above, we construct a 2-tape N that operates as follows: on input x, N guesses in tape 1 a word $w = b_0 \cdots b_m \in \Gamma^*$ that is intended to be the trace of an accepting computation of M on x. It remains to show how to verify if w is indeed such a trace. We may assume that w is generated in such a way that the transitions from b_i to b_{i+1} are plausible, e.g., if the instruction (corresponding to the tuple number) in b_i causes a transition to a new state q then

b_{i+1} has state q. Furthermore, the state in b_m is accepting. It remains to check that the symbol scanned under each head is the correct one.

For each $j = 1, \ldots, k$, we will check the symbols under head j is correct. To do this for a particular j, first N re-positions head 1 at the beginning of w. This takes $O(m)$ steps. Then N scans the word w from left to right, carrying out the actions specified each composite symbol b_i $(i = 0, \ldots, m)$ using its own input head to move as directed by the instructions in b_i, and using tape 2 to act as tape j of M. If N discovers that the symbol under its own input head does not agree with the symbol indicated in b_i, or the symbol scanned on its tape 2 does not agree with the symbol indicated in b_i, then N rejects at once. Otherwise, it proceeds to check the symbols for head $j + 1$. When all tapes are verified in this manner, then N accepts.

Note that the checking of tape contents is entirely deterministic. We now prove that this N accepts if and only in M accepts. If M accepts, then clearly N accepts. Suppose N accepts with a particular word $w = b_0 \cdots b_m \in \Gamma^*$. We can construct inductively for each $i \geq 0$, a unique sub-computation path $C_0, \ldots, C_i$ such that each b_i is a trace of C_i and C_0 is the initial configuration on input x. In particular, there is an accepting computation path of length m. Finally, we note that the time used by N is $O(m)$; moreover, if the accepting computation makes r reversals, then N makes $r + O(1)$ reversals. **Q.E.D.**

Note that we can apply the nondeterministic linear speedup result to N to reduce the time complexity from $O(t)$ to $max\{n+1, t(n)\} = t(n)$, but the number of work-tape would increase to 5. But if $t(n)$ is sufficiently fast growing, we can first apply speedup result to M then apply the above construction to achieve 2-tape machine with time complexity $t(n)$ rather than $O(t)$.

The cited paper of Maass also shows a language accepted by a real-time deterministic 2-tape machine that requires $\Omega(n^2/\log^5 n)$ time on any 1-tape nondeterministic machine. This shows that the Book-Greibach-Wegbreit result cannot be improved to 1-tape.

Remark: There are complexity differences between k and $k + 1$ tapes for all k when we restrict attention to real-time computations: more precisely, there are languages accepted in real-time by some $(k + 1)$-tape acceptor but not by any real-time k-tape acceptor. Rabin [32] proved this for $k = 1$ and Aanderaa [1] showed this in general. See also [31].

2.6 Simulation by Time

In this section, we will bound space and reversal resources in terms of time. The results are of the form

$$X\text{-}TIME\text{-}SPACE\text{-}REVERSAL(t, s, r) \subseteq Y\text{-}TIME(t')$$

where $X, Y \in \{D, N\}$. Generally, we show this by showing how to simulate a X-mode machine M using time-space-reversal (t, s, r) using a Y-mode machine N in

time t'. We then say we have a 'simulation by time'. We think of simulation by time as an attempt to minimize time without consideration of other resources. In the next two sections, we obtain similar results on simulations by space and by reversal. The importance of these three sections taken together is how they exemplify the vastly different computational properties of each of these resources.

Theorem 10 *Let t be a complexity function, $X = D$ or N. Then*

$$XTIME(t) \subseteq X\text{-}TIME\text{-}SPACE\text{-}REVERSAL(t, O(t), O(t)).$$

Proof. Observe that if a k-tape M accepts in time t then it clearly accepts in space kt and in reversal $(k+1)t$. **Q.E.D.**

In particular, $XTIME(f) \subseteq XSPACE(O(f)) = XSPACE(f)$, by the space compression theorem.

Theorem 11 *Let* M *be a deterministic or nondeterministic acceptor. For all complexity functions r, s, if* M *accepts in space-reversal (s, r) then*
a) each phase of M *has length $O_{\mathrm{M}}(n+s)$, and*
b) M *accepts in time $O_{\mathrm{M}}((n+s)\cdot r)$.*

Proof. A computation path $\overline{C}$ of M can be uniquely divided into phases which correspond to the time periods between reversals. Choose $\overline{C}$ so that no configuration is repeated in $\overline{C}$. There is a constant $O_{\mathrm{M}}(1)$ such that if all tape heads of M remain stationary for $O_{\mathrm{M}}(1)$ consecutive steps then some configuration would be repeated; by choice of $\overline{C}$, some tape head must move in any consecutive $O_{\mathrm{M}}(1)$ steps. There are at most n head motions that is attributable to the input head – the rest is clearly bounded by s. Hence if the simultaneous space-reversal of $\overline{C}$ is (s, r), then each phase has $O_{\mathrm{M}}(n+s)$ steps. This proves (a). Since there are r phases, the time of $\overline{C}$ is $O_{\mathrm{M}}(rs)$, proving (b). **Q.E.D.**

Corollary 12 *For $X = D, N$,*

$$X\text{-}SPACE\text{-}REVERSAL(s, r) \subseteq XTIME(O((n+s)\cdot r)).$$

Lemma 13 *Let* M *be fixed k-tape acceptor, x be an input and $h > 0$ any integer. Let*

$$CONFIGS_h(x) = \{\langle q, w_i, n_i\rangle_{i=0}^{k} : w_0 = x, \sum_{j=1}^{k} |w_j| \leq h\}$$

be the set of configurations of M *with input x using at most space h. Then*

$$|CONFIGS_h(x)| = n \cdot O_{\mathrm{M}}(1)^h$$

where $n = |x| + 1$.

Proof. There are at most $n+1$ positions for head 0, and at most $(h+2)$ positions for each of heads $1, \ldots, k$, in a configuration of $CONFIGS_h(x)$. There are at most $\binom{h+k-1}{k-1}$ ways to distribute h cells among k tapes. These h cells can be filled with tape symbols in at most d^h ways where d is the number of tape symbols of M, including with the blank symbol. Let q be the number of states in M. Thus:

$$|CONFIGS_h(x)| \leq q \cdot (n+2) \cdot (h+2)^k \cdot \binom{h+k-1}{k-1} \cdot d^h = n \cdot O(1)^h.$$

Q.E.D.

Theorem 14 *If* M *is a Turing machine that accepts in space s then* M *accepts in time $n \cdot O_{\mathrm{M}}(1)^{s(n)}$.* M *can be deterministic or nondeterministic.*

Proof. Let $\overline{C}$ be a shortest accepting computation path on input x using space $s(|x|)$. Then the configurations in $\overline{C}$ are all distinct and by the previous lemma, the length of $\overline{C}$ is upper bounded by $n \cdot O_{\mathrm{M}}(1)^{s(n)}$. **Q.E.D.**

Corollary 15 *With $X = D, N$, we have*

$$XSPACE(s) \subseteq X\text{-}TIME\text{-}SPACE(n \cdot O(1)^{s(n)}, s(n)).$$

Theorem 16 $NSPACE(s) \subseteq DTIME(n \cdot O(1)^{s(n)})$.

Proof. Let M be a nondeterministic k-tape machine accepting in space s. The theorem is proved by showing a deterministic N that accepts the same language $L(\mathrm{M})$ in time $n \cdot O(1)^{s(n)}$. It is instructive to first describe a straightforward simulation that achieves $O(n^2 \log n \cdot O(1)^{s(n)})$.

On input x, N computes in *stages*. In *stage* h (for $h = 1, 2, \ldots$), N has in tape 1 exactly h cells marked out; the marked cells will help us construct configurations using at most h space. Then N lists on tape 2 all configurations in $CONFIGS_h(x)$. Since $|CONFIGS_h(x)| = n \cdot O(1)^h$, and each configuration needs $h + \log n$ space (assuming the input head position is in binary and input word is not encoded with a configuration), we use a total of $n \log n \cdot O(1)^h$ space. On a separate track below each configuration in tape 2, we will mark each configuration as having one of three possible statuses: *unseen*, *seen* and *visited*. Initially, all configurations are marked 'unseen', except for the initial configuration which is marked 'seen'. Now N performs a sequence of $\leq |CONFIGS_h(x)|$ *sweeps* of the input data, where each sweep corresponds to *visiting* a particular 'seen' configuration; after the sweep, that 'seen' configuration will have a 'visited' status. To pick a configuration for sweeping, N picks out the leftmost 'seen' configuration C on tape 2. When visiting configuration C, N first generates all configurations derivable from C, say C_1 and C_2, and puts them on two different tapes. Then N goes through tape 2 again, this time to locate the occurrences of C_1 and C_2 in tape 2. This can be done in time $n \log n \cdot O(1)^h$.

If C_i is located in tape 2, we mark it 'seen' if it is currently 'unseen'; otherwise do nothing. Then we mark C itself as 'visited'. If we ever mark an accepting configuration as 'seen', we can accept at once; otherwise, when there are no more 'seen' configurations, we halt and reject. The total time in stage h is seen to be $n^2 \log n \cdot O(1)^h$. The correctness of this simulation follows from the fact that the input x is accepted by M iff N accepts at some stage h, $h \leq s(|x|)$. The total time to accept is

$$\sum_{h=1}^{s(n)} n^2 \log n \cdot O_1(1)^h = n^2 \log n \cdot O_2(1)^{s(n)}.$$

We now improve on the $O(n^2 \log n)$ factor. The idea [7] is to represent the input head positions implicitly. Let us call a *storage configuration* to be a 'standard' configuration except that information about the input tape (*i.e.*, the input word and the input head position) is omitted. Thus, for a configuration that uses h space, the corresponding storage configuration is just the current state, the contents of the work tapes and the positions of the work heads. This can be stored in $O(h)$ space. In stage h, we first compute on tape 2 a list L of all the storage configurations using space h. This list can be represented by a string of length $h \cdot O(1)^h = O(1)^h$. Then we duplicate L $n+2$ times. So tape 2 contains the string

$$T = \underbrace{L\#L\#\cdots\#L}_{n+2}.$$

This string represents every 'standard' configuration C as follows. If the input head of C is at position i ($i = 0, 1, \ldots, n+1$) and its storage configuration is S then C is represented by the $i+1$st occurrence of S in T. Again, we assume that the all the configurations in T are initially 'unseen' except the initial configuration is 'seen'. As long as T has any 'seen' configuration, we do repeated *sweeps* as before to 'visit' the leftmost 'seen' configuration. To visit C, we must mark as 'seen' any successor C' of C that is still 'unseen'. Note that there are $O(1)$ such successors and these are within $O(1)^h$ distance from C. Hence the marking process takes only $O(1)^h$ time.

To find the next 'seen' configuration to visit, we would also like to spend $O(1)^h$ steps in finding it. It turns out that this can only be done in the amortized sense. The idea is to keep track of the leftmost and rightmost 'seen' configuration: for each 'seen' configuration, we can store two flags, *leftmost* and *rightmost*, which will be set to true or false according as the configuration is leftmost and/or rightmost. Note that if the next leftmost 'seen' configuration lies to the left of the configuration being visited, then we can find it in $O(1)^h$ steps. This is a consequence of our policy to always visit the leftmost unseen configuration first. But if it lies to the right, there is basically no bound (*i.e.*, the maximum number $n \cdot O(1)^h$ of steps may be needed). However, we shall charge the cost to traverse an entire block L of configurations to the last visited configuration on that block. Then it is not hard to see that each visited configuration is charged at most once (assuming as we may, that each

configuration has at most two successors). Thus the charge of $O(1)^h$ per visited configuration is maintained. The total charges over all visited configurations is then $n \cdot O(1)^h$.

There is one other detail to mention: since the input head position in a configuration of T is implicit, we need to know the current input symbol for any configuration that we visit in T. This is easy to take care of by using the input head of the simulator to track the position of head 2. When head 2 is inside the ith copy of L in T, we assume that the simulator is scanning the ith cell in its input tape. This concludes our description. There are $n \cdot O(1)^h$ sweeps and each sweep takes $O(1)^h$ time, the total time of stage h is $n \cdot O(1)^h$. This implies the stated bound in our theorem. **Q.E.D.**

There is another basic method of doing a time simulation of nondeterministic space. First we review the connection between Boolean matrices and digraphs. A digraph G on n vertices can be represented by its adjacency matrix A where A is an $n \times n$ matrix with its (i,j)th entry $A_{i,j} = 1$ if and only if there is an edge from the vertex i to vertex j. We assume that $A_{i,i} = 1$ for all i. Recall that the product of two Boolean matrices is defined as in ordinary matrix multiplication except that addition becomes logical-or '$\vee$' and multiplication becomes logical-and '$\wedge$'. It is easy to see that if $B = A^2$ (A squared) then $B_{i,j} = 1$ if and only if there is a vertex k such that $A_{i,k} \wedge A_{k,j} = 1$, i.e. there is a path of length at most 2 from i to j. Arguing the same way, with $C = B^2 = A^4$, we see that $C_{i,j} = 1$ if and only if there is a path of length at most 4 from i to j. Therefore we see that A^* given by

$$A^* := A^{2^k}$$

(where k is the least integer such that $2^k \geq n$) then $A^*_{i,j} = 1$ if and only if there is a path from i to j. The matrix A^* is called the *transitive closure* of A. There is a well-known method attributed to Warshall and to Floyd for computing the transitive closure A^*. For $s = 0, 1, \ldots, n$, let $A^{(s)}$ be the matrix defined as follows: $A^{(s)}_{i,j} = 1$ if and only if there is a path from i to j such that the intermediate nodes (excluding i and j) lie in the set $\{1, \ldots, s\}$. Hence $A^* = A^{(n)}$. Clearly $A^{(0)} = A$. It is also easy to see that $A^{(s+1)}$ is obtained as follows:

$$A^{(s+1)}_{i,j} = A^{(s)}_{i,j} \vee \left(A^{(s)}_{i,s+1} \wedge A^{(s)}_{s+1,j} \right).$$

It is easy to see that this algorithm has complexity $O(n^3)$ on a random-access machine. It turns out (Exercises) that we can accomplish this on a Turing machine also, assuming that the matrix A is given in row-major order. We exploit this connection between paths and transitive closure. In particular, if we regard configurations as nodes of a graph, and edges correspond to the $\vdash$ relation, then deciding if a word is accepted amounts to checking for a path from the start configuration to any final configuration. We leave the details as an exercise. This alternative method is slightly less efficient with complexity $(nO(1)^{s(n)})^3 = n^3 O(1)^{s(n)}$. We will encounter the transitive closure method again.

A consequence of this theorem is that the class *NLOG* of languages accepted in nondeterministic logarithmic space is in *P*. Although theorem 16 remains valid if we replace *NSPACE* with *NTIME* the next theorem will show something slightly stronger. But first we note a fact similar to lemma 13.

Lemma 17 *Let* M *have k-tapes and x an input word. For any* $h \geq 1$*, let* $CONFIGS'_h(x)$ *denote the set of configurations of M that can be reached from the initial configuration on input x using at most h steps. Then*

$$|CONFIGS'_h(x)| = O_{\mathrm{M}}(1)^h.$$

Proof. As before, if q is the number of states in M and d is the number of tape symbols of M, plus one, then $|CONFIGS'_h(x)| \leq q \cdot (h+2)^{k+1} \cdot d^h = O(1)^h$. The difference is that now, the input head position n_0 satisfies $n_0 \leq h+1$. **Q.E.D.**

Thus $|CONFIGS'_h(x)|$ does not depend on $n = |x|$.

Theorem 18 $NTIME(t) \subseteq DTIME(O(1)^t)$.

Proof. The proof proceeds as in theorem 16 but we now enumerate over configurations in $CONFIGS'_h(x)$ rather than $CONFIGS_h(x)$. **Q.E.D.**

Theorems 16 and 18 remain valid if we use running complexity instead of accepting complexity (Exercise). The next result shows that reversal in the fundamental mode is bounded by a single exponential time.

Theorem 19 *Let* M *be a deterministic Turing machine accepting in* $r(n)$ *reversals. Then* $AcceptTime_{\mathrm{M}}(n) = n \cdot O_{\mathrm{M}}(1)^{r(n)}$ *(assuming the left-hand side is defined).*

Proof. Let M be a k-tape machine and $\overline{C}$ an accepting computation path on an input of length $n > 0$. The theorem follows if we show that the length of $\overline{C}$ is $n \cdot O_{\mathrm{M}}(1)^{r(n)}$. Call a step $C_{j-1} \vdash C_j$ *active* if any work head moves during this step from a non-blank symbol; otherwise the step is *inactive*. Thus, during an inactive step, the input head can freely move and the work heads may move from a blank symbol. We obtain a bound on the number of active steps. By assumption, $\overline{C}$ has $r \leq r(n)$ phases. Let m_i be the number of active steps in phase i $(i = 1, \ldots, r)$.

For any configuration C_j, recall that each head h $(h = 0, \ldots, k)$ tends in some direction $d_h \in \{-1, 0, +1\}$. The *potential* of tape h in C_j is the number of non-blank cells that would be encountered as head h moves in the direction d_h, starting from its current cell in C_j. If $d_h = 0$ the potential is defined to be zero. For example, if head h is at the rightmost non-blank symbol, and $d_h = +1$ then the potential is 1; but if $d_h = -1$ then the potential would be the total number of non-blank cells on tape h. The *potential* of C_j is the sum of the potentials of all work tapes (so we discount the potential of the input tape). Clearly the potential of C_j is at most the space usage in C_j. Therefore, at the start of phase i the potential is at most

$$k \sum_{v=1}^{i-1} m_v$$

since the number of active steps until that moment is $\sum_{v=1}^{i-1} m_v$ and each active step can create at most k new non-blank cells, where k is the number of work heads. Suppose inductively that $m_v \leq cn(k+1)^v$ for some $c > 0$. This is true for $v = 1$. Now each active step in a phase consumes at least one unit of potential, so m_i is bounded by the potential at the beginning of the ith phase:

$$m_i \leq k \sum_{v=1}^{i-1} m_v \leq k \sum_{v=1}^{i-1} cn(k+1)^v < cn(k+1)^i.$$

Therefore in r phases, the total number of active steps is

$$\sum_{v=1}^{r} m_v < cn(k+1)^{r+1} = n \cdot O_{\mathrm{M}}(1)^r.$$

It remains to bound the number of inactive moves. There are $O_{\mathrm{M}}(1)$ consecutive steps in which no head moves, and there are at most n moves by the input head in any phase. Hence there are at most $n + O_{\mathrm{M}}(m_i)$ inactive moves in the ith phase. Summing over all phases gives at most $nr + O_{\mathrm{M}}(1)^r$ inactive moves. **Q.E.D.**

Corollary 20 *DREVERSAL*$(r(n)) \subseteq$ *D-TIME-REVERSAL*$(n \cdot O(1)^{r(n)}, r(n))$

Remark: If reversals by input heads are not counted, then we get instead

$$\textit{DREVERSAL}(r) \subseteq \textit{D-TIME-REVERSAL}(n^2 O(1)^{r(n)}, r(n)).$$

2.7 Simulation by Space

We will present two techniques for simulating other resources so as to minimize deterministic space. These space simulation techniques are quite different than the time simulation techniques of the previous section.

Theorem 21 (Extended Savitch's theorem) *If* $t(n) \geq 2s(n) \geq \log n$ *then*

$$\textit{N-TIME-SPACE}(t(n), s(n)) \subseteq \textit{DSPACE}(s(n) \log \left(\frac{t(n)}{s(n)}\right)).$$

Proof. Given a nondeterministic machine M accepting in simultaneous time-space (t, s), it suffices to show that $L(\mathrm{M})$ is accepted in space $s \log(t/s)$ by some deterministic N. (We require $t(n) \geq 2s(n)$ simply to prevent $\log(t/s)$ from vanishing.) An input x of length n is accepted by M iff there is an accepting computation path $\overline{C}$ whose time-space is $(t(n), s(n))$. As in lemma 13, for any $h \geq 1$, let $\mathit{CONFIGS}_h(x)$

be the set of configurations of M on input x using space at most h. For configurations $C, C' \in CONFIGS_h(x)$, define the predicate $PATH_h(C, C', m)$ to be true if there is a sub-computation path from C to C' of length $\leq m$.

Claim A: Assume that the input x is freely available, we can evaluate $PATH_h(C, C', m)$ in deterministic space $O(m + h + \log n)$. *Proof of claim.* Consider the tree $T(C)$ rooted at C and whose paths comprise all those sub-computation paths from C of length $\leq m$ and involving configurations in $CONFIGS_h(x)$. It suffices to search for C' in $T(C)$ using a standard depth-first search. At any moment during the search, we are at some node labeled C'' in $T(C)$. We first check if C'' is the C' we are looking for. If not, we try to visit an unvisited successor of C'' in $CONFIGS_h(x)$, provided C'' is at depth less than m. If this is not possible, we backtrack up the tree. To backtrack, we must be able to generate the parent C''' of C'': this is easy if we had stored the instruction of M that transformed C''' into C''. The space to store this backtracking information is just $O(1)$ per level or $O(m)$ overall. We also need $O(h+\log n)$ to store the current node C'' and the target configuration C'. When we have searched the entire tree without finding C', we conclude that $PATH_h(C, C', m)$ is false. This proves our claim.

Claim B: Assume $h \geq \log n$ and that the input x is freely available, we can evaluate $PATH_h(C, C', m)$ in in deterministic space $O(h \log(2 + \frac{m}{h})$. *Proof of claim.* We use a following simple observation:

> (*) $PATH_h(C, C', m)$ holds iff both $PATH_h(C, C'', \lfloor m/2 \rfloor)$ and $PATH_h(C'', C', \lceil m/2 \rceil)$ hold, for some $C'' \in CONFIGS_h(x)$.

Thus we can evaluate $PATH_h(C, C', m)$ recursively: if $m \leq h$, we use claim A to directly evaluate $PATH_h(C, C', m)$ in space $O(h)$, as desired. Otherwise, if $m > h$, we use observation (*) to make two recursive calls to the predicate. Here are the details: To evaluate $PATH_h(C, C', m)$, assume tape 1 of the machine N stores the value h, tape 2 stores the current arguments (C, C', m) while tape 3 acts as the recursion stack. We then systematically enumerate all configurations $C'' \in CONFIGS_h(x)$. For each C'', we recursively evaluate $PATH_h(C, C'', m/2)$ and $PATH_h(C'', C', m/2)$ in this order. But before making first recursive call, we write the pair $(C', 0)$ on top of the recursion stack, where the Boolean value 0 indicates that this is the first of the two recursive calls. Then the current arguments on tape 2 are updated to $(C, C'', m/2)$ and we continue recursively from there. Similarly, before making the second recursive call, we write the pair $(C, 1)$ on the recursion stack and update the current arguments on tape 2 to $(C'', C', m/2)$. If either one of these recursive calls returns with "failure" (predicate is false), we generate the next configuration in $CONFIGS_h(x)$ and use it place of C''; in case C'' is the last configuration in our enumeration of $CONFIGS_h(x)$, we return from the current call $PATH_h(C, C', m)$ with "failure". If both recursive calls returns with "success" (predicate is true), we return from the current call with "success".

The recursion depth in evaluating $PATH_h(C, C', m)$ is $\lceil k \rceil = \lceil \log(m/h) \rceil \geq 1$. The space to store the arguments for each recursive call is $O(h + \log n)$ where the $\log n$ comes from having to represent the input head position (the actual input x need not be stored). So the total space used is $O((\log n + h)\log(m/h))$, which is $O(h \log(m/h))$ when $h \geq \log n$.

Finally, that x is accepted by M if and only if $PATH_h(C_0, C_a, m)$ is true for some $h \leq s(n)$, $m \leq t(n)$, where $C_a \in CONFIGS_h(x)$ is an accepting configuration and C_0 the initial configuration on input x. By modifying M, we may assume that C_a is unique in $CONFIGS_h(x)$. Since $s(n)$ and $t(n)$ are unknown, N must search for h and m systematically, using a doubly-nested loop:

```
for p = 1, 2, ...,
    for h = 1, ..., p,
        let m = h2^⌊p/h⌋ and C_0, C_a ∈ CONFIGS_h(x).
        Accept if PATH_h(C_0, C_a, m) is true,
```

The correctness follows from two remarks:
(i) For any given value of p, the maximum space used in the inner loop is $O(p)$. This is because the space to evaluate $PATH_h(C_0, C_a, m)$ is order of $h \lg(m/h) \leq p$.
(ii) If input x is accepted by M then the double-loop accepts for some $p = O(s \log(t/s))$; otherwise, the double-loop runs forever. **Q.E.D.**

We obtain several interesting corollaries, including the well-known result of Savage [34].

Corollary 22
(i) (Savitch's theorem) *If* $s(n) \geq \log n$ *then* $NSPACE(s) \subseteq DSPACE(s^2)$.
(ii) If $s(n) \geq n$ *then* $NTIME(s) \subseteq DSPACE(s)$.
(iii) If $s(n) \geq n$, $N\text{-}SPACE\text{-}REVERSAL(s, r) \subseteq DSPACE(s \log r)$.

Proof. (i) Use the fact that $NSPACE(s) = N\text{-}TIME\text{-}SPACE(O(1)^s, s)$ for $s(n) \geq \log n$.
(ii) This follows since $NTIME(t) \subseteq N\text{-}TIME\text{-}SPACE(2t, t)$ but $N\text{-}TIME\text{-}SPACE(2t, t) \subseteq DSPACE(t)$ by applying the theorem.
(iii) This follows since $N\text{-}SPACE\text{-}REVERSAL(s, r) \subseteq N\text{-}TIME\text{-}SPACE((n + s)r, s)$ and $N\text{-}TIME\text{-}SPACE((n + s)r, s) \subseteq DSPACE(s \log r)$ by applying the theorem. **Q.E.D.**

We remark that our formulation of the generalized Savitch theorem is motivated by the desire to combine (i) and (ii) of this corollary into one theorem.

If $s(n) = o(\log n)$, Savitch's theorem yields $NSPACE(s) \subseteq DSPACE(\log^2 n)$. Monien and Sudborough have improved this to

$$NSPACE(s) \subseteq DSPACE(s(n) \log n).$$

In chapter 7, we further improve upon Monien and Sudborough via a strengthening of theorem 21. Indeed, chapter 7 shows four distinct extensions of Savitch's theorem!

Corollary 23

(i) $DSPACE(n^{O(1)}) = NSPACE(n^{O(1)})$.

(ii) $DSPACE(O(1)^n) = NSPACE(O(1)^n)$.

This corollary justifies our notation '*PSPACE*' since we need not distinguish between '*D-PSPACE*' and '*N-PSPACE*'. The notations *EXPS* and *EXPSPACE* are likewise justified.

We next simulate deterministic reversals by deterministic space. Istvan Simon [35] had shown how to simulate simultaneous time-reversal by space. Our next result strengthens his result to space-reversal.

Theorem 24 *If* $s(n) \geq \log n$,

$$D\text{-}SPACE\text{-}REVERSAL(s, r) \subseteq DSPACE(r \log s).$$

Proof. Given a deterministic k-tape M accepting in space-reversal (s, r), it suffices to construct a deterministic N accepting $L(\mathrm{M})$ in space $r \log s$. For any configuration C in a computation path $\overline{C}$, the *trace* of C refers to the head tendencies in C (relative to $\overline{C}$) and to information that remains in C after we discard all the tape contents except for what is currently being scanned.[8] More precisely, the trace of C is defined as

$$\langle q, b_0, \ldots, b_k, n_0, \ldots, n_k, d_0, \ldots, d_k \rangle$$

where q is the state in C, the b_i's are the scanned symbols, the n_i's are the head positions, and the d_i's are the head tendencies. We remark that the n_i's are absolute positions, unlike our usual convention of using relative positions in configurations. To *simulate the jth step* means to compute the trace of the jth configuration of the computation path. We shall also need the concept of a *partial trace*: this is like a trace except any number of the b_i's can be replaced by a special symbol '$*$'. We say b_i is *unknown* if $b_i = *$. Thus a trace is also a partial trace although for emphasis, we may call a trace a *full* trace.

On tape 1 of N we assume that we had already computed a sequence

$$\tau_1, \tau_2, \ldots, \tau_m$$

where τ_i is the trace of the first configuration in phase i. To begin, we can always place the trace τ_1 of the initial configuration on tape 1. Call the last phase (phase

[8] Warning: in this book we use the term 'trace' for several similar but non-identical concepts. The reader should check each context.

m) represented in this sequence the *current phase*. This means that our simulation has reached some step in the mth phase.

To simulate a step of a phase, it turns out that we need to simulate k steps from previous phases and thus we can use recursion. To keep track of these recursive calls, we use tape 2 of N as a recursion stack. On the recursion stack, we will in general store a sequence of the form

$$\sigma_{i_1}, \sigma_{i_2}, \ldots, \sigma_{i_v} (v \geq 1)$$

where $m = i_1 > i_2 > \cdots > i_v \geq 1$ and each σ_{i_j} is a partial trace of a configuration in the i_jth phase. Furthermore, all the σ_{i_j}'s, except possibly for σ_{i_v}, are not full traces. Note that σ_{i_1} is always in the current phase (phase m). Intuitively, we are really trying to evaluate σ_{i_1} but its evaluation calls for the evaluation of σ_{i_2}, which calls for the evaluation of σ_{i_3}, etc. Thus σ_{i_v} is the top entry of the recursion stack.

This is how the recursive calls arise. Suppose that at some point in our simulation we managed to compute the full jth trace τ where the jth configuration belongs to the current phase. Also assume that the recursion stack contains τ as its only entry. We now want to simulate the $(j+1)$st step. Note that we can obtain the partial trace of the $(j+1)$st configuration from the full jth trace: the only unknowns about the $(j+1)$st trace are the symbols under any head that has moved in the transition from the jth configuration to the $(j+1)$st configuration. At this moment, we replace τ on the recursion stack by partial $(j+1)$st trace σ_{i_1} (as the only stack entry).

Now inductively, assume that we have $\sigma_{i_1}, \ldots, \sigma_{i_v}$ $(v \geq 1)$ on the stack. Let

$$\sigma_{i_v} = \langle q, b_0, \ldots, b_k, n_0, \ldots, n_k, d_0, \ldots, d_k \rangle$$

be the partial trace on top of the stack. We may assume that b_0 is known (i.e. not equal to $*$) since this is on the input tape. Let $h \geq 1$ be the smallest index such that $b_h = *$ (if σ_{i_v} is a full trace, then let $h = k+1$). We take following action depending on two cases:

(1) If $h \leq k$ then we want to determine b_h. Note that we know the position $(= n_h)$ of b_h and hence from the contents of tape 1, we can determine the phase preceding phase i_v in which cell n_h was last visited. If this cell had never been visited before, we can also determine this fact and conclude that b_h is the blank symbol $\sqcup$. Suppose this cell was last visited by configuration C in phase i_{v+1} $(i_v > i_{v+1} \geq 1)$. Note that in the configuration following C, head h must no longer scan cell n_h. Our goal is to compute the trace of C. We say the trace of C is *sought after* by σ_{i_v}. To do this, we copy the trace of the starting configuration of phase i_{v+1} to the top of the stack. This trace, $\sigma_{i_{v+1}}$, is available in tape 1.

(2) If $h = k+1$ then σ_{i_v} is a full trace. There are two cases: (i) If this is the trace sought after by the preceding partial trace $\sigma_{i_{v-1}}$ then we can fill in one of the

unknown symbols in $\sigma_{i_{v-1}}$ and erase σ_{i_v} from the top of the stack. (ii) If this is not the trace sought after by the preceding partial trace then we simply replace σ_{i_v} on the stack by its successor partial trace. By definition, if $v = 1$ then case (ii) is applied.

It is clear that we eventually convert the partial trace σ_{i_1} into a full trace, and hence repeat the cycle. We need to consider the action to take when we enter a new phase. Suppose $\tau = \sigma_{i_1}$ had just turned into a full trace. Just before we replace τ on the stack by its successor τ', as described in (2), we check whether τ is the beginning of a new phase, and if so we first store it on tape 1. In fact, the information indicating whether τ is the start of a new phase could already be determined as we initially form the partial trace corresponding to τ.

The total amount of space required on tapes 1 and 2 of N is $O(r \log s)$ since each partial trace uses $O(\log s)$ space. The assumption $s(n) \geq \log n$ is needed to represent the input head positions. **Q.E.D.**

Corollary 25
(i) (Istvan Simon) $\mathit{D\text{-}TIME\text{-}REVERSAL}(t, r) \subseteq \mathit{DSPACE}(r \log t)$.
(ii) $\mathit{DREVERSAL}(O(1)) \subseteq \mathit{DSPACE}(\log n) = \mathit{DLOG}$.
(iii) For $r(n) \geq \log n$, $\mathit{DREVERSAL}(r) \subseteq \mathit{DSPACE}(r^2)$.

To see (i) use the fact that $\mathit{D\text{-}TIME\text{-}REVERSAL}(t, r) \subseteq \mathit{D\text{-}SPACE\text{-}REVERSAL}(t, r)$. For (ii) and (iii), use the fact that

$$\mathit{DREVERSAL}(r) \subseteq \mathit{D\text{-}TIME\text{-}REVERSAL}(n \cdot O(1)^r, r).$$

As application of part (ii) of this corollary, we note that since the palindrome language can be accepted in linear time by an acceptor making three reversals, it can be accepted in log space.

Since time, space and reversals are not independent, it is worth pointing out some conditions under which the theorem $\mathit{D\text{-}SPACE\text{-}REVERSAL}(s, r) \subseteq \mathit{DSPACE}(r \log s)$ is stronger than its corollary (i), $\mathit{D\text{-}TIME\text{-}REVERSAL}(t, r) \subseteq \mathit{DSPACE}(r \log t)$. For instance, let $s(n) = \theta(n) = O(\log r(n))$. Then corollary (i) implies $\mathit{D\text{-}SPACE\text{-}REVERSAL}(s, r) \subseteq \mathit{DSPACE}(r \log n)$ while the theorem yields the stronger $\mathit{D\text{-}SPACE\text{-}REVERSAL}(s, r) \subseteq \mathit{DSPACE}(r \log \log n)$.

Remark: Rytter and Chrobak [33] have shown that the the converse of corollary (ii),

$$\mathit{DLOG} \subseteq \mathit{DREVERSAL}(O(1)),$$

holds, provided we do not count reversals made by the input head. This extra condition of Rytter and Chrobak is essential, as pointed out by Liśkiewicz [9]: Book and Yap [5] proved that all tally languages in $\mathit{DREVERSAL}(O(1))$ are regular. On the other hand, Mehlhorn and Alt (see section 11) has given a non-regular

[9] Private communication.

tally language. Thus corollary (ii) is a proper inclusion. Nevertheless, we cannot strengthen corollary (ii) to $DREVERSAL(O(1)) \subseteq DSPACE(s)$ for any $s(n) = o(\log n)$, because the language $\{a^n b^n : n > 0\}$ belongs to $DREVERSAL(O(1))$ but not to $DSPACE(s)$.

2.8 Simulation by Reversal

We consider simulation techniques that seek to minimize reversals. Reversal complexity seems much more 'powerful' than time and space complexity. For example, to double an arbitrarily large tape segment on a 2-tape Turing machines takes only 2 reversals – yet the time and space charged against this activity would be linear. Our intuition about reversals, relative to time or space, is much less developed. This is illustrated by a somewhat unexpected result from Baker and Book [2] about reversal complexity in the nondeterministic mode:

Theorem 26 $RE \subseteq NREVERSAL(2)$.

Proof. Without loss of generality, let L be a recursively enumerable language accepted by a simple Turing machine M. Suppose $x \in L$ and let $C_0, C_1, \ldots, C_k$ be the accepting computation path of M on x. Assume that all the C_i's are encoded by strings of the same length. We construct a two-tape nondeterministic N that accepts L as follows: on input x, N guesses some sequence $C_0 \# C_1 \# \cdots \# C_k$ on tapes 1 and 2 (so the two tapes have identical contents). Now head 2 reverses until it is at the #-separator between C_{k-1} and C_k; while doing this N can verify if C_k is an accepting configuration. Next, after reversing the direction of head 1, we can easily check that C_{k-1} (on tape 2) and C_k (on tape 1) are consecutive configurations of some computation path. If not, N immediately rejects; in particular, N rejects if $|C_{k-1}| \neq |C_k|$. Without any further head reversals, we can continue in this fashion to check that C_{i-1} (on tape 2) and C_i (on tape 1) are consecutive configurations of some computation path, for $i = k-1, k-2, \ldots, 1$. We must ensure that C_0 is the initial configuration on input x; but it is simple to ensure this (without any additional head reversals) while we are guessing the C_i's. **Q.E.D.**

This result, combined with the earlier

$$DREVERSAL(r) \subseteq DTIME(nO(1)^r),$$

tells us that there is no fixed complexity function f such that any language accepted nondeterministically in reversal $r(n)$ can be accepted deterministically in reversal $f(r(n))$ (contrast with theorem 18 and Savitch's theorem for time and space). Thus reversal seems to be rather different than time or space. Later in this book, we see that when reversal is simultaneously bounded with either time or space, then the corresponding complexity classes are better behaved.

2.8.1 Basic Techniques for Reversals

In this subsection, we show that reversal complexity in the fundamental mode is also relatively well-behaved. These results are from [10]. First we give a series of technical lemmas.

Lemma 27 (Natural Number Generation) *Given as input any integer $r > 0$ in unary, the string*

$$\bar{0}\#\bar{1}\#\bar{2}\#\cdots\#\overline{2^r-1}\#$$

can be generated by a 2-tape Turing machine M *making $O(r)$ reversals. Here $\bar{m}$ denotes the binary representation of the integer m of length r, prefixed with 0's if necessary.*

Proof. M first generates the pattern $(0^r\#)^{2^r}$ on tape 1. This can be done within $O(r)$ reversals, using a simple doubling method. Call each portion of the tape 1 between two consecutive # symbols a *segment*. Similarly, M generates a pattern $P_1 = (01)^{2^r}$ on tape 2.

Now M uses r *stages*, making a constant number of reversals per stage, to 'fix' the r bits in each *segment* in tape 1. (The final string $\#\bar{0}\#\cdots\#\overline{2^r-1}\#$ is going to be the contents of tape 1 at the end of the stages, so 'fixing' a bit means making it the symbol 0 or 1 that should appear finally.) For example, with $r = 3$, the rightmost bit of the $2^3 = 8$ segments alternates between 0 and 1, i.e. has the pattern $P_1 = 10101010$. The next bit has pattern $P_2 = 11001100$, and the final bit has pattern $P_3 = 11110000$.

Suppose that at the beginning of the $(i+1)$st $(i \geq 0)$ stage, M has fixed the last i bits for each segment on tape 1, and suppose the pattern

$$P_{i+1} = (0^{2^i}1^{2^i})^{2^{r-i}}$$

is inductively available on tape 2. Here the first bit of a segment refers to its least significant bit. Note that the $(i+1)$st bit of the jth segment is exactly the jth bit of pattern P_{i+1}.

In the $(i+1)$st stage M needs to know which bit in the jth segment is the $(i+1)$st bit. This is solved by placing a special mark on another track below the ith bit of each segment. These marks are easily updated at each stage. Now with only one phase, M can fix the $(i+1)$st bit for each segment of P_r on tape 1.

Using a constant number of sweeps, M can generate the pattern $P_{i+2} = (0^{2^{i+1}}1^{2^{i+1}})^{2^{r-i-1}}$ from P_{i+1}. At the end of the rth stage, the string $\bar{0}\#\bar{1}\#\bar{2}\#\cdots\overline{2^r-1}\#$ is on tape 1 of M. This completes the proof. **Q.E.D.**

A string of the form

$$x_1\#x_2\#\cdots\#x_n\#, n \geq 1$$

($x_i \in \Sigma^*$, $\# \notin \Sigma$) is called a *list* of n *items* (x_i is the ith item). A list is said to be in *normal form* if n is a power of 2 and each x_i has the same length. The next lemma shows how to convert any list into one in normal form.

Lemma 28 (List Normalization) *There is a 2-tape Turing machine* M *which, given a list* $x_1\#x_2\#\cdots\#x_n\#$ *of* n *items on its input tape, can construct another list* $y_1\#y_2\#\cdots\#y_{2^k}\#$ *in normal form using* $O(\log n)$ *reversals. Therefore,*

(a) $2^{k-1} < n \leq 2^k$ *;*

(b) each y_i $(i = 1, \ldots, 2^k)$ *has length* $\max_{i=1,\ldots,n} |x_i|$*; and*

(c) each y_i $(i = 1, \ldots, n)$ *is obtained by padding* x_i *with a prefix of zeroes and each* y_j $(j = n+1, \ldots, 2^k)$ *is a string of zeroes.*

Proof. First M computes the unary representation $z_0 \in \{0\}^*$ of $\max_{i=1,\ldots,n} |x_i|$ as follows: With $O(1)$ reversals, M initializes tape 1 to have all the odd numbered items from the original list and tape 2 to have all the even numbered items. So tape 1 and 2 contain the lists $x_1\#x_3\#x_5\#\cdots$ and $x_2\#x_4\#x_6\#\cdots$, respectively. In another pass, M compares x_{2i-1} on tape 1 with x_{2i} on tape 2 $(i = 1, 2, \ldots, \lceil n/2 \rceil)$, marking the longer of the two words. In $O(1)$ passes, M can produce a new list $z_1\#z_2\#\cdots\#z_{\lceil n/2 \rceil}\#$ of these marked words. M repeats this process: splits the list into two with roughly the same number of items, compares and marks the longer item of each comparison, produces a new list consisting of only the marked items. After $O(\log n)$ reversals, M is left with a list containing only one item. This item has the longest length among the x_i's. It is now easy to construct z_0.

The next goal is to construct a string of the form

$$(z_0\#)^{2^k} \quad (\text{where } k = \lceil \log_2 n \rceil).$$

Suppose we already have $(z_0\#)^{2^i}$. Using $x_1\#x_2\#\cdots\#x_n\#$ and $(z_0\#)^{2^i}$, M can compare n with 2^i: if $n \leq 2^i$ then we are done, otherwise we will construct $(z_0\#)^{2^{i+1}}$ from $(z_0\#)^{2^i}$ using $O(1)$ reversals.

Finally, from $(z_0\#)^{2^k}$, we can easily construct the desired $y_1\#y_2\#\cdots\#y_{2^k}\#$. **Q.E.D.**

A more complicated problem is computing the transitive closure of a matrix. It turns out that the key to computing transitive closure is a fast matrix transposition algorithm:

Lemma 29 (Matrix Transposition) *There is a 2-tape Turing machine* M *such that, given an* $n \times m$ *matrix* $A = (a_{ij})$*,* M *can compute the transpose* A^T *of* A *using* $O(\log \min\{m, n\})$ *reversals. Here we assume both* A *and* A^T *are stored in row major form.*

Proof. Because of the preceding list normalization lemma, we may assume that A satisfies the property $m = 2^k$ for some integer $k \geq 1$ and each entry of A has the same length. Let us first show how to compute A^T in $O(\log m)$ reversals.

For each $i = 0, 1, \ldots, k$, and for $j = 1, 2, \ldots, 2^i$, let $A_j^{(i)}$ denote the $n \times 2^{k-i}$ matrix consisting of the columns indexed by the numbers

$$((j-1)2^{k-i}+1), ((j-1)2^{k-i}+2), \ldots, (j2^{k-i}).$$

For example, $A_j^{(k)}$ is the jth column of A and for each i,

$$A = A_1^{(i)}|A_2^{(i)}|\cdots|A_{2^i}^{(i)}$$

(where | means concatenation of matrices). An *i-row representation* of A is a string consisting of the row-major forms of $A_1^{(i)}, A_2^{(i)}, \ldots, A_{2^i}^{(i)}$, listed in this order, separated by '$'.

Let $A^{(i)}$ denote the i-row representation of A. The lemma follows if we can show how to obtain $A^{(i+1)}$ from $A^{(i)}$ in $O(1)$ reversals. This is because the input is $A^{(0)}$ (the row major form of A) and the desired output is $A^{(k)}$ (the column major form of A).

In order to do the $A^{(i)} \rightarrow A^{(i+1)}$ transformation, we need some auxiliary data. Define the block pattern:

$$P^{(i)} = ((w_0^{2^{k-i-1}} w_1^{2^{k-i-1}} \#)^n \$)^{2^i} (i = 0, \ldots, k-1)$$

where $w_0 = 0^s$ and $w_1 = 1^s$, and s is the length of each entry of A. Each $w_0^{2^{k-i-1}}$ (respectively, $w_1^{2^{k-i-1}}$) 'marks' a left (respectively, right) half of the rows of $A_j^{(i)}$, $j = 1, 2, \ldots, 2^i$. $P^{(i)}$ helps us to 'separate' the rows of each $A_j^{(i)}$ into 'left half' and 'right half'.

We may inductively assume that each tape has two tracks with the following contents: $A^{(i)}$ is on track 1 and $P^{(i)}$ is positioned directly underneath $A^{(i)}$ on track 2. The case $i = 0$ is initialized in $O(1)$ reversals. A copy of $P^{(i+1)}$ can be made on track 2 of tape 2 using $P^{(i)}$. Now it is easy to obtain $A^{(i+1)}$ on track 1 of tape 2.

It is not hard to show that A^T can also be computed in $O(\log n)$ reversals. We leave the details as an exercise. Hence if we first determine the smaller of m and n in $O(1)$ reversals, we can then apply the appropriate matrix transposition algorithm to achieve the $O(\log \min\{m, n\})$ bound. **Q.E.D.**

Lemma 30 (Parallel Copying Lemma) *There is a 2-tape Turing machine* M *which, given an input* $x = 0^n \# x_1 x_2 \cdots x_m$, *where* $x_i \in \Sigma^*$, $(i = 1, \ldots, m)$, *produces string* $y = x_1^{2^n} x_2^{2^n} \cdots x_m^{2^n}$ *within* $O(n)$ *tape reversals, where we assume that* M *can recognize the boundary between blocks* x_i *and* x_{i+1}, $i = 1, \ldots, m-1$.

Proof. Using $O(n)$ reversals, M can get a string $S = (x_1 x_2 \cdots x_m)^{2^n}$. Notice that $x_1^{2^n} x_2^{2^n} \cdots x_m^{2^n}$ is the transpose of S if we regard S as being a $m \times 2^n$ matrix. **Q.E.D.**

From the last two important lemmas, we get immediately,

Lemma 31 (Matrix Multiplication) *There is a 2-tape Turing machine* M *such that, given two $n \times n$ Boolean matrices $A = (a_{ij})$ and $B = (b_{ij})$ in row major form,* M *computes their product $AB = (c_{ij})$ in $O(\log n)$ reversals.*

Proof. By lemma 29, we can obtain the transpose B^T of B within $O(\log n)$ reversals. Let

$$A = (a_{11} \cdots a_{1n} a_{21} \cdots a_{2n} \cdots a_{n1} \cdots a_{nn})$$

and

$$B^T = (b_{11} \cdots b_{n1} b_{12} \cdots b_{n2} \cdots b_{1n} \cdots b_{nn}).$$

By lemma 30, we can get

$$A_1 = (a_{11} \cdots a_{1n})^n (a_{21} \cdots a_{2n})^n \cdots (a_{n1} \cdots a_{nn})^n$$

and

$$B_1^T = (b_{11} \cdots b_{n1} b_{12} \cdots b_{n2} \cdots b_{1n} \cdots b_{nn})^n$$

within $O(\log n)$ reversals. Then $O(1)$ more reversals will give AB. So $O(\log n)$ reversals are enough for $n \times n$ Boolean matrix multiplication. **Q.E.D.**

Lemma 32 (Matrix Transitive Closure) *There is a 2-tape Turing machine* M *such that given an $n \times n$ Boolean matrix $A = (a_{ij})$, stored in row major form,* M *computes the transitive closure A^* of A in $O(\log^2 n)$ reversals.*

Proof. Since $A^* = (E + A)^m$ for any $m \geq n$, where E is the identity matrix, we can reduce this problem to $\log n$ matrix multiplications. **Q.E.D.**

Sorting seems a very important tool for reversal complexity. Here we give an upper bound for sorting.

Lemma 33 (Sorting) *There is a 2-tape Turing machine* M *that, given a list*

$$x_1 \# x_2 \# \cdots \# x_n \#$$

on its input tape, can construct the sorted list

$$x_{\pi(1)} \# x_{\pi(2)} \# \cdots x_{\pi(n)} \#$$

in $O(\log n)$ reversals. Here each item x_i is assumed to have the form $(k_i \& d_i)$ where k_i is a binary number representing the sort key, d_i is the data associated with k_i and & some separator symbol.

Proof. By lemma 28, we can assume that the input list is in normal form. We will be implementing a version of the well-known merge sort. The computation will be accomplished in k phases, where $n = 2^k$. To initialize the first phase, we use the parallel copying lemma to construct on tape 1 a string of the form

$$(x_1\$)^n \#(x_2\$)^n \# \cdots \#(x_n\$)^n \#$$

(for some new symbol \$) using $O(\log n)$ reversals.

For $i = 1, 2, \ldots, k$, let $f(i) = 2^k - \sum_{j=1}^{i-1} 2^j = 2^k - 2^i + 2$. So $f(1) = 2^k$ and $f(k) = 2$. At the beginning of the $(i+1)$st phase ($i = 0, 1, \ldots, k-1$), we will inductively have on tape 1 a string of the form

$$B_1 \# B_2 \# \cdots \# B_{2^{k-i}} \#$$

where each B_j has the form

$$(x_{j,1}\$)^{f(i+1)} \% (x_{j,2}\$)^{f(i+1)} \% \cdots \% (x_{j,2^i}\$)^{f(i+1)}$$

where $x_{j,1}, x_{j,2}, \ldots, x_{j,2^i}$ are distinct items from the original list, already sorted in non-decreasing order. Call B_i the ith *block*. The substring in a block between two consecutive '%' is called a *subblock*. Observe phase 1 has been properly initialized above.

We also need some auxiliary data to carry out the induction. Let $w_0 = 0^s 1$ where $s = |x_i|$ (for all i). For this, we assume that at the beginning of the $(i+1)$st phase, we also store on tape 1 the patterns

$$P_i = \left(w_0^{2^i} \#\right)^n$$

and

$$Q_i = \left(w_0^{f(i)} \#\right)^n$$

Note that Q_i can be easily obtained from P_{i-1} and Q_{i-1} within $O(1)$ reversals and P_i can be obtained from P_{i-1} in another $O(1)$ reversals. Again, phase 1 is easily initialized.

Let us now see how we carry out phase i+1. First we split the string $B_1 \# B_2 \# \cdots \# B_{2^{k-i}} \#$ into two lists containing the odd and even numbered blocks: tape 1 now contains $B_1 \# B_3 \# \cdots \# B_{2^{k-i}-1} \#$ and tape 2 contains $B_2 \# B_4 \# \cdots \# B_{2^{k-i}} \#$. This can be done using $O(1)$ reversals.

Our goal is to 'merge' B_{2j-1} with B_{2j}, for all $j = 1, 2, \ldots, 2^{k-i-1}$ in parallel. Let

$$B_{2j-1} = (y_1\$)^{f(i+1)} \% (y_2\$)^{f(i+1)} \% \cdots \% (y_{2^i}\$)^{f(i+1)}$$

and

$$B_{2j} = (z_1\$)^{f(i+1)} \% (z_2\$)^{f(i+1)} \% \cdots \% (z_{2^i}\$)^{f(i+1)}$$

We begin by comparing the first copy of y_1 with the first copy of z_1. Suppose $y_1 \geq z_1$ ('$\geq$' here is the ordering on the items, as defined by the sort key). Then we 'mark' the first copy of z_1 and move head 2 to the first copy of z_2 in the block B_{2j}. We can compare the second copy of y_1 with this copy of z_2, marking the smaller of y_1 and z_2. If y_1 is smaller, we move to the first copy of y_2 and next compare y_2 with z_2. Otherwise, z_2 is smaller and we next compare the 3rd copy of y_1 with the first copy of z_3. In general, suppose we compare (some copy of) y_i with (some

copy of) z_j. We mark the smaller of the two. If y_i is smaller, we next move head 1 to the first copy of y_{i+1} and move head 2 to the next copy of z_j and proceed with comparing these copies of y_{i+1} and z_i; Otherwise, z_j is smaller and the roles of y and z are exchanged in the description. (For correctness, we will show below that there is a sufficient number of copies of each item.)

Eventually, one of the blocks is exhausted. At that point, we scan the rest of the other block and mark the first copy of each remaining item. Notice that each subblock now contains exactly one marked copy. We then proceed to the next pair of blocks (B_{2j+1} and B_{2j+2}).

After we have compared and marked all the pairs of blocks, we will get two strings S_1 and S_2, with one copy of an item in each subblock of each block marked, on the two tapes, respectively. Our goal is to produce the merger of B_{2j-1} and B_{2j} from S_1 and S_2. Call the merged result B'_j. We will scan S_1 and output on tape 2 a partially instantiated version of B'_j. Our preceding marking algorithm ensures that if a marked copy of item w in S_1 is preceded by h copies of w then w has been compared to and found larger than h other items in S_2. In the merge result, we want to place these h smaller items before w. Since each item should be repeated $f(i+2)$ times in its subblock in B'_j we must precede w with

$$h(1 + |z_1\$|f(i+2))$$

blank spaces to accommodate these h subblocks which will be placed there in another pass. With the help of the pattern Q_{i+2}, we can leave the required amount of blank spaces for each subblock in B'_j. More precisely, we make a copy of Q_{i+2} in a track of tape 2 and use it as a 'template' which has the block and subblock structure already marked out. As we scan string S_1, for each unmarked copy of an item preceding its marked version, we skip a subblock of Q_{i+2}. When we reach a marked version of item w in S_1, we will copy that $f(i+2)$ successive copies of w into a subblock of Q_{i+2}. To see that there are enough copies of w on S_1 following this marked w, observe that there are a total of $f(i+1)$ copies of w in its subblock in S_1 and since at most 2^i copies of w precedes its marked version, there are at least $f(i+1) - 2^i \geq f(i+2)$ copies left. When we finish this process, we scan the partially formed B'_j and the string S_2 simultaneously, and fill in the remaining subblocks of B'_j. We finally have a new list of blocks:

$$B'_1 \# B'_2 \# \cdots \# B'_{2_{k-i-1}} \#$$

where each B'_j has the required form

$$(x_{j,1}\$)^{f(i+2)} \% (x_{j,2}\$)^{f(i+2)} \% \cdots \% (x_{j,2^{i+1}}\$)^{f(i+2)}.$$

This completes the proof. **Q.E.D.**

For some applications, we need the sorting algorithm to be stable:

Corollary 34 (Stable Sorting) *The above sorting algorithm can be made stable in this sense: if the output list is*

$$x_{\pi(1)}\#x_{\pi(2)}\#\cdots\#x_{\pi(n)}\#$$

where each $x_{\pi(i)}$ *has the key-data form* $k_{\pi(i)}\&d_{\pi(i)}$*, then for each* $i = 1, \ldots, n-1$*,* $k_{\pi(i)} = k_{\pi(i+1)}$ *implies* $\pi(i) < \pi(i+1)$*.*

Proof. To do this, we first number, in another track (say track K) of the tape, the items in the string to be sorted sequentially. This takes $O(\log n)$ reversals. Next use the method of lemma 33 to sort the string. After sorting, those items with a common sort key will form a contiguous block. We apply the sorting method again to each block of items, where we now sort each block by the numbers of items (as written on track K). To do this in $O(\log n)$ reversals, we must do this second sorting for all the blocks simultaneously − our above method can be modified for this. **Q.E.D.**

2.8.2 Reversal is as powerful as space, deterministically

We now have the tools to prove the main result of this subsection, which is an efficient simulation of a space-bounded computation by a reversal-bounded computation (all deterministic). The idea is roughly this: to simulate a machine using $s(n)$ space, we first use the Natural Number Generation lemma to generate all configurations using space $s(n)$. Then we use the Parallel Copying Lemma to obtain a string S containing *many* copies of these configurations. With two identical copies of S, we can march down these two copies in a staggered fashion (as in the proof of the Baker-Book theorem), to extract a computation path. This method is reversal efficient because most of the reversals are made when we generated S.

Theorem 35 *Suppose* $s(n) = \Omega(\log n)$*. Then any language* $L \in DSPACE(s(n))$ *can be accepted by a 2-tape deterministic Turing machine M within* $O(s(n))$ *reversals.*

Proof. First we assume that $s(n)$ is reversal constructible and $s(n) = \Omega(n)$.

Let N be a deterministic Turing machine N that accepts L in space $s(n)$. We construct a 2-tape Turing machine M that accepts L in $O(s(n))$ reversals. Fix any input x of length n.

(a) We first construct an integer $s = \theta(s(n))$ in unary, using s reversals (recall the definition of reversal constructible).

(b) Given an input x of length n, M first generates a string of the form

$$W_1 = \bar{0}\#\bar{1}\#\bar{2}\#\cdots\#\overline{2^s - 1}\#$$

We can suppose that all the possible configurations of N on input x are included in the string W_1.

(c) From string W_1, using $O(1)$ reversals, we can construct another string

$$W_2 = z_0 \# z_1 \# \cdots \# z_{2^s-1} \#$$

such that $|z_m| = s$ and z_m is the 'successor configuration' of configuration $\overline{m}$ in the string W_1, $m = 0, 1, \ldots, 2^s - 1$. If $\overline{m}$ is a terminal configuration, we may indicate this by using some suitable z_m.

Combining strings W_1 and W_2 into one tape, we get a string

$$W_3 = u_1 \# u_2 \# \cdots \# u_{2^s}$$

with each u_m $(m = 1, 2, \ldots, 2^s)$ is a 2-track string of the form

C_m
C'_m

where C_m and C'_m are configurations of N on input x such that $C_m \vdash C'_m$.

(d) Next we construct two identical strings S_1 and S_2 on tapes 1 and 2 of M, respectively, using $O(s)$ reversals:

$$S_1 = S_2 = \left((u_1\$)^{2^s} \#(u_2\$)^{2^s} \# \cdots \#(u_{2^s}\$)^{2^s}\%\right)^{2^s}$$

We will call each substring of the form $(u_1\$)^{2^s} \#(u_2\$)^{2^s} \# \cdots \#(u_{2^s}\$)^{2^s}$ (as determined by two consecutive %-symbols) a *block* of S_1 or of S_2. For each $j = 1, 2, \ldots, 2^s$ we call the substring $(u_j\$)^{2^s}$ a u_j *-segment.*

(e) With these two strings, we now construct the computation path

$$P = C_0 \vdash C_1 \vdash \cdots \vdash C_i \vdash \cdots$$

of N on input x in another $O(1)$ reversals as follows.

(e1) Consider the first u_{i_1}-segment in S_1 where the upper track of u_{i_1} contains the initial configuration C_0 of N on input x. Begin by marking the first copy of u_{i_1} in this segment and place head 1 between the first and second copies of u_{i_1} in the u_{i_1}-segment. (By 'marking' of a copy of u_{i_1}, we mean a special symbol is placed at the end of that copy on a separate track. This marking amounts to nothing in the following procedure, but helps the visualization of the proof.) Moreover, place head 2 at the beginning of the first block of string S_2;

(e2) Inductively, suppose we have already found and marked a sequence $u_{i_1}, u_{i_2}, \ldots, u_{i_q}$ on the string S_1 on tape 1 such that the lower track of u_{i_j} is identical to the upper track of $u_{i_{j+1}}$, $j = 1, 2, \ldots, q-1$. Hence the upper track of u_{i_l} contains the configuration C_l in the above path P for $l = 1, 2, \ldots, q$. Suppose also that head 1 is placed between the first and second copies of u_{i_q} in the qth block in S_1 and that head 2 is placed at beginning of the qth block in S_2. Our goal is to find a segment $(u_{i_{q+1}}\$)^{2^s}$ in the qth block of S_2 such that the lower track of u_{i_q} is identical to the upper track of $u_{i_{q+1}}$. Let C_{q+1} be the configuration in the lower track of u_{i_q}. If C_{q+1} is accepting (respectively, terminal), then we accept (respectively, reject) at once. Otherwise, we look for an occurence C_{q+1} in the first upper configuration of each segment of the qth block of S_2. Note that to do this without any head reversals, we use the $2^s - 1$ successive copies of C_{q+1} in the current segment of S_1.

We will surely find the desired segment $(u_{i_{q+1}}\$)^{2^s}$ in the qth block of the string S_2. In particular, if the $2^s - 1$ copies of C_{q+1} in the current segment are used up, the desired $u_{i_{q+1}}$ must be the very last segment (the u_{2^s}-segment) in the qth block. Head 2 is now between the first and second copies of $u_{i_{q+1}}$ in the $u_{i_{q+1}}$-segment.

Our goal now is to return to the inductive basis: with head 1 between the first and second copies of $u_{i_{q+1}}$ in the $q+1$st block of S_1, and head 2 at the beginning of the $q+1$st block of S_2. But the procedure just described can be applied with simple changes (viz., we reverse the roles of S_1 and S_2 and look for the first occurence of $u_{i_{q+1}}$ in the next block of S_1). When this is done, we are ready for next iteration looking for $u_{i_{q+2}}$.

If we exhaust the entire string S_1 without accepting or rejecting in the above procedure, then N must be in a loop. Then M will reject.

We now return to two assumptions we made at the beginning of this proof: (a) First we indicate how to modify the above proof for the case $s(n) = o(n)$. The above assumed that the entire input string x is stored with each configuration. This information is now omitted, although the input head position is retained. Then by sorting the string W_1 by the positions of the input head, followed by one sweep of the input x, we are able to record the 'current input symbol' into each of the modified configurations. A similar sort can be applied to W_2 in its construction. Then the entire procedure can be carried out as before. (b) Finally, we show how to avoid the assumption that $s(n)$ is reversal constructible. The problem arises because we cannot do the usual trick of trying successive values of s by increments of one. This is because reversal, unlike space, is not reusable and it would lead to $s(n)^2$ reversals overall. But we can easily fix this by doubling the value of s at each stage ($s = 1, 2, 4, 8, \ldots$, etc). **Q.E.D.**

Corollary 36 *For any* $s(n) = \Omega(\log n)$,

$$DSPACE(s) \subseteq DREVERSAL(s).$$

Thus, for deterministic Turing machines, reversal as a complexity resource is at least as powerful as space. Combined with Savitch's result, we obtain:

Corollary 37 *For any* $s(n) = \Omega(\log n)$,

$$NSPACE(s(n)) \subseteq DREVERSAL(s(n)^2).$$

Using the fact that $DREVERSAL(r) \subseteq DSPACE(r^2)$ for $r(n) = \Omega(\log n)$ (see previous section), we get important consequences:

Corollary 38

$$PLOG := DSPACE(\log^{O(1)} n) = DREVERSAL(\log^{O(1)} n)$$

$$PSPACE := DSPACE(n^{O(1)}) = DREVERSAL(n^{O(1)})$$

$$EXPS := DSPACE(O(1)^n) = DREVERSAL(O(1)^n)$$

In short, in the fundamental mode of computation, space and reversals are polynomially related.

Finally, it turns out that the above results and techniques yields a tape-reduction theorem of the form:

Theorem 39 (Tape reduction for reversals) *Let* $r(n) = \Omega(\log n)$. *If* L *is accepted by a determinisitic multitape machine within* $r(n)$ *reversals then it is accepted by a 2-tape deterministic machine within* $O(r(n)^2)$ *reversals.*

Further results on reversal complexity can be found in [9].

2.8.3 Reversal is more powerful than time, deterministically

We now prove a result of Liśkiewicz [26] showing that deterministic reversal is stronger than deterministic time by a square factor:

Theorem 40 *For all complexity function* $t(n) \geq n$, $DTIME(t) \subseteq DREVERSAL(\sqrt{t})$.

A fundamental problem that we need to solve is the problem of retrieving data from a table. We have already seen this problem, but let us now give it an abstract description. A *table* T is just a $m \times 2$ matrix,

$$T = k_1\$d_1\#k_2\$d_2\#\cdots\#k_m\$d_m, \qquad (k_i, d_i \in \Sigma^*).$$

Each k_i is called a *key* and d_i is called the *data item associated to* k_i. We may assume that the keys k_i's are distinct. The *retrieval problem* is formulated as follows: given the table T on tape 1 and a key $k \in \Sigma^*$ on tape 2, to locate k in T and then write its associated data d out to some specified output tape; if k is not a key in T, we output some special symbol. Using the previous techniques, we can solve this problem in $O(\log m)$ reversals: first, we generate $(k\$)^m$ on tape 3 in $O(\log m)$ reversals. Then we can search for occurence of k in T in the obvious way: compare the ith copy of k on tape 3 to k_i in tape 2. If they are equal, we can output d_i on the output tape, otherwise, we go to the next i.

A key idea in Liśkiewicz's proof is what we shall call *turbo-charged retrieval*: the table T is now replaced by

$$T' = (k_1\$d_1\$)^m\#(k_2\$d_2\$)^m\#\cdots\#(k_m\$d_m\$)^m$$

and the search argument k is replaced by $(k\$)^m$. We want the output to be $(d\$)^m$ where d is the data associated to k.

Lemma 41 *The turbo-charged retrieval problem can be solved in $O(1)$ reversals.*

Proof. We proceed as in the retrieval method above that uses $O(\log m)$ reversals, except that we skip the $O(\log m)$ reversals needed to make m copies of k. Once a copy of the key k is located in T', it is a simple matter to write out the desired output $(d\$)^m$ in $O(1)$ additional reversals. **Q.E.D.**

Next, we reduce the problem of simulating a deterministic machine M that accepts in time $t(n)$ to a sequence of retrievals. On input of length n, let $h = \sqrt{t(n)}$. If C, D are of configurations of M, we say that D is a *h-successor* of C if C derives D in exactly h steps, unless the computation path from C terminates in less than h steps in which case D is the terminal configuration. Intuitively, we want to construct a table T whose pairs (k_i, d_i) correspond to pairs (C, D) where D is a h-sucessor of C. Then, starting from the initial configuration C_0, we can do a sequence of h retrievals from T to obtain $C_1, C_2, \ldots, C_h$ where C_{i+1} is the h-successor of C_i. Of course, we should turbo-charge T and C_0 so that we only use $O(1)$ reversals per turbo-charged retrieval. The problem with this scenario is that the turbo-charging is too expensive because there are $\Omega(2^t)$ distinct configurations in T and this would require $\Omega(t)$ reversals for turbo-charging.

This brings us to the second idea of the simulation: we shall consider *fragments* of configurations. Let M have k work tapes. We first partition each tape (input tape as well as work tapes) into *h-blocks* where each block consists of h consecutive cells. We may assume that an h-block occupies the cells numbered $(i-1)h+1, (i-1)h+2, \ldots, i \cdot h$ for some $i \in \mathbb{Z}$; we may call this the ith block. An *extended h-block* is just 3 consecutive blocks on some tape; these blocks are called the *left-*, *center-* and *right-blocks*, respectively, of the extended block. An *h-fragment* is a sequence

$$F = F_0\%F_1\%\cdots\%F_k$$

where each F_i is the contents of an extended h-block of tape i. (We simply say 'block', 'fragment', etc, when the h is understood or irrelevant.) The 'contents' of a cell shall include the current state q if that cell is currently scanned: if a cell contains symbol a and is currently being scanned, then its contents is the pair (q, a). We say that the fragment $F = F_0\% \cdots \%F_k$ is *centered* if, for each $i = 0, \ldots, k$, some cell in the center-block of F_i is being scanned. Note that in a centered fragment, the state q is duplicated $k + 1$ times. For any configuration C, there is a unique centered h-fragment associated to C.

The importances of h-fragments is that there are $O(1)^h$ of them. If F is a centered h-fragment, we define an h-fragment G to be the *h-successor* of F as follows: let F be associated to some configuration C, and D be the h-successor of C. Then G is the h-fragment obtained from D by considering the contents of the blocks used in F. Note that G need not be centered; it is also easy to see that G is uniquely defined (it is independent of our choice of C). Let T_h be the search table whose keys are precisely all the $O(1)^h$ centered h-fragments, and the data associated to a fragment F is just the h-successors G of F.

Lemma 42 *Given h in unary, we can set up the search table T_h in $O(h)$ reversals.*

Proof. We first generate a sequence

$$k_1 \# k_2 \# \cdots \# k_m$$

of all centered h-fragments. This is done using the number generation technique and discarding those "numbers" that do not correspond to centered h-fragments. In $O(1)$ reversals, convert this to the trivial identity table:

$$T_h = k_1 \$ d_1 \# k_2 \$ d_2 \# \cdots \# k_m \$ d_m$$

where $d_i = k_i$ for each i. Next, we 'scan' this table h times, and each time we replace the data d_i by its 1-successor fragment. Each 'scan' actually takes $O(1)$ reversals. **Q.E.D.**

We then turbo-charge table T_h to

$$T'_h = (k_1 \$ d_1 \$)^m \# (k_2 \$ d_2 \$)^m \# \cdots \# (k_m \$ d_m \$)^m$$

This amounts to parallel copying, and can be done in $O(h)$ reversals. The final step is to carry out the simulation.

We need to describe the representation of configurations. Let C be a configuration of M. For $i = 0, \ldots, k$, let the nonblank contents W_i of tape i be contained in blocks ℓ to u:

$$W_i = B_\ell B_{\ell+1} \cdots B_{u-1} B_u.$$

Let us note that W_0 always uses $\lceil n/h \rceil$ blocks; but for $i = 1, \ldots, k$, we initially represent only one block in W_i (this block is all blank except for the indication of

the position of head i). During the simulation, we will add more blocks of blanks as needed. Normally, we would represent C as $W_0\%W_1\%\cdots\%W_k$. But to turbo-charge the representation, we represent C as $C = W_0'\%W_1'\%\cdots\%W_k'$ where

$$W_i' = (B_\ell\$)^m\#(B_{\ell+1}\$)^m\#\cdots\#(B_u\$)^m.$$

We now return to the proof of the main theorem. We shall operate in *stages*. In stage $i = 1, 2, 3, \ldots$, we store the value $h = 2^i$ in unary on tape 1. Using the stored valued of h, we can set up the turbo-charged table T_h on tape 2, and the turbo-charged initial configuration C_0 on tape 3. Setting up tapes 2 and 3 each takes $O(h)$ reversals. We now do the following "block simulation step" for a total of h times:

Block Simulation Step. Inductively, let tape 3 contain some turbo-charged configuration C. We first extract onto tape 4 the extended blocks corresponding to head positions on each tape of M:

$$(B_L^0\$)^m\#(B_C^0\$)^m\#(B_R^0\$)^m\%\cdots\%(B_L^k\$)^m\#(B_C^k\$)^m\#(B_R^k\$)^m. \tag{2.5}$$

Here, $B_L^iB_C^iB_R^i$ represents an extended block of tape i. This can be done in $O(1)$ reversals. Using the matrix transpose technique, in $O(1)$ reversals, we convert this into

$$(B_L^0B_C^0B_R^0\$)^m\#(B_L^1B_C^1B_R^1\$)^m\#\cdots\#(B_L^kB_C^kB_R^k\$)^m.$$

Doing this one more time, we convert it to the "turbo-charged h-fragment"

$$(F\%)^m = (B_L^0B_C^0B_R^0\#B_L^1B_C^1B_R^1\#\cdots\#B_L^kB_C^kB_R^k\%)^m.$$

We can use this as the search key for a turbo-charged retrieval on the table T_h. We may assume that the retrieved data $(G\%)^m$ is placed in tape 4, replacing $(F\%)^m$. By reversing the above process, we convert $(G\%)^m$ into the form equation (2.5) in $O(1)$ reversals; this is still stored in tape 4. Then in $O(1)$ reversals, we can update the configuration C in tape 3 using the contents of tape 4. This ends a "block simulation step". The number of reversals in $O(1)$.

Clearly, if C is the configuration at time t then after a block simulation step, C is the configuration at time $t+h$ (or else it is the terminal configuration). Thus after h block simulation steps, we have reached terminal configuration if the computation path terminates in $\leq h^2$ steps. If a terminal configuration, we can accept or reject accordingly. Otherwise, we double h and go to the next stage. The total number of reversals in stage h is $O(h)$. Clearly if the input is accepted in $\leq t(n)$ steps then the value of h is bounded by $2\sqrt{t(n)}$. Hence the total number of reversals is easily seen to be $O(\sqrt{t(n)})$. This proves the theorem of Liśkiewicz.

2.9 Complementation of Space Classes

In this section, we prove that deterministic and nondeterministic space classes, with some reasonable technical restrictions, are closed under complement. This result for nondeterministic space classes solves a major open problem problem dating back to Kuroda in 1964. The solution was independently obtained by Immerman [22] and Szelepcsényi [37]. Their solution was a surprise in two ways: the solution was surprisingly simple, and many researchers had expected the opposite result. The technique for this result has wider applications that the interested reader may further pursue (e.g., [28, 38, 24, 6]).

This section uses running complexity rather than accepting complexity. Recall our notation for running complexity classes uses the subscript 'r', as in $DSPACE_r(s)$ or $NSPACE_r(s)$. It is also crucial that the complexity function $s = s(n)$ has the property that for all $n \in \mathrm{N}$, $s(n)$ is defined and $< \infty$. Suppose a machine M has such a running space complexity $s(n)$. Then for all computation paths, on any input, use only a finite amount of space. It is important to realize that this does not preclude infinite computation paths.

2.9.1 Complement of Deterministic Classes

To motivate the proof, observe that deterministic running time classes are closed under complementation:

$$co\text{-}DTIME_r(t) = DTIME_r(t) \tag{2.6}$$

In proof, suppose a deterministic acceptor M runs in time t. We can easily define a deterministic acceptor N that runs in time t such that $co\text{-}L(\mathrm{M}) = L(\mathrm{N})$: N simply rejects iff M accepts, but in all other respects, N behaves exactly as M. This proves (2.6). Note that the restriction $t(n) < \infty$ is essential and implies that M always halts. The main result of this subsection is to show the space analogue of (2.6) using a technique of Sipser [36].

Theorem 43 *(Complement of deterministic space classes) If $s(n) < \infty$ for all n, then $co\text{-}DSPACE_r(s) = DSPACE_r(s)$.*

The above trick of reversing the roles of acceptance and rejection is insufficient because the fact that M runs in a finite amount of space does not guarantee that M halts. The proof of theorem 43 therefore hinges upon the ability to guarantee halting; indeed it is easily seen to be an immediate consequence of:

Lemma 44 *For every deterministic Turing acceptor* M *that runs in space $s(\cdot)$, there is a deterministic* N *that runs in space s such that $L(\mathrm{M}) = L(\mathrm{N})$ and* N *halts on all computation paths.*

One idea for proving this result is to keep two running counts for the number of steps t taken and the amount of space s used: if $t > nO_{\mathrm{M}}(1)^s$, we know that we must be looping. However, this proof requires that $s(n) \geq \log n$ in order to store the value of t (why?). The following proof places no such restrictions on $s(\cdot)$.

Proof of lemma 44. Without loss of generality, assume that M has only one work tape. This means that if $C \vdash C'$ then $space(C') - space(C)$ is 0 or 1. Let us also fix an input x of M throughout the following proof. For any $h > 0$, let $V = CONFIGS_h(x)$ be the set of configurations of M on input x using space at most h. Let $G = G_h = (V, E)$ be the digraph whose edges comprises $(C, C') \in V^2$ such that $C \vdash_M C'$. Since M is deterministic, each node of G has outdegree at most 1. It is also not hard to see that G has indegree bounded by some constant $O_{\mathrm{M}}(1)$. The graph G has many properties, as illustrated by figure 2.4.

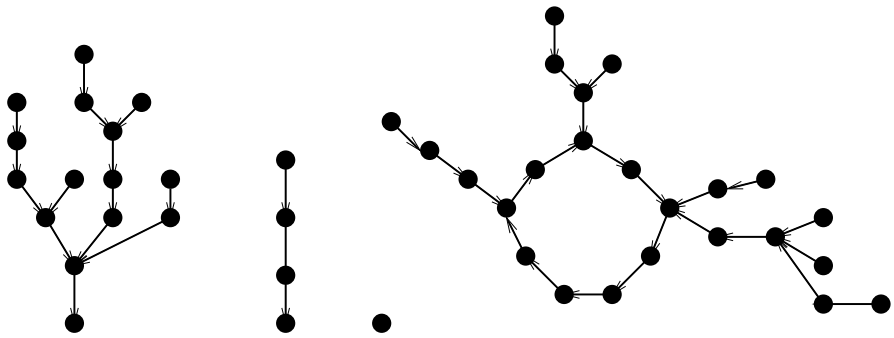


Figure 2.4: Components of a digraph G with outdegree ≤ 1.

Each acyclic component of G will just be a rooted tree. If a component is not a rooted tree, then it has a unique cycle C and has a set of trees, each rooted at some node of C. In this discussion, the edges of a tree are always directed towards the root.

For any configuration $C \in V$, let $T(C)$ denote the subgraph of G induced by the subset of configurations $C' \in V$ that can reach C. Note that by our machine conventions, if $C \vdash C'$ then $space(C) \leq space(C')$; hence the space usage of each configurations in $T(C)$ is at most $space(C)$. If C is a node in a directed cycle of G then $T(C)$ an the entire component of G; otherwise, $T(C)$ is a tree. Unfortunately, to discover if C is in a cycle may take a long time. For our application, an alternative easy-to-check condition is sufficient: we say C is *expectant* if $C \vdash C'$ and $space(C') > space(C)$. The following is immediate:

Fact. If C is expectant or terminal then $T(C)$ is a tree.

We describe a search procedure for $T(C)$ where C is expectant or accepting (hence terminal). This procedure returns "success" if the initial configuration $C_0(x)$

is in $T(C)$, and "failure" otherwise. We will use the standard depth-first search (DFS) order of visiting the nodes. In the usual implementation of DFS, we need to represent the path from the root to the currently visited node, maintained on a (recursion) stack. This information is needed to return from a node to its parent. However, in a DFS on $T(C)$ we can avoid storing this information since the parent of a node C' is simply the successor of C'. This trick allows us to implement the search in $O(h)$ space.

Search(C): // configuration C, return "success" if $T(C)$ is a tree.
    for each child C' of C
        Recursively *search*(C').
        If search of C' is successful, return("success").
    return("failure"). //no recursive calls were successful.

Let us show how to implement this *search* on a Turing machine N using $O(space(C))$ amount of space, assuming that x is available on the input tape. Let N use two work tapes only. Tape 0 stores the input x; tape 1, it stores the current node C that is being visited; tape 2 is used as scratch space. At the start of the recursive call, N checks if C is equal to $C_0(x)$ and if so, returns with "success". Else, if C is a leaf, return with "failure". Otherwise, N will generate the first child C_1 of C and store it on tape 1, and recursively call *search* on C_1. We leave as an exercise to work out the details of this generation. In general, suppose we have just returned from a recursive *search* on some child C_i of C. By assumption, we a copy of C_i on tape 1. It is easy to generate C from C_i, since $C_i \vdash C$. If the *search* of C_i is a success, we return success from C; otherwise, we try to generate the next child C_{i+1} to repeat the recursive search. If C has no more children, we return "failure". This completes the description of *search*.

To conclude the proof of the lemma, we show how to use the *search* procedure. N will compute in *stages*. In *stage* h ($h = 1, 2, \ldots$) N systematically generates all configurations C that use space h. At the start of stage h, we initialize a *found* flag to false. For each expectant or accepting C, N calls the *search*(C). If the search is successful we take one of two actions: if C is accepting, we accept (terminating the entire computation). Otherwise, C is expectant, and we set the *found* flag to true. After we have examined all configurations using space h, we check the *found* flag. This flag is true iff $C_0(x)$ can reach some expectant configuration using space h. In this case N proceeds to stage $h+1$. Otherwise, we terminate the entire computation and reject.

Why is this procedure correct? If this procedure accepts, then $C_0(x)$ reaches some accepting configuration. If this procedure rejects, $C_0(x)$ does not reach any accepting configuration using space $\leq h$, and moreover, it can never reach any configuration using more than space h. Hence we may safely reject. Thus acceptance

or rejection decisions by our procedure are always correct. But we must guard against the possibility of looping. But our requirement for proceeding from stage h to the stage $h+1$ ensures that the original machine M actually runs in at least $h+1$ space if we reach stage $h+1$. Since $s(n) < \infty$, we must get a decision in some finite stage. **Q.E.D.**

As an interesting consequence of the previous lemma, we have:

Theorem 45 *If $n \leq s(n) < \infty$ is an exact space-complexity then $s(n)$ is space-constructible.*

Proof. Let M run in space exactly s. By the previous lemma, we may assume that M halts on all inputs. Now construct another acceptor N as follows: given an input of length n, N ignores the input except for noting its length. Then for each word x of length n, N simulates M on x, marking out the number of cells used. It is clear that N will mark out exactly $s(n)$ cells. The condition that $s(n) \geq n$ is needed in order to cycle through all words of length n. **Q.E.D.**

Note: Hopcroft and Ullman, [21] originally showed that $DSPACE_r(s)$ is closed under complementation under the restriction $s(n) = \Omega(\log n)$s, using the idea of a counter mentioned above. Later Hartmanis and Berman [12] proved that, in case the input alphabet is unary, we can drop the assumption $s(n) = \Omega(\log n)$. The above proof from Sipser shows that we can drop this assumption in general.

Jianer Chen has shown that deterministic reversal classes are also closed under complement:

Theorem 46 *Let* M *accept in $f(n) < \infty$ reversals where f is time-constructible. Then there is a machine* N *that accepts L(M) in $O(f)$ reversals and always halts.*

2.9.2 Complement of Nondeterministic Classes

We prove the Immerman-Szelepcsényi result:

Theorem 47 *(Complement of nondeterministic space classes) If $\log n \leq s(n) < \infty$ for all n then $NSPACE_r(s) = co\text{-}NSPACE_r(s)$.*

For the proof, let us fix M to be any acceptor running in space $s(n) \geq \log n$. We also fix the input word x. For any $h \geq 0, m \geq 0$, let $REACH_h(m)$ denote the set of all configurations of M using space at most h that can be reached from the initial configuration $C_0(x)$ in at most m steps. The *(reachability) census function* $\alpha_h(m)$ (or simply $\alpha_h(m)$) is given by

$$\alpha_h(m) = |REACH_h(m)|.$$

Note that $\alpha_h(0) = 1$ and $\alpha_h(m) \leq \alpha_h(m+1)$. Moreover, if $\alpha_h(m) = \alpha_h(m+1)$ then for all $j > m$, $\alpha_h(m) = \alpha_h(j)$. Our basic goal is to compute this census function.

But first, let us see how such a census function is used. This is seen in a nondeterministic subroutine

$$Check(C, h, m, \alpha_h(m))$$

that "accepts" iff $C \in REACH_h(m)$.

```
Check(C, h, m, α_h(m)):
    begin checking
    c_0 ← 0; // Initialize counter
    For each configuration C' using space at most h,
        begin // iteration
        Nondeterministically guess a path using ≤ h space
        and ≤ m steps, starting from C_0(x);
            If this path reaches C then accept;
            If this path reaches C' then increment c_0 by 1;
        end // iteration
    If c_0 = α_h(m) then reject;
    Else loop;
    end // checking.
```

Note that this procedure terminates in one of three ways: *accept, reject, loop.* These are indicated (respectively) by entering state q_a, state q_r or reaching any terminal configuration that is neither accepting nor rejecting. Assuming $h \geq \log |x|$, it is not hard to see that the space used by the algorithm is

$$h + \log m$$

(the $\log m$ space to count the number of steps in guessed paths). This algorithm is remarkable in three ways. First, we notice that this algorithm does not take all inputs, but only those $\langle C, h, m, p \rangle$ whose last three arguments are constrained by the equation $p = \alpha_h(m)$. Such inputs are called well-formed for this algorithm.[10] Logically speaking, the last argument is redundant because h and m determines $\alpha_h(m)$. However, from a complexity viewpoint, this argument is not redundant because it is not easy to compute $\alpha_h(m)$ from h and m. Second, we make a distinction between rejection (*i.e.*, halting in the reject state q_r) and looping. In general, looping means that the machine either does not halt or enters a terminal state that is neither accepting nor rejecting. Third, our acceptor has the property that on any well-formed input:
(a) at least one path accepts or rejects;
(b) there does not exist two paths, one accepting and the other rejecting.

[10] Cf. assumption (C) in chapter 1 (§4).

In general, we call a nondeterministic acceptor M *unequivocal* if it satisfies (a) and (b) above. We then say that M *unequivocally accepts* its (well-formed) inputs.

Recall in section 2 the definition of univalent (nondeterministic) transducers that compute transformations. We want to use such transducers to compute $\alpha_h(m)$. The input k and output $\alpha_h(m)$ are represented in binary.

Lemma 48 *Let $h \geq \log |x|$. There is a univalent transducer* N *that computes $\alpha_h(m)$ for any $h, m \geq 0$. The running space of* N *is $O(h + \log m)$.*

Proof. Suppose that we have recursively computed $\alpha_h(m-1)$; note that this value can be stored in space $O(h + \log |x|)$. This is computed univalently, meaning that at least one partial computation path that leads to the value $\alpha_h(m-1)$ and enters a special state q_1; furthermore, all paths that lead to state q_1 yield the same value. Assuming that we have reached this special state q_1, we then call the following *Count* subroutine that computes $\alpha_h(m)$ from $\alpha_h(m-1)$:

```
Count(h, m, α_h(m − 1)):
begin counting;
    c_1 ← 0; // initialize counter
    For each configuration C using space at most h,
        begin // outer iteration
        c_2 := 0; // initialize counter
        For each configuration C′ using space at most h,
            begin // inner iteration
                res ← Check(C′, h, m − 1, α_h(m − 1));
                If ((res="accepts") and (C′ = C or C′ ⊢ C)) then c_1 ← c_1 + 1 and break;
            end // inner iteration
        end // outer iteration
    Return(c_1);
end // counting.
```

The counter c_1 is incremented if we found C that can be reached in at most m steps. Note that once counter c_1 is incremented, we break out of the inner loop. Hence, c_1 counts the number of configurations that are reachable in at most m steps. If the nondeterministic subroutine *Check* loops, then we automatically loop.

Let us verify that the procedure for computing $\alpha_h(m)$ is univalent. The key observation is that for every configuration C, the paths (restricted to the inner iteration only) have these properties:

- If $C \in REACH_h(m)$ then there is a non-looping path that increments c_1. Moreover, every path that fails to increment c_1 must loop.

- If $C \notin REACH_h(m)$ then c_1 is never incremented. Moreover, there is a non-looping path.

The space used is $O(h + \log m)$. Our lemma is proved. **Q.E.D.**

We conclude the proof of the main theorem. On input x, we show how to accept x if and only if x is not accepted by M.

```
MAIN PROCEDURE
0.   h ← log |x|. // initialize
1.   // Start of stage h
2.   For m = 1, ..., m_0, compute the α_h(m),
          where m_0 is the first value satisfying α_h(m_0) = α_h(m_0 − 1).
3.   For each accepting configuration C using space ≤ h,
3.1.      Call Check(C, h, m_0, α_h(m_0));
3.2.      If Check accepts, we reject.
3.3.      (N.B. If Check loops, we automatically loop.)
4.   "Either accept or else increment h and go back to step 2."
```

Note that since $h \geq \log |x|$, and $m_0 = |x| O_{\mathrm{M}}(1)^h$, the previous lemma ensures that we use only $O(h)$ space. Since *Check* is an unequivocal procedure, it is clear that the main procedure is also unequivocal. We must now explain step 4. We must decide whether to accept or to repeat the procedure with a larger value of h. Perhaps *Check* did not accept within the for-loop in step 3 because h is not "large enough", *i.e.*, M could accept the input with more space. How do we know whether this is the case? Well, this is the case provided there is a path in which M attempts to use more than h space. To detect this situation, it is easy to put a 'hook' into the *Count* procedure that will set a flag *Continue* to *true* iff there exists a configuration $C \in REACH_h(m)$ where $C \vdash C'$ for some C' that uses $\geq h + 1$ space. (So C is 'expectant' in the sense of the proof of the deterministic result.) So the *Continue* flag is properly set after step 2. Now in step 4 we will accept if *Continue=false*; otherwise we increment h and go back to step 2.

This concludes our proof.

2.10 *The Complexity of Palindromes

We consider the *palindrome language* over $\{0, 1\}$,

$$L_{pal} = \{w : w \in \{0,1\}^* \text{ and } w = w^R\}.$$

The simultaneous time-space complexity of L_{pal} is remarkably well-understood (in view of what little we know about the complexity of other languages). We first prove

[10]* This optional section is independent of the rest of the book.

a lower bound on the simultaneous time-space complexity of L_{pal} using an counting technique based on 'crossing sequences'. The notion of crossing sequences is fairly general. It was originally developed by Hennie [18] for deterministic simple Turing machines, but we shall see that the technique extends to nondeterministic computations. It has recently resurfaced in proving lower bounds on VLSI computations.

Consider a computation path $\bar{C} = (C_t : t \geq 0)$ of a Turing machine M on a fixed input w_0 of length n. Recall (§6, proof of theorem 16) that the *storage configuration* of M (at instant t) refers to the $(2k+1)$-tuple $\langle q, w_1, n_1, \ldots, w_k, n_k \rangle$ that is identical to the configuration C_t except that the input word w_0 and input head position n_0 are omitted. For integer i, the *i-crossing sequence of $\bar{C}$* is $(s_1, \ldots, s_m)$ where $m \geq 0$ is the number of times the input head crosses the boundary between the ith and the $(i+1)$st cells of tape 0, and s_j is the storage configuration of M during the jth crossing. Note that i may be restricted to $0, \ldots, n$ by our standard machine convention. Consecutive storage configurations in a crossing sequence represent crossings in opposite directions. The sum of the lengths of crossing sequences of $\bar{C}$ at positions i, for all $i = 0, \ldots, n$, is at most the time of the computation path. Observe that crossing sequences combine elements of time and of space, so it is not surprising to see that they will be used to obtain a simultaneous lower bound on these two resources. The basic property we exploit is described in the following simple lemma:

Lemma 49 *(fooling lemma) Let* M *be a nondeterministic machine that accepts the words $u = u_1 u_2$ and $v = v_1 v_2$. If there exist accepting computations paths π_u, π_v for u and v such that the crossing sequences of these paths at position $|u_1|$ for input u and at position $|v_1|$ for input v are identical then* M *also accepts $u_1 v_2$ and $v_1 u_2$.*

Proof. By induction on the length of the $|u_1|$-crossing sequence, we can splice together an accepting computation path for $u_1 v_2$ composing of sections that come alternately from π_u and π_v. **Q.E.D.**

The following is a useful inequality:

Lemma 50 *Let $A = (a_{i,j})$ be an $m \times n$ matrix where the $a_{i,j}$ are real numbers. Let $c_j = \frac{1}{m} \sum_{i=1}^{m} a_{i,j}$ be the average value of the numbers on the jth column. Then*

$$\max_{i=1,\ldots,m} \left(\sum_{j=1}^{n} a_{i,j} \right) \geq \sum_{j=1}^{n} c_j.$$

Proof. The left-hand side of the inequality is at least

$$\frac{1}{m} \left(\sum_{j=1}^{n} a_{1,j} \right) + \frac{1}{m} \left(\sum_{j=1}^{n} a_{2,j} \right) + \cdots + \frac{1}{m} \left(\sum_{j=1}^{n} a_{m,j} \right) = \frac{1}{m} \sum_{i=1}^{m} \sum_{j=1}^{n} a_{i,j},$$

which is seen to equal the right-hand side. **Q.E.D.**

Hennie originally showed that a simple Turing machine requires time $\Omega(n^2)$ to L_{pal}. We now generalize this result to nondeterministic multi-tape machines; in this more general setting, we only lower bound the product of time and space.

Theorem 51 *Let t and s be non-decreasing complexity functions. If*

$$L_{pal} \in \textit{N-TIME-SPACE}(t, s)$$

then

$$t(n) \cdot s(n) = \Omega(n^2).$$

Proof. Suppose L_{pal} is accepted by some nondeterministic M in time t and space s. Without loss of generality, consider palindromes of even length; the same proof can be modified for palindromes of odd length. Let S_n denote the set of palindromes of length $2n$. Thus $|S_n| = 2^n$. For each $w \in S_n$, let $\bar{C}_w$ denote some accepting computation path for w. Let R_n be the set consisting of all the $\bar{C}_w$'s, $w \in S_n$. For $i = 0, \ldots, n$, let $R_{n,i}$ denote the multiset[11] of i-crossing sequences of computation paths in R_n. For $u, v \in S_n$, let u_1 and v_1 denote the prefixes of u and v of length i. If $u_1 \neq v_1$ then the fooling lemma implies that the i-crossing sequences of $\bar{C}_u$ and $\bar{C}_v$ are distinct. Since there are 2^i distinct prefixes of length i in S_n, it follows that $|R_{n,i}| \geq 2^i$. Let $r_{n,i}$ denote the average length of crossing sequences in $R_{n,i}$ – since $R_{n,i}$ is a multiset, this is a weighted average which takes the multiplicity of a crossing sequence into account. Applying the inequality in the preceding lemma yields

$$t(2n) \geq \sum_{i=1}^{n} r_{n,i}$$

where the matrix entries $a_{i,j}$ correspond to the length of the jth crossing sequence in the ith computation path. Since at most $|R_{n,i}|/2$ of these crossing sequences have length $\geq 2r_{n,i}$, hence at least $|R_{n,i}|/2 \geq 2^{i-1}$ of these crossing sequences have length $< 2r_{n,i}$. Now there are $\sigma = O_{\mathrm{M}}(1)^{s(2n)}$ storage configurations since M accepts in space $s(2n)$. Let $\tau_{n,i} = \sigma^{2r_{n,i}}$. So there are at most $\tau_{n,i}$ crossing sequences of length $< 2r_{n,i}$. Hence $\tau_{n,i} \geq 2^{i-1}$, or $O_{\mathrm{M}}(1)^{s(2n)2r_{n,i}} \geq 2^{i-1}$. Taking logarithms, $s(2n)r_{n,i} = \Omega_{\mathrm{M}}(i)$. Hence

$$s(2n) \cdot t(2n) \geq s(2n) \sum_{i=1}^{n} r_{n,i} = \sum_{i=1}^{n} \Omega_{\mathrm{M}}(i) = \Omega_{\mathrm{M}}(n^2).$$

Q.E.D.

The preceding lower bound is essentially tight in the sense of the next result. Its proof is left as an exercise.

[11] We say 'multiset' instead of 'set' here to indicate that each i-crossing sequence $\chi \in R_{n,i}$ is associated with a multiplicity. In other words, each χ is tagged with the computation path $\bar{C}_w \in R_n$ from which χ is derived. The number of distinct tags for χ is its multiplicity.

Theorem 52 *Let $s(n) \geq \log n$ be space-constructible. Then*

$$L_{pal} \in D\text{-}TIME\text{-}SPACE(\frac{n^2}{s(n)}, s(n)).$$

We might add that the above proofs go through if we had used *marked palindromes* instead; this is defined as the language over $\{0, 1, \#\}$ consisting of words of the form $x\#x^R$ for all $x \in \{0,1\}^*$. The language of marked palindromes is intrinsically less complex than palindromes.[12] The above results on palindromes motivate an interesting composite-complexity class:

Definition 9 Let f be a complexity function, M a nondeterministic machine. We say M accepts in *time-space product* f if for each $w \in L(\mathrm{M})$, there is an accepting computation path of w that uses time-space (p, q) such that $pq \leq f(|w|)$. We denote the class of such languages by $NTIME \times SPACE(f)$. ■

Thus the time-space product complexity of palindromes is $\Theta(n^2)$. This definition extends to products of other pairs (or tuples) of resources. Observe that for any complexity functions t, s,

$$N\text{-}TIME\text{-}SPACE(t, s) \subseteq NTIME \times SPACE(t \cdot s).$$

In the next section we will consider space-reversal product complexity classes.

2.11 Absolute Lower Bounds

The well-known class of *regular languages* is equal to $NSPACE(0)$, i.e., those languages accepted by acceptors using no space. As noted, machines using no space must be 0-tape Turing acceptors: these machines are also called (nondeterministic) *finite automata*. The properties of regular languages are very well understood. For instance, we know that

$$NSPACE(0) = NSPACE(O(1)) = DSPACE(0).$$

It is generally regarded that space bounds of $o(\log n)$ or time bounds of $o(n)$ are uninteresting for Complexity Theory. In particular, we generally consider

$$D\text{-}TIME\text{-}SPACE(n + 1, \log n)$$

as the smallest interesting class for Complexity Theory. This section shows that, for a certain resource ρ, there are absolute lower limits on the complexity in the following sense: there exists a complexity function g_ρ such that if L is a non-regular

[12] The language of palindromes is context-free but not deterministic context-free. Marked palindromes are deterministic context-free.

language accepted in f units of resource ρ then $f = \Omega(g_\rho)$. Thus g_ρ is a lower bound on the ρ complexity of any non-regular language. We show two results of this kind. Both proofs have the same structure. But before we do that, it is instructive to dispose of the following simple observation:

Lemma 53 *If $t(n) \leq n$ for any n then NTIME$(t(n))$ is regular.*

Proof. Let an acceptor M accept in time $t(n)$ and suppose $t(n_0) \leq n_0$ for some n_0. So for each word x of length n_0, M does not attempt to read the first blank symbol to the right of x. This means that for any word with prefix x, M would behave in exactly the same way. Since x is arbitrary, every input of length greater than n_0 is accepted or rejected according to the behavior of M on its prefix of length n_0. Any such language is seen to be regular. **Q.E.D.**

We now prove a result from Hong [19]. Recall the definition of product complexity classes in the last section.

Theorem 54 *Let f be complexity function such that $f(n) = o(n)$. Then the product class NSPACE $\times$ REVERSAL(f) is precisely the class of regular languages. In this result, reversals made by the input head are not counted.*

We remark that if reversals by input heads are counted (as in our usual definition of reversal complexity) then this theorem would have been a direct consequence of our previous lemma (using the fact that the product of space and reversal is big-Oh of time).

Proof. Suppose L is accepted by a k-tape M in space-reversal product f. To show the theorem, it suffices to prove that L is regular. Let M have as tape alphabet Γ (including the blank $\Box$), input alphabet Σ, and state set Q. Let

$$\Delta = \{\mathit{first}, \mathit{last}\} \times Q \times \Gamma^k$$

where *first* and *last* are two special symbols. Let $w \in \Sigma^*$ with $|w| \geq 2$. In the following, if $d = \mathit{first}$ then $w[d]$ denotes the first symbol of w and if $d = \mathit{last}$ then $w[d]$ denotes the last symbol. We define the binary relation R_w on Δ as follows. Let

$$t = \langle d, q, b_1, \ldots, b_k \rangle, t' = \langle d', q', b'_1, \ldots, b'_k \rangle$$

be tuples in Δ. Then $\langle t, t' \rangle \in R_w$ iff there is a sub-computation path $\bar{C} = (C_1, \ldots, C_m)$ where:

(i) C_1 corresponds to starting M in state q with w as input, with its input head scanning $w[d]$, and the head i scanning b_i (for $i = 1, \ldots, k$).

(ii) The head on each work-tape stays stationary throughout $\bar{C}$. The input head may move. The symbols under the work heads may change.

(iii) In C_{m-1}, the input head is scanning $w[d']$ and in C_m, it is scanning a blank (so it has left the region on the tape containing w). The state and symbols scanned by the work-tape heads in C_m are $q', b'_1, \ldots, b'_k$.

For instance, R_w is empty implies that if we start the machine with w as the input word, in any initial state, with the input head at the first or last symbol of w and with any symbols under the work-tape heads, then one of two things must happen:

(1) The input-head never leaves the area of w, thus violating (iii).

(2) Eventually some work-tape head will make a move *while* the input head is still scanning w, violating (ii).

Note that since $|\Delta| = O_{\mathrm{M}}(1)$, there are $O_{\mathrm{M}}(1)$ distinct relations of the form R_w (over all w). Let α_1 denote the number of such relations R_w. We call R_w the *characteristic relation* of w.

For each $h \geq 0$, let $L^{[h]}$ denote those words w in L for which the minimal space-reversal product of any accepting computation of M for w is h. (Reversals by input head not counted.) Hence the sets $L^{[h]}$ form a partition of L. If $L^{[h]}$ is empty for all but finitely many values of h, then it is easy to show that L is regular (we leave this as an exercise). So for the sake of contradiction, let us assume $L^{[h]}$ is non-empty for infinitely many h. If $L^{[h]}$ is non-empty, then let $\mu(h)$ denote the length of the shortest word in $L^{[h]}$; else, we define $\mu(h) = 0$.

We claim that $\mu(h) < (\alpha_1 + 2)(h+1)$, where α_1 is the constant described earlier. To see this, suppose otherwise. Then for some $w \in L^{[h]}$, we have $\mu(h) = |w| \geq (\alpha_1 + 2)(h+1)$. Let $\bar{C}$ be the accepting computation path on input w that has space-reversal product of h. Let an *active* step of $\bar{C}$ refer to one in which at least one of the work heads actually moves (the input head's movement does not count). If $\bar{C}$ makes (r, s) reversal-space then the number of active steps of $\bar{C}$ is at most $rs \leq h$. Divide w into $h+1$ subwords each of length at least $\alpha_1 + 2$: a simple pigeon-hole argument shows that for some such subword u, all the work heads of M are stationary whenever the input head is scanning u. More precisely, every step of the form $C \vdash C'$ where the input head in C is scanning a symbol in u is an inactive step. Among the $\alpha_1 + 1$ of prefixes of u of length ≥ 2, there must be two prefixes v and v' with the same characteristic relation, $R_v = R'_v$. If v' is the shorter of the two prefixes, replace v with v' in w to obtain a shorter word w'. It follows from the definition of characteristic relations that w' is also accepted by M, with the same space-reversal product complexity as w. (This is essentially the fooling lemma in the previous section.) This contradicts our choice of w as the shortest word in $L^{[h]}$.

We conclude that for any shortest length word w in $L^{[h]}$, $|w| < (\alpha_1 + 2)(h+1)$ or $h = \Omega_{\mathrm{M}}(|w|)$. Since $f(|w|) \geq h$, and there are infinitely many such w's, we conclude $f(n) \neq o(n)$. This contradicts our assumption on f. **Q.E.D.**

We next prove a similar result of Hopcroft and Ullman [21]. To do this, we extend the notion of characteristic relations to allow the work-tape heads to move.

With M, Q, Σ and Γ as above, we now define for any integer $h \geq 1$:

$$\Delta_h = \{\mathit{first}, \mathit{last}\} \times Q \times (\Gamma^h)^k \times \{1, \ldots, h\}^k.$$

(The previous definition of Δ may by identified with the case $h = 1$.) Consider a tuple

$$t = \langle d, q, w_1, \ldots, w_k, n_1, \ldots, n_k \rangle \in \Delta_h.$$

Each word w_i is of length h and, roughly speaking, denotes the contents of the ith tape of M and n_i indicates the position of head i in w_i. For any $w \in \Sigma^*$, $|w| \geq 2$, we again define a binary relation R_w^h over Δ_h as follows. Let $t \in \Delta_h$ be as given above and $t' \in \Delta_h$ be the primed version. A pair $\langle t, t' \rangle$ is in R_w^h iff there is a sub-computation path $\bar{C} = (C_1, \ldots, C_m)$ such that

(i) C_1 corresponds to starting M with w as input string, input head scanning $w[d]$, and for $i = 1, \ldots, k$: the non-blank part of tape i is w_i with head i scanning $w_i[n_i]$.

(ii) The work-tape heads stays within the w_i's throughout the computation. The input head similarly stays within w except during the last configuration C_m.

(iii) In C_{m-1}, the input head is scanning $w[d']$ and in C_m, the input head has left the region of w. The state, contents of the work-tapes and the corresponding head positions in C_m are given by the rest of the tuple t'.

Note that

$$|\Delta_h| = h^k O_{\mathrm{M}}(1)^h = O_{\mathrm{M}}(1)^h.$$

For any set X with n elements, the number of binary relations over X is 2^{n^2}. Hence if α_h denotes the number of distinct relations R_w^h, over all possible words w, then there exists some constant $C = O_{\mathrm{M}}(1)$ that does not depend on h such that

$$\alpha_h \leq 2^{C^h}$$

We can now prove the result of Hopcroft and Ullman:

Theorem 55 *Let s be a complexity function such that $s(n) = o(\log \log n)$. Then DSPACE(s) is the set of regular languages.*

Proof. Let L be accepted by some M in space $s(n)$. The proof proceeds quite similarly to the previous one. For each $h \geq 0$, let $L^{[h]}$ be the set of words in L accepted in space h but not in space $h-1$. If $L^{[h]}$ is empty for all but finitely many values of h then L is regular. So for the sake of contradiction, assume otherwise. Define $\mu(h)$ to be the length of the shortest word in $L^{[h]}$ when $L^{[h]}$ is non-empty; otherwise $\mu(h) = 0$. If w is a word of shortest length in a non-empty $L^{[h]}$ then we claim

$$|w| = \mu(h) \leq 2^{C^{s(|w|)}}. \tag{2.7}$$

Otherwise, there exists a pair of distinct prefixes u and v of w that have identical characteristic relations, $R_u^h = R_v^h$. Assuming that u is the longer of the two prefixes, we can decompose w into $w = vv'w' = uw'$ where $v' \neq \epsilon$. It is easy to verify that vw' is also accepted by M in the same space h (again, the fooling lemma argument). This contradicts our choice of w as the shortest word. From (2.7) and the fact that there are infinitely many such words w, we conclude that $s(n) \neq o(\log\log n)$. This contradicts our assumption on $s(n)$. **Q.E.D.**

We can conceive of other methods of defining languages and complexity that admit non-regular languages with space complexity bounded by functions growing slower than $\log\log n$. A simple way to do this would be to assume that the Turing machine comes equipped with a complexity function h such that on input of length n, $h(n)$ cells are automatically marked out before the computation begins, and that the Turing machine never exceeds the marked cells. Hence the machine uses space $h(n)$, which of course can be chosen arbitrarily slowly growing. Our preceding proofs would no longer be valid; indeed, it is easy to see that even non-recursively enumerable languages can be accepted this way (how?). We refer to [30] for languages with such low space complexity.

To end this section, we note that the preceding absolute lower bounds are essentially tight in the following sense:
(i) There are non-regular languages whose space-reversal product is $O(n)$. This is illustrated by the palindrome language: it is non-regular and it can be accepted in linear space and $O(1)$ reversals.
(ii) There non-regular languages in the class $DSPACE(\log\log n)$. In fact, the following language is an example.

$$L_0 = \{\bar{1}\#\bar{2}\#\cdots\#\overline{n-1}\#\bar{n} : n \text{ is a natural number}\}$$

where $\bar{m}$ denotes the binary representation of the integer m. There is another candidate language due to Mehlhorn and Alt[29]. It is rather interesting because it is over a single letter alphabet. First let $q(n)$ denote the smallest number that does not divide an integer n.

$$L_1 = \{1^n : n \text{ is a natural number and } q(n) \text{ is a power of two }\}$$

where 1^n is the unadic representation of n.

We leave it as an exercise to show that both these languages have the required properties.

2.12 Final remarks

This chapter introduces the basic model of computation that is to be the basis for comparing all future models of computation. This choice is not essential to the theory because of the computational theses in chapter 1. The results in this chapter

are of two types: (I) technical ones about Turing machines in particular and (II) relationships about complexity classes. Although we are mainly interested type-(II) results, some of the model-specific results are unavoidable. This would be true regardless of our choice of model of computation. Our choice of the Turing model is particularly fortunate in this regard because the literature contains a wealth of these model-specific results. We have restricted ourselves to those type (I) results that are needed later.

The three simulation sections use the very distinct techniques for each resource. They illustrate the fundamentally different properties of these resources. A common aphorism expressing this difference between time and space is the following: *time is irreversible but space is reusable.*[13] Both the Savitch and Istvan Simon techniques exploit the reusability of space. But what of reversals? Within a phase of a computation, all the individual actions are independent of one another. This is exactly the nature of time in parallel computation. This prompted Hong Jia-wei to identify reversals with parallel time. This identification is essentially correct except for low level complexity; we will return to the precise relationships in later chapters. Since time and space without reversal is useless (why?), we offer this complement to the above aphorism: *reversal is the ingredient to convert reusable space into time.*

The interplay of the triumvirate is best illustrated by deterministic computation: if $t_{\mathrm{M}}(n), s_{\mathrm{M}}(n)$ and $r_{\mathrm{M}}(n)$ are the exact running time, space and reversal of a halting deterministic Turing machine M, then there are constants $c_1, c_2, c_3 > 0$ that make the following fundamental inequalities true:

$$s_{\mathrm{M}} + r_{\mathrm{M}} \leq c_1 t_{\mathrm{M}} \leq c_2(n + s_{\mathrm{M}})r_{\mathrm{M}} \leq c_3 t_{\mathrm{M}}^2.$$

A basic challenge of complexity theory is to refine these inequalities and to extend them to other computational modes.

[13] Algebraically, this means that we must sum (respectively, take the maximum of) the time (respectively, space) used by several independent computations.

Exercises

[2.1] Construct a simple Turing machine that on input $x \in \{0,1\}^*$ converts x to $x-1$ where we regard x as a binary integer. Hence the machine decrements its input. For instance if $x = 0110100$ then the machine halts with 0110011. Write out the transition function δ of your machine explicitly.

[2.2] (i) Give a nondeterministic Turing acceptor that accepts the complement of the palindrome language L_{pal} using acceptance space of $\log n$. The exercise in last question is useful here. (Note that the acceptor in example 2 uses linear space).
(ii) Show (or ensure) that your machine accepts in linear time. (It is probably obvious that it accepts in $O(n \log n)$ time.)

[2.3] Give an algorithm for obtaining $\delta(N)$ from $\delta(\mathrm{M})$ having the nondeterministic speedup properties described in theorem 2. What is the complexity of your algorithm?

[2.4] Consider the recognition problem corresponding to multiplying binary numbers: MUL is the language comprising those triples of the form $\langle a, b, c\rangle$ where a, b, c are binary numbers and $ab = c$. Show that MUL is in $DLOG$.

[2.5] Define 'rejection complexity' for deterministic machines such that the acceptance and rejection time complexity of L are both $f(\cdot)$ iff the running time complexity of L is also f. Can this be extended to nondeterministic machines? Remark: If acceptance complexity corresponds to 'proofs' in formal proof systems then rejection complexity corresponds to 'disproofs' of (or, counter-examples to) non-theorems.

[2.6] (i) Show that if $f(n) = \omega(n)$ and f is time-constructible, then $XTIME_r(f) = XTIME(f)$, $X = D, N$.
(ii) Show a similar result for space-constructible f, but no longer assuming $f(n) = \omega(n)$.

[2.7] * Show that the following functions are time-constructible for any positive integer k:
(i) $n \log^k n$, (ii) n^k, (iii) $n!$ (factorial), (iv) k^n. (See [23] for a systematic treatment of such proofs.)

[2.8] Redo the last problem for reversal-constructible. (Recall that reversal-constructible is defined slightly differently than in the case of time or space.)

[2.9] Suppose $f(n) \geq n$ and $g(n)$ is an integer for all n, and both f and g are time-constructible.
(i) Show that the functional composition $f \circ g$ is time-constructible. (*Note:*

$f \circ g(n) = f(g(n))$. It is easy to see that $f(n)+g(f(n))$ is time-constructible, but we want $f(g(n))$.)
(ii) If $f(n)$ is also an integer for all n, show that $f \cdot g$ (product) is time-constructible.

[2.10] * (i) Show that $\log n$ is space-constructible.
(ii) Define $\log^* n$ to be the largest integer m such that $\exp(m) \leq n$ where $\exp(0) = 0$ and $\exp(m+1) = 2^{\exp(m)}$. Show that $n \log^* n$ is space-constructible.
(iii) (Ladner, Freedman) Show that there exists a space-constructible f such that $f(n) = O(\log \log n)$ and, for some $C > 0$, $f(n) \geq C \log \log n$ infinitely often. *Hint:* $f(n)$ is the largest prime p such that every prime $q \leq p$ divides n. Some basic knowledge of number theory (mainly the prime number theorem) is required for this problem.
(iv) (Seiferas) Show that $\log \log n$ is not space-constructible. *Hint:* prove that any space-constructible function not in $\Omega(\log n)$ must be $O(1)$ infinitely often.

[2.11] For most applications, a weaker notion of constructibility suffices: A function f is *approximately time-constructible* if there is a time-constructible function f' such that $f = \Theta(f')$. Redo the above exercises but using the "approximate" version of time-constructibility instead. It should be much easier.

[2.12] Show that the two languages at the end of section 9 are non-regular and can be accepted in $O(\log \log n)$ space. For non-regularity, the reader should use the 'pumping lemma for regular languages'.

[2.13] Show that if a function is time-constructible, then it is space-constructible.

[2.14] Prove that for any constant k, $NTIME(k)$ is a proper subclass of $NSPACE(0)$.

[2.15] There are three possible responses to the following statements: True, False or Perhaps. 'Perhaps' reflects the uncertainty from what we know (assume only the results in this chapter). If your answer is True or False, you must give brief reasons in order to obtain full credit.
(a) T/F/P: $NSPACE(n) \subseteq DTIME(O(1)^n)$.
(b) T/F/P: $NTIME(n) \subseteq DTIME(2^n)$.
(c) T/F/P: $NREVERSAL(n) \subseteq DREVERSAL(O(1)^n)$.

[2.16] (Hartmanis) For every $t(n) \geq n \log n$, if M is a simple Turing machine accepting in time t then M accepts in space $O(t(n)/\log t(n))$. *Note:* This result has been improved in two ways: First, this result has been shown for multitape Turing machine by Hopcroft, Valiant and Paul (but Adleman and

Loui provided a very different proof). For simple Turing machines, Paterson shows that if a language is accepted by a simple Turing machine in time t then it can be accepted in space $t^{1/2}$. See chapter 7 for these results and references.

[2.17] (i) Construct a Turing machine that computes the Boolean matrix $\mu_{x,h}$ and then the transitive closure $\mu^*_{x,h}$ (say, using the Floyd-Warshall algorithm) in time $O(m^3)$ where the matrix μ_c is $m \times m$. (We already know that transitive closure can be computed in $O(m^3)$ time on a random access machine – the question is whether a Turing machine can achieve the same time bound. The organization of the Boolean matrix on the tapes is crucial.)
(ii) Give an alternative proof of theorem 16 using the previous result on computing transitive closure.

[2.18] Modify the proof of theorem 18 for the corresponding result for running complexity.

[2.19] * Show that $DTIME(n+1) \neq DTIME(O(n))$.

[2.20] (Hopcroft-Greibach) If $L \in NTIME(n+1)$ then L has the form $h(L_1 \cap \cdots \cap L_k)$ where the L_i's are context-free languages and h is a letter homomorphism.

[2.21] This exercise gives a characterization of $NTIME(t)$ for any complexity function t. Let $h : \Sigma \to \Gamma^*$ be a homomorphism and t a complexity function. We say h is *t-bounded on L* if for all n large enough, for all $x \in h(L)$ with $|x| \geq n$, there exists $w \in L$ such that $h(w) = x$ and $|w| \leq t(|x|)$. [14] For any complexity class K, define $H_t[K]$ to be

$$\{h(L) : L \in K \wedge h \text{ is } t\text{-bounded on } L\}.$$

Prove that for all t, $NTIME(t)$=$H_t[NTIME(n+1)]$. Conclude from this that $NTIME(n+1)$ is closed under non-erasing homomorphism. (Note: h is *non-erasing* if $h(b) \neq \epsilon$ for all $b \in \Sigma$.)

[2.22] Prove theorem 52.

[2.23] (Hennie) Show that every simple Turing acceptor for the palindrome language L_{pal} takes time $\Omega(n^2)$.

[2.24] (Ó'Dúnlaing) For $k \geq 1$, let $(\{0,1\}^*, L_k)$ be the language consisting of binary strings of period k: say a word w has period k if $|w| \geq k$ and for all $i = 1, 2, \ldots, |w| - k - 1$, $w[i] = w[i+k+1]$.

[14] Our definition here differs from the the literature because of our use of acceptance time rather than running time. The usual definition simply says: $\exists c > 0$, for all $w \in L$, $|w| \leq ct(|x|)$.

(i) Show that the complement of L_k can be accepted by a 'small' nondeterministic finite automaton, say with $2k+2$ states.
(ii) Prove that L_k cannot be accepted by any nondeterministic finite automaton with less than 2^k states.

[2.25] A worktape of a Turing machine is said to be a *pushdown store* if it is initialized with one special symbol (called *bottom symbol*) which is never erased, and tape head is constrained so that (1) if it moves left, it must write a blank on the current cell (this is called a *pop move*, and (2) the blank symbol is never written under any other circumstances. When the head moves moves right, we call it *push move*. These requirements imply that the tape head is always scanning the rightmost non-blank cell on the tape (called the *top symbol*) or (temporarily for one moment only) the blank symbol on the right of the top symbol. An *auxiliary pushdown automata* (apda) is a nondeterministic 2-tape Turing acceptor in which tape 1 is a pushdown store and tape 2 is an ordinary worktape. For apda's, we only count the space usage on tape 2; that is, space on the pushdown store is not counted. Show the following to be equivalent:
(a) L is acceptd by a deterministic apda using space $s(n)$.
(b) L is acceptd by a nondeterministic apda using space $s(n)$.
(c) $L \in DTIME(O(1)^{s(n)})$.

[2.26] (Fischer, Meyer, Rosenberg) A *counter machine* (CM) is a multitape Turing machine except that instead of tapes, the CM has 'counters'. A counter contains a non-negative integer which the CM can increment, decrement or test to see if it is equal to zero. More precisely, if the CM has k counters, we may represent its transition table by a set of tuples (instructions) of the form

$$\langle q, a, z_1, \ldots, z_k, q', d_0, d_1, \ldots, d_k \rangle$$

where q, q' are states, a is an input symbol, $z_1, \ldots, z_k \in 0, 1$ and $d_0, d_1, \ldots, d_k \in \{-1, 0, +1\}$. The above instruction is *applicable* if the current state is q, the input head is scanning the symbol a, and the ith counter ($i = 1, \ldots, k$) contains a zero iff $z_i = 0$. To *apply* the instruction, make the next state q', move the input head in the direction indicated by d_0, and increment or decrement the i^th counter by d_i. The *space* used by the counter on an input w is the largest integer attained by any counter during the computation. Show that for any CM accepting in space f, there is another CM accepting the same language but using space only $f^{1/2}$. (Thus counter machines exhibit a 'polynomial space speedup'.) *Hint:* show how to replace one counter by two counters where the new counters never contain a number larger than the square-root of the largest number in the original counter.

[2.27] (P. Fischer) Show a linear speedup result for simple Turing machines. *Hint:*

the Linear Speedup theorem as stated fails but what stronger assumption about $t(\cdot)$ must be made?

[2.28] Show how to reduce the retrieval problem (section 2.8.3) to the sorting problem, thereby giving another $O(\log m)$ deterministic reversal solution. [Recall the retrieval problem is where we are given, say, on tape 1 an integer $h \geq 1$ in unary, on tape 2 a word $c \in \{0,1\}^*$, and on tape 3 the table

$$c_1\$d_1\#c_2\$d_2\#\cdots c_m\$d_m\$, \qquad (c_i, d_i \in \{0,1\}^*)$$

where $m = 2^h$, and the c_i's are assumed distinct. We want to output d_i on tape 4 where $c = c_i$ for some $i = 1, \ldots, m$. You may assume that c *does* occur among $c_1, \ldots, c_m$.

[2.29] (Chrobak and Rytter) Show that if we do not count reversals made by the input head then $DLOG$ can be simulated in constant reversals.

[2.30] Give a proof that the class $DSPACE(s(n))$ is closed under complementation if $s(n) \geq \log n$, using the idea of counters described in section 8.

[2.31] Show that the census technique can also be used to show that the class $NSPACE_e(s(n))$ of languages accepted by *unequivocal* acceptors in space $s(n) \geq \log n$ is closed under complementation. Recall that on any input, the computation tree of an unequivocal acceptor has at least one accepting path or one rejecting path (rejection is by entering the special state q_r only). Moreover, the computation tree cannot have both accepting and a rejecting path.

[2.32] (Fischer, Meyer, Rosenberg) Suppose we allow each work-tape to have more than one reading head, but the number is fixed for each tape, depending on the particular machine. Furthermore, when two or more tape heads are scanning the same cell, this information is known to the machine. Formalize this *multihead multitape model* of Turing machines. Show that any such machine can be simulated by ordinary multitape machines without time loss.

[2.33] Let t, s, r be the running time, space and reversal of a halting Turing machine M. Does the fundamental inequalities $s + r = O_{\mathrm{M}}(t) = O_{\mathrm{M}}((s+n)r) = O_{\mathrm{M}}(t^2)$ (see the concluding section) hold for nondeterministic M?

[2.34] * Define the static complexity of a problem L to be the smallest sized Turing machine that will accept L. 'Machine size' here can be variously interpretated, but should surely be a function of the tape alphabet size, the number of tapes and the number of states. Does every problem L have a unique static complexity? How sensitive is this complexity to change in Turing machine conventions?

[2.35] * Is there some form of linear speedup theorem for reversal complexity?

[2.36] * Can the tape reduction theorem for reversal complexity be improved?

[2.37] * In the inclusion $D\text{-}SPACE\text{-}REVERSAL(s,r) \subseteq DSPACE(r \log s)$, can we replace the left-hand side by $N\text{-}SPACE\text{-}REVERSAL(s,r)$ possibly at the expense of a large space bound on the right-hand side? E.g., $N\text{-}SPACE\text{-}REVERSAL(s,r) \subseteq DSPACE(rs/\log s)$.

[2.38] * Prove or disprove: let $t(n) = \Omega(n^2)$, $r(n) = \Omega(\log t(n))$, $s(n) = \Omega(\log t(n))$, $r(n) \cdot s(n) = O(t(n))$. Then $DTIME(t) - D\text{-}SPACE\text{-}REVERSAL(s,r) \neq \emptyset$.

Bibliography

[1] S. O. Aanderaa. On k-tape versus $(k-1)$-tape real time computation. In R. M. Karp, editor, *Complexity of computation*, pages 74–96. Amer. Math. Soc., Providence, Rhode Island, 1974.

[2] B. S. Baker and R. V. Book. Reversal-bounded multipushdown machines. *Journal of Computers and Systems Science*, 8:315–322, 1974.

[3] R. V. Book and S. A. Greibach. Quasi-realtime languages. *Math. Systems Theory*, 4:97–111, 1970.

[4] R. V. Book, S. A. Greibach, and B. Wegbreit. Time and tape bounded Turing acceptors and AFL's. *Journal of Computers and Systems Science*, 4:606–621, 1970.

[5] R.V. Book and Chee Yap. On the computational power of reversal-bounded machines. *ICALP '77*, 52:111–119, 1977. Lecture Notes in Computer Science, Springer-Verlag.

[6] S.R. Buss, S.A. Cook, P.W. Dymond, and L. Hay. The log space oracle hierarchy collapses. Technical Report CS103, Department of Comp. Sci. and Engin., University of California, San Diego, 1987.

[7] Ee-Chien Chang and Chee K. Yap. Improved deterministic time simulation of nondeterministic space for small space: a note. *Information Processing Letters*, 55:155–157, 1995.

[8] Martin Davis. Why Gödel didn't have Church's Thesis. *Information and Computation*, 54(1/2):3–24, 1982.

[9] Jian er Chen. *Tape reversal and parallel computation time*. PhD thesis, Courant Institute, New York University, 1987.

[10] Jian er Chen and Chee-Keng Yap. Reversal complexity. *SIAM J. Computing*, to appear, 1991.

[11] Patrick C. Fisher. The reduction of tape reversal for off-line one-tape Turing machines. *Journal of Computers and Systems Science*, 2:136–147, 1968.

[12] J. Hartmanis and L. Berman. A note on tape bounds for sla language processing. *16th Proc. IEEE Symp. Found. Comput. Sci.*, pages 65–70, 1975.

[13] J. Hartmanis, P. M. Lewis II, and R. E. Stearns. Hierarchies of memory limited computations. *IEEE Conf. Record on Switching Circuit Theory and Logical Design*, pages 179–190, 1965.

[14] J. Hartmanis and R. E. Stearns. On the computational complexity of algorithms. *Trans. Amer. Math. Soc.*, 117:285–306, 1965.

[15] Juris Hartmanis. Computational complexity of one-tape Turing machine computations. *Journal of the ACM*, 15:325–339, 1968.

[16] Juris Hartmanis. Tape-reversal bounded Turing machine computations. *Journal of Computers and Systems Science*, 2:117–135, 1968.

[17] F. C. Hennie and R. E. Stearns. Two-tape simulation of multitape Turing machines. *Journal of the ACM*, 13(4):533–546, 1966.

[18] Frederick C. Hennie. One-tape, off-line Turing machine computations. *Information and Computation*, 8(6):553–578, 1965.

[19] Jia-wei Hong. A tradeoff theorem for space and reversal. *Theoretical Computer Science*, 32:221–224, 1984.

[20] Jia-wei Hong. *Computation: Computability, Similarity and Duality.* Research notices in theoretical Computer Science. Pitman Publishing Ltd., London, 1986. (available from John Wiley & Sons, New York).

[21] J. Hopcroft and J. Ullman. Some results on tape bounded Turing machines. *Journal of the ACM*, 16:168–188, 1969.

[22] Neil Immerman. Nondeterministic space is closed under complement. *Structure in Complexity Theory*, 3:112–115, 1988.

[23] Kojiro Kobayashi. On proving time constructibility of functions. *Theoretical Computer Science*, 35:215–225, 1985.

[24] K.-J. Lange, B. Jenner, and B. Kirsig. The logarithmic alternation hierarchy collapses: $A\Sigma_2^L = A\Pi_2^L$. *Proc. Automata, Languages and Programming*, 14:531–541, 1987.

[25] Ming Li. On one tape versus two stacks. Technical Report Tech. Report TR84-591, Computer Science Dept., Cornell Univ., Jan. 1984.

[26] Maciej Liśkiewicz. On the relationship between deterministic time and deterministic reversal. *Information Processing Letters*, 45:143–146, 1993.

[27] Wolfgang Maass. Quadratic lower bounds for deterministic and nondeterministic one-tape Turing machines. *16th Proc. ACM Symp. Theory of Comp. Sci.*, pages 401–408, 1984.

[28] Stephen R. Mahaney. Sparse complete sets for *NP*: solution to a conjecture of Berman and Hartmanis. *Journal of Computers and Systems Science*, 25:130–143, 1982.

[29] H. Altand K. Mehlhorn. A language over a one symbol alphabet requiring only $O(\log \log n)$ space. *SIGACT news*, 7(4):31–33, Nov, 1975.

[30] Burkhard Monien and Ivan Hal Sudborough. On eliminating nondeterminism from Turing machines which use less than logarithm worktape space. In *Lecture Notes in Computer Science*, volume 71, pages 431–445, Berlin, 1979. Springer-Verlag. Proc. Symposium on Automata, Languages and Programming.

[31] W. J. Paul, J. I. Seiferas, and J. Simon. An information-theoretic approach to time bounds for on-line computation. *Journal of Computers and Systems Science*, 23:108–126, 1981.

[32] Michael O. Rabin. Real time computation. *Israel J. of Math.*, 1(4):203–211, 1963.

[33] W. Rytter and M. Chrobak. A characterization of reversal-bounded multipushdown machine languages. *Theoretical Computer Science*, 36:341–344, 1985.

[34] W. J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computers and Systems Science*, 4:177–192, 1970.

[35] Istvan Simon. On some subrecursive reducibilities. Technical Report Tech. Rep. STAN-CS-77-608, Computer Sci. Dept., Stanford Univ., April, 1977. (PhD Thesis).

[36] Michael Sipser. Halting space-bounded computations. *Theoretical Computer Science*, 10:335–338, 1980.

[37] Róbert Szelepcsényi. The method of forcing for nondeterministic automata. *Bull. European Association Theor. Comp. Sci.*, pages 96–100, 1987.

[38] Seinosuke Toda. $\Sigma_2 SPACE(n)$ is closed under complement. *Journal of Computers and Systems Science*, 35:145–152, 1987.

Contents

Chapter 3

Introduction to the Class NP

February 10, 1999

3.1 Introduction

According to the principles outlined in chapter 1, the class $P = DTIME(n^{O(1)})$ (resp. $NP = NTIME(n^{O(1)})$) corresponds to those problems that are time-tractable when we compute in the fundamental (resp. nondeterministic) mode. Clearly $P \subseteq NP$; the P versus NP (or '$P \neq NP$') question asks if this inclusion is proper. Since most real world computers operate in the fundamental mode, P is clearly very important. The practical significance of NP is not immediately evident. In 1971, Cook [4] proved a theorem connecting P and NP that was destined to play a decisive role in Complexity Theory. Cook showed that the well-known problem of recognizing satisfiable Boolean formulas[1] is the "hardest" problem in NP in the sense that if this problem is in P then $P = NP$. Shortly afterwards, Karp [7] showed that a large list of problems that are of practical importance in the field of operations research and combinatorial optimization are also hardest in this sense. Independently, Levin [11] proved a theorem similar to Cook's result. The list of hardest problems has since grown to include hundreds of problems arising in practically every area of the computational literature: see Garey and Johnson [5] for a comprehensive catalog up till 1979. The list highlights the fact that NP is empirically (not just theoretically) a very important class. The added significance of the Cook-Karp discoveries is that these problems in NP have defied persistent attempts by researchers to show that they are in P. Although we are probably a long way from settling the P versus NP question, most researchers believe that P is not equal to NP. Because of this belief, a proof that a certain problem is hardest for NP is taken as strong evidence of the

[1] Actually, Cook stated his theorem in terms of recognizing the set of tautologous Boolean formulas.

fundamental intractability[2] of the problem. For designers of efficient algorithms, this is invaluable information: it warns that unless one has new and deep insights to these problems that escaped previous researchers, one should not expect to find polynomial time solutions.

We introduce a well-known problem that is not known to be fundamentally tractable:

TRAVELING SALESMAN OPTIMIZATION PROBLEM (TSO)	
GIVEN:	An $n \times n$ matrix $D = (d_{i,j})$ whose entries are either non-negative integers or ∞, and $d_{i,i} = 0$ for all i.
FIND:	A cyclic permutation $\pi = (\pi(1), \pi(2), \ldots, \pi(n))$ of the integers $1, 2, \ldots, n$ such that $D(\pi)$ is minimized: $D(\pi) := d_{\pi(1),\pi(2)} + d_{\pi(2),\pi(3)} + \cdots + d_{\pi(n-1),\pi(n)} + d_{\pi(n),\pi(1)}$.

The TSO represents the problem of a salesman who wants to visit each of n cities exactly once and then return to his original city. The entry $d_{i,j}$ of the matrix D represents the distance from city i to city j. Each cyclic permutation π corresponds to a *tour* of the n cities where the traveling salesman visits the cities $\pi(1), \pi(2), \cdots, \pi(n)$, and $\pi(1)$ in turn: here it does not matter which is the first city and, without loss of generality, we always let $\pi(1) = 1$. $D(\pi)$ is called the *length* of the tour.

The brute force method of solving this problem involves generating all $(n-1)!$ cyclic permutations and choosing the permutation with the minimal tour length. Thus the method takes $\Omega((n-1)!)$ time. There is a dynamic programming method due to Karp and Held that improves on this, giving $O(n2^n)$ time (see Exercises).

The reader may have observed that the TSO is not a recognition problem which is what our theory normally treats. In the next section we illustrate the empirical fact that for most optimization or functional problems Q that are not known to be fundamentally tractable, we can easily derive a corresponding recognition problem Q' with the property that Q can be solved in polynomial time iff Q' is in the class P. We say that Q and Q' are *polynomially equivalent* in this case. Note that we have not defined what it means for a functional problem like TSO to be solved in polynomial time (in the fundamental mode): this can easily be done and we leave it to the reader.

3.2 Equivalence of Functional and Recognition problems

In this section, we consider three important functional problems, and in each case give a related recognition problem that is polynomially equivalent to the functional

[2]We use the term 'fundamental intractability' for any problem (not necessarily a recognition problem) that cannot be solved in polynomial time when computing in the fundamental mode.

problem. This will give evidence for our claim in chapter one that recognition problems are adequate for the study of tractability.

We have just described the traveling salesman optimization problem. The corresponding recognition problem is

TRAVELING SALESMAN DECISION PROBLEM (TSD) GIVEN: A matrix D as in TSO and an integer b. PROPERTY: There exists a tour π such that $D(\pi) \leq b$.

This example illustrates the typical way we describe recognition problems in this book: the well-formed inputs for the problem are described under the heading GIVEN, where some fixed but reasonable encoding (as discussed in section 5, chapter 1) of these inputs is implicitly assumed. We then identify the recognition problem with the language whose members encode those well-formed inputs satisfying the predicate given under the heading PROPERTY. We want to show the following

Proposition A. *The TSO problem is fundamentally tractable iff the TSD is in P.*

Suppose that the TSO problem is fundamentally tractable. To solve the decision problem (on input $\langle D, b\rangle$), first find the optimal tour π_{opt}. Compute $D(\pi_{opt})$ and accept iff $D(\pi_{opt}) \leq b$. Under reasonable assumptions, and using the fact that π_{opt} can be found in polynomial time, the described procedure for the TSD clearly takes polynomial time.

Conversely, suppose that TSD is fundamentally tractable. Let R be an algorithm that solves the TSD problem in polynomial time in the fundamental mode. First, we find $b_{\min}$, the minimal tour length using a binary search of the range $[0, m]$ where m is the sum of all the finite entries of D: each probe of the binary search involves a call to the algorithm R as subroutine. The number of probes is $O(\log m)$ which is linear in the size of the input. Next we construct the minimal tour as follows. We regard D as representing a directed graph $G(D)$ on n vertices where an edge $\{i, j\}$ is present iff $d_{i,j} < \infty$. Now we successively 'test' each edge $\{i, j\}$ (in any order) to see if the removal of that edge would result in an increase in the minimal tour length. More precisely, we replace $d_{i,j}$ with ∞ and call the subroutine R to see if the minimal tour length of the modified matrix D exceeds $b_{\min}$. If it does not increase the minimal tour length, we will permanently remove edge $\{i, j\}$; otherwise we conclude that the edge $\{i, j\}$ is necessary in any minimal tour involving the remaining edges of $G(D)$. Proceeding in this fashion, we will terminate after at most $n^2 - 2n$ tests. Note that exactly n edges will remain in the final graph, and this gives the required tour. The entire procedure is seen to require only a polynomial number of calls to R. This completes the proof of the proposition.

The transformation of the TSO problem into TSD is typical. We now illustrate a similar transformation on a graph coloring problem. Let $G = (V, E)$ be an (undirected) graph. For any positive integer k, a function $C : V \to \{1, \ldots, k\}$ is

called a *k-coloring* of G if adjacent vertices in G are assigned distinct 'colors', i.e., $\{u, v\} \in E$ implies $C(u) \neq C(v)$. If a k-coloring exists for G, then G is *k-colorable*. The *chromatic number* of G denoted by $\chi(G)$ is the minimum number k such that G is k-colorable. The *graph coloring problem* asks for a coloring of an input graph G using the $\chi(G)$ colors. The *chromatic number problem* is to determine $\chi(G)$ for any input graph G. Both problems are abstractions of problems such as constructing examination time tables: the vertices represent courses such as *Computer Science I* or *Introduction to Ichthyology*. We have an edge $\{u, v\}$ if there are students taking both courses u and v. Nodes colored with the same color will hold examinations in the same time slot. So the graph is k-colorable iff there is a conflict-free examination time table with k time slots. Determining the chromatic number is clearly an optimization problem and again there is no known polynomial-time algorithm for it. We can derive a corresponding recognition problem as follows:

GRAPH COLORABILITY PROBLEM
GIVEN: Graph G and integer k.
PROPERTY: G is k-colorable.

Proposition B. *The graph coloring problem, the chromatic number problem and the graph colorability are polynomially equivalent.* (see Exercises for proof)

The last pair of problems considered in this section relate to the satisfiability problem that arises in the subjects of logic and mechanical theorem proving. The *satisfying assignment problem* is the problem where, given a CNF formula F, we are required to find an assignment that satisfies F (if F is satisfiable). The definitions for this problem appear in the appendix of this chapter. The recognition problem version of this problem is the following:

SATISFIABILITY PROBLEM (SAT)
GIVEN : A CNF formula F.
PROPERTY : F is satisfiable.

Proposition C. *The satisfying assignment problem and* SAT *are polynomially equivalent.* (see Exercises for a proof)

3.3 Many-One Reducibility

In showing that the TSO is tractable if the TSD is tractable (or vice-versa), we have illustrated the very important idea of 'reducing one problem to another'. In this section, we formalize one version of this idea; chapter 4 embarks on a more systematic study.

We now use the concept of a transformation and transducer as defined in chapter 2 (section 2).

Definition 1 Let $(\Sigma, L), (\Gamma, L')$ be languages, and $f : \Sigma^* \to \Gamma^*$ be a transformation computed by a transducer M. We say that L is *many-one reducible* to L' *via* f (or *via* M) if for all $x \in \Sigma^*$,

$$x \in L \text{ iff } f(x) \in L'.$$

If, in addition, M runs in deterministic polynomial time, then we say L is *many-one reducible* to L' *in polynomial time* and write

$$L \leq_m^P L'.$$

■

The above reducibility is also known as *Karp reducibility*. To illustrate this reducibility, we consider a simplification of the satisfiability problem. If k is any positive integer, a kCNF formula is one whose clauses have exactly k literals. Consider the problem of recognizing satisfiable 3CNF formulas: call this problem 3SAT. Thus 3SAT $\subseteq$ SAT (as languages). We show:

Lemma 1 SAT $\leq_m^P$ 3SAT.

In a formal proof of this lemma, we would have to construct a deterministic transducer M running in polynomial time such that for any input word x over the alphabet of SAT, if x encodes (resp. does not encode) a satisfiable CNF formula, then $f_{\mathrm{M}}(x)$ encodes (resp. does not encode) a satisfiable 3CNF formula. First of all, note that it is not hard for M to verify whether the input is well-formed (i.e. encodes a CNF formula) or not. If x is ill-formed, M can easily output a fixed unsatisfiable formula. So assume that x does encode a CNF formula F. We will show how to systematically construct another 3CNF formula G such that G is satisfiable iff F is. The actual construction of the transducer M to do this is a straightforward but tedious exercise which we omit. (However, the reader should convince himself that it can be done.)

For each clause C in F we will introduce a set of clauses in G. Let $C = \{u_1, u_2, \ldots, u_m\}$. First assume $m > 3$. We introduce the set of clauses

$$\{u_1, u_2, z_1\}, \{\bar{z_1}, u_3, z_2\}, \{\bar{z_2}, u_4, z_3\}, \ldots, \{\bar{z}_{m-4}, u_{m-2}, z_{m-3}\}, \{\bar{z}_{m-3}, u_{m-1}, u_m\}$$

where $z_i, (i = 1, \ldots, m-3)$ are new variables. If $m = 3$, then we simply put C into G. If $m = 2$, we introduce the following two clauses:

$$\{u_1, u_2, z_1\}, \{\bar{z_1}, u_1, u_2\}.$$

If $m = 1$, we introduce the following four clauses:

$$\{u_1, z_1, z_2\}, \{u_1, \bar{z_1}, z_2\}, \{u_1, z_1, \bar{z_2}\}, \{u_1, \bar{z_1}, \bar{z_2}\}.$$

G has no other clauses except those introduced for each C in F as described above. To show that G is satisfiable iff F is satisfiable, it is easy to verify that for any satisfying assignment I to the variables occurring in F, we can extend I to the newly introduced variables (the z_i's) so that the extension also satisfies all the clauses in G. Conversely, if I is an assignment that satisfies G, then the restriction of I to the variables occurring in F will satisfy F. The reader should check these claims. This concludes the proof of lemma 1.

The following two lemmas concerning $\leq_m^P$ are basic.

Lemma 2 *If $L \leq_m^P L'$ and $L' \in P$ then $L \in P$.*

Proof. By definition, there is a deterministic transducer M running in $p(n)$ time (for some polynomial $p(n)$) such that $L \leq_m^P L'$ via f_{M}. Also, there exists an acceptor M$'$ which accepts L' in $p'(n)$ time (for some polynomial $p'(n)$). We construct a machine N which accepts L by 'calling' M and M$'$ as 'subroutines'. On input x, N first computes $f_{\mathrm{M}}(x)$ by simulating M. Then N imitates the actions of M$'$ on input $f_{\mathrm{M}}(x)$, accepting iff M$'$ accepts. Clearly N accepts L and runs in $O(p(n)+p'(p(n)))$ time, which is still polynomial. **Q.E.D.**

Lemma 3 *$\leq_m^P$ is transitive.*

Proof. Let $L \leq_m^P L'$ via some transducer M and $L' \leq_m^P L''$ via some M$'$, where both M and M$'$ run in polynomial time. The lemma follows if we construct another polynomial-time M$''$ such that L is many-one reducible to L'' via M$''$. This is done in a straightforward manner where M$''$ on input x simulates M on x, and if y is the output of M then M$''$ simulates M$'$ on y. The output of M$''$ is the output of M$'$ on y. Clearly $|y|$ is polynomial in $|x|$ and hence M$''$ takes time polynomial in $|x|$. **Q.E.D.**

The next definition embodies some of the most important ideas in this subject.

Definition 2 Let K be a class of languages. A language L' is *K-hard* (*under $\leq_m^P$*) if for all $L \in K$, $L \leq_m^P L'$. L' is *K-complete* (*under $\leq_m^P$*) if L' is K-hard (under $\leq_m^P$) and $L' \in K$. ■

Since we will not consider other types of reducibilities in this chapter, we will omit the qualification 'under $\leq_m^P$' when referring to hard or complete problems. We also say L is *hard* (resp. *complete*) for K if L is K-hard (resp. K-complete). From lemma 2, we have

Corollary 4 *Let L be NP-complete. Then $L \in P$ iff $P = NP$.*

Thus the question whether $P = NP$ is reduced to the fundamental tractability of *any NP*-complete problem. But it is not evident from the definitions that *NP* contains any complete problem.

3.4 Cook's Theorem

In this section we shall prove

Theorem 5 (Cook) SAT *is NP-complete.*

This was historically the first problem shown to be *NP*-complete[3] and it remains a natural problem whereby many other problems L can be shown to be *NP*-hard by reducing SAT to L (perhaps by using transitivity). This avoids the tedious proof that would be necessary if we had to directly show that every language in *NP* can be reduced to L. We just have to go through this tedium once, as in the proof of Cook's theorem below.

To show that a problem L is *NP*-complete we have to show two facts: that L is in fact in *NP* and that every problem in *NP* can be reduced to L in polynomial time. For most *NP*-complete problems the first fact is easy to establish. In particular, the three recognition problems in section 2 are easily shown to be in *NP*: for the TSD problem, we can easily construct a Turing acceptor which on input $\langle D, b\rangle$ uses nondeterminism to guess a tour π and then deterministically computes the tour length $D(\pi)$, accepting iff $D(\pi) \leq b$. To see that this is a correct nondeterministic procedure, we note that if the input is in TSD, then there is a π which satisfies $D(\pi) \leq b$ and the acceptor will accept since it will guess π along some computation path. Conversely, if the input is not in TSD, then every choice of π will lead to a rejection and by definition, the acceptor rejects the input. Since the procedure clearly takes only a polynomial number of steps, we conclude that TSD is in *NP*. Similarly, for the graph colorability problem (resp. SAT), the acceptor guesses a coloring of the vertices (resp. an assignment to the variables) and verifies if the coloring (resp. assignment) is valid. Let us record these observations in:

Lemma 6 *The TSD, the graph colorability problem and SAT are in NP.*

We will use simple Turing machines as defined in chapter 2. In particular, the transition table $\delta(M)$ of simple Turing acceptor M is a set of quintuples

$$\langle q, b, q', b', d\rangle$$

saying that in state q and scanning b on the tape, M can move to state q', change b to b', and move the tape-head in the direction indicated by $d \in \{-1, 0, +1\}$. In chapter 2 we showed that a multi-tape machine can be simulated by a 1-tape machine with at most a quadratic blow-up in the time usage. A very similar proof will show:

[3]Cook's theorem in Complexity Theory is comparable to Gödel's Incompleteness Theorem in Computability Theory: both play a paradigmatic role (in the sense of Kuhn [10]). In Kuhn's analysis, a *scientific paradigm* involves a system of views, methodology and normative values for doing research. A further parallel is that the relation of P to *NP* can be compared to the relation between the recursive sets and the recursively enumerable sets.

Lemma 7 *If* M *is a multi-tape Turing acceptor accepting in time* $t(n)$ *then there exists a simple Turing acceptor* N *accepting the same language in time* $(t(n)+n)^2$*. Moreover,* N *is deterministic if* M *is deterministic.*

The import of this lemma is that for the purposes of defining the class of languages accepted in polynomial time in deterministic (resp. nondeterministic) mode we could have used the simple Turing machines instead of multitape Turing machines. This is also a simple demonstration of the polynomial simulation thesis described in chapter 1.

Proof that every language $L \in NP$ is reducible to SAT: The rest of this section is devoted to this proof. By the preceding lemma, we may assume that L is accepted by a simple Turing acceptor M in time $p(n)$ for some polynomial $p(n)$. To show that L is reducible to SAT, we must show a transducer N (computing the transformation f_{N}) running in polynomial time such that for each input word x over the alphabet of L, $f_{\mathrm{N}}(x)$ is in SAT iff x is in L. The word $f_{\mathrm{N}}(x)$ will encode a CNF formula. We shall describe this formula only, omitting the tedious construction of N. As usual, once the transformation f_{N} is understood, the reader should have no conceptual difficulty in carrying out the necessary construction.

For the following discussion, let x be a fixed input of M where $|x| = n$. Let

$$C_0, C_1, \ldots, C_{p(n)} \tag{3.1}$$

be the first $p(n)$ configurations in some computation path of M on x. (If the computation path has less than $p(n)$ configurations, then its last configuration is repeated.) The CNF formula (encoded by) $f_{\mathrm{N}}(x)$ will specify conditions for the existence of an accepting computation path of the form (3.1), i.e., the formula will be satisfiable iff there exists an accepting computation path (3.1). In order to do this, $f_{\mathrm{N}}(x)$ uses a large number of Boolean variables which we now describe. Each variable denotes a proposition ('elementary statement') about some hypothetical computation path given by (3.1). We will assume that the instructions (tuples) in $\delta(\mathrm{M})$ are numbered in some canonical way from 1 to $|\delta(\mathrm{M})|$.

Variable	*Proposition*
$S(q,t)$	Configuration C_t is in State q.
$H(h,t)$	The tape-Head is scanning cell h in configuration C_t.
$T(b,h,t)$	Cell h contains the Tape symbol b in configuration C_t.
$I(j,t)$	Instruction j in $\delta(M)$ is executed in the transition $C_t \vdash C_{t+1}$.

In this table, the meta-variable[4] t is the 'time indicator' that ranges from 0 to $p(n)$, the meta-variable q ranges over the states in M, and the meta-variable$b$ ranges

[4]I.e., the variable we use in describing actual Boolean variables of $f_{\mathrm{N}}(x)$. These meta-variables do not actually appear in $f_{\mathrm{N}}(x)$.

over the tape symbols (including the blank) of M. The meta-variable h is the 'head position indication': it ranges over the $2p(n)+1$ values

$$-p(n), -p(n)+1, \ldots, -1, 0, 1, 2, \ldots, p(n)-1, p(n)$$

since in $p(n)$ steps, M cannot visit any cell outside this range. The meta-variable j is the "instruction indicator" that ranges from 1 to $|\delta(M)|$, the number of instructions in $\delta(M)$. Since the number of values for q, b and j is $O_{\mathrm{M}}(1)$, and there are $O(p(n))$ values for t and h, we see that there are $O((p(n))^2)$ variables in all. Therefore each variable can be encoded by a word of length $O(\log n)$. We emphasize that these are *Boolean* variables; so the above propositions associated with them are purely informal.

We now introduce clauses into the CNF formula $f_{\mathrm{N}}(x)$ that enforce the above interpretation of the Boolean variables. By this we mean that if v is a Boolean variable that stands for the proposition P_v as described in the above table then for any satisfying assignment I to $f_{\mathrm{N}}(x)$, there exists an accepting computation path of the form (3.1) such that $I(v) = 1$ precisely when the proposition P_v is true of (3.1). For instance, the variable $S(q_0, 0)$ stands for the proposition

The state in configuration C_0 is the initial state q_0.

It is clear that this proposition is true of (3.1). Hence we must set-up $f_{\mathrm{N}}(x)$ so that if assignment I satisfies it then necessarily $I(S(q_0, 0)) = 1$.

The clauses in $f_{\mathrm{N}}(x)$ correspond to the following eight conditions. For each condition we specify a set of clauses that we will include in $f_{\mathrm{N}}(x)$.

1. The configuration C_0 is the initial configuration of M on input x. Suppose that $x = a_1 a_2 \cdots a_n$ where a_i are input symbols. To enforce condition 1, we introduce the following set of clauses, each containing only one variable:

   $$\{S(q_0, 0)\},$$

   $$\{T(a_1, 1, 0)\}, \{T(a_2, 2, 0)\}, \ldots, \{T(a_{n-1}, n-1, 0)\}, \{T(a_n, n, 0)\}$$

   $$\{T(\square, h, 0)\} \quad \text{for} \quad h \notin \{1, \ldots, n\}$$

   Thus, for the first clause to be true, the variable $S(q_0, 0)$ must be assigned 1, a fact that we have already concluded must hold. Similarly, the next n clauses ensure that the input x is contained in cells 1 to n. The remaining clauses ensure that the rest of the cells contain the blank symbol $\square$.

2. In each configuration C_t, M is in exactly one state.
   Let S_t be the set of variables $\{S(q, t) : q \text{ is a state of M}\}$. Then condition 2 amounts to ensuring that exactly one variable in S_t is assigned '1'. For this,

we have a convenient abbreviation. If X is any set of Boolean variables, let $\mathbf{U}(X)$ denote the set of clauses consisting of the clause X and the clause $\{\bar{u}, \bar{v}\}$ for each pair of distinct variables u, v in X. For example, if $X = \{x, y, z\}$ then

$$\mathbf{U}(X) = \{\{x, y, z\}, \{\bar{x}, \bar{y}\}, \{\bar{x}, \bar{z}\}, \{\bar{y}, \bar{z}\}\}.$$

Note that an assignment to X satisfies $\mathbf{U}(X)$ iff the assignment assigns a value of '1' to exactly one variable in X. ($\mathbf{U}$ stands for 'unique'.) Therefore, for each t, condition 2 is enforced by introducing into $f_{\mathrm{N}}(x)$ all the clauses in $\mathbf{U}(S_t)$.

3. There is a unique symbol in each cell h of each C_t.
   For each h, t, condition 3 is enforced by the clauses in $\mathbf{U}(T_{h,t})$ where $T_{h,t}$ is defined to be the set $\{T(b, h, t) : b \text{ is a tape symbol of M}\}$.

4. There is a unique head position h in each C_t. This is enforced by the clauses in $\mathbf{U}(H_t)$ where $H_t = \{H(h, t) : h = -p(n), -p(n) + 1, \ldots, -1, 0, 1, \ldots, p(n)\}$.

5. The last configuration is accepting.
   Introduce the single clause $\{S(q_a, p(n))\}$ where q_a is the accept state.

6. Cells that are not currently scanned in C_t must contain the same symbol as in C_{t+1}.
   This can be ensured by introducing the three-literal clause

   $$\{\overline{T(b, h, t)}, \overline{T(c, h, t + 1)}, H(h, t)\}.$$

   for all h, t and tape symbols b, c, where $b \neq c$.

7. For each $t < p(n)$ there is a unique instruction that causes the transition $C_t \vdash C_{t+1}$.
   This is ensured by the clauses in $\mathbf{U}(I_t)$ where $I_t = \{I(j, t) : j = 1, \ldots, |\delta(\mathrm{M})|\}$.

8. Changes in successive configurations follow according to the transition table of M.
   For each state q, and tape symbol b (possibly $\square$), let $\delta(q, b) \subseteq \{1, \ldots, |\delta(\mathrm{M})|\}$ be the set instruction numbers such that $j \in \delta(q, b)$ iff the first two components of the jth instruction are q and b. In other words, $j \in \delta(q, b)$ iff the jth instruction is applicable to any configuration whose state and scanned symbol are q and b respectively. For each t, h, we introduce this clause:

   $$\{\overline{T(b, h, t)}, \overline{S(q, t)}, \overline{H(h, t)}\} \cup \{I(j, t) : j \in \delta(b, q)\}.$$

   In addition, for each j where the jth instruction is $\langle q, b, q', b', d\rangle$, introduce three clauses:

   $$\{\overline{I(j, t)}, S(q', t + 1)\},$$
   $$\{\overline{I(j, t)}, \overline{H(h, t)}, T(b', h, t + 1)\}$$

$$\{\overline{I(j,t)}, \overline{H(h,t)}, H(h+d,t+1)\}$$

We ought to remark that to ensure that if the machine gets stuck or halts in less than $p(n)$ steps, then we assume there are rules in $\delta(\mathrm{M})$ to replicate such configurations.

It is curious to observe that, with the exception of the first condition, the rest of the clauses in $f_{\mathrm{N}}(x)$ are not directly related to the input x.

To show that the CNF formula $f_{\mathrm{N}}(x)$ consisting of all the clauses introduced under the above eight conditions is correct, we have to show that $f_{\mathrm{N}}(x)$ is satisfiable iff x is in L. Suppose $f_{\mathrm{N}}(x)$ is satisfiable by some assignment I. Then we claim that an accepting computation path of the form (3.1) exists. This is seen inductively. Clearly the configuration C_1 is uniquely 'determined' by condition 1. Suppose that $C_1, \ldots, C_t$ are already defined. Then it is easy to define C_{t+1}. Hence sequence (3.1) can be defined. But condition 5 implies that $C_{p(n)}$ is accepting. Conversely, if an accepting computation path exists then it determines an assignment that satisfies $f_{\mathrm{N}}(x)$.

Finally, we indicate why N can be constructed to run in polynomial time. Note that for any set X of k variables, the formula $\mathbf{U}(X)$ contains $O(k^2)$ clauses, and these clauses can be generated in $O(k^3)$ time. It is not hard to verify that all the clauses in conditions 1-8 can be generated in $O((p(n))^3)$ time. This concludes the proof of Cook's theorem.

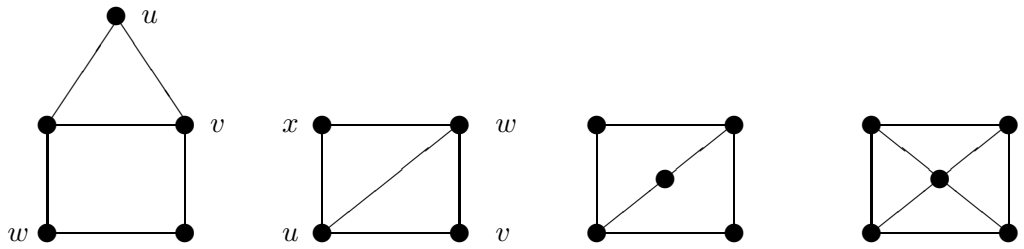
We have the following interesting by-product of the above proof:

Corollary 8 *There is a polynomial-time computable transformation t of an arbitrary Boolean formula F to a* CNF *formula $t(F)$ such that F and $t(F)$ are co-satisfiable (i.e., F is satisfiable iff $t(F)$ is satisfiable).*

See the Exercises for a direct proof of this result.

3.5 Some Basic *NP*-Complete Problems in Graph Theory

We study a few more *NP*-complete problems. This will give the reader a feeling for some of the techniques used in proving *NP*-completeness. We typically prove that a language L is *NP*-hard by reducing a known *NP*-hard problem L' to L. This indirect approach is particularly effective if one chooses an L' that is rather similar to L: the list of over 300 *NP*-complete problems in [5] is a good resource when making this choice. We will assume the standard terminology of graph theory and throughout this book, the terms 'node' and 'vertex' are fully interchangeable. The following four problems will be considered in this section.



(a) (b) (c) (d)

Figure 3.1: Some graphs

VERTEX COVER PROBLEM. GIVEN: Graph $G = (V, E)$ and integer k. PROPERTY: G has a vertex cover of size $\leq k$.

A *vertex cover* of G is a subset $V' \subseteq V$ such that $\{u, v\} \in E$ implies that u or v is in V'. For example, the graph in Figure 3.1(a) has a vertex cover $\{u, v, w\}$ of size 3 but none of size 2.

CLIQUE PROBLEM. GIVEN: $G = (V, E)$ and k. PROPERTY: There exists a clique of size $\geq k$ in G.

A *clique* of G is a subset $V' \subseteq V$ such that each pair of distinct vertices u and v in V' are adjacent in G. For example, the graph in Figure 3.1(b) has two cliques $\{u, v, w\}$ and $\{u, x, w\}$ of size 3 but none of size 4.

INDEPENDENT SET PROBLEM. GIVEN: $G = (V, E)$ and k. PROPERTY: There exists an independent set of size $\geq k$ in G.

An *independent set* of G is a subset $V' \subseteq V$ such that no two distinct vertices u and v in V' are adjacent in G. Thus, the graph in Figure 3.1(b) has an independent set $\{v, x\}$ of size 2 but none of size ≥ 3.

HAMILTONIAN CIRCUIT PROBLEM. GIVEN: $G = (V, E)$. PROPERTY: G has a Hamiltonian circuit.

A *Hamiltonian circuit* $C = (v_1, v_2, \ldots, v_n)$ of G is a cyclic ordering of the set of vertices of G such that $\{v_i, v_{i+1}\}$ (for $i = 1, \ldots, n$) are edges of G (here v_{n+1} is taken to be v_1). For instance, the graph in Figure 3.1(d) has a Hamiltonian circuit, but the one in Figure 3.1(c) has none.

These four problems are easily seen to be in *NP* using the usual trick of guessing and verifying. The first three problems appear very similar. The precise relationship between vertex covers, cliques and independent sets is given by the following result. The *complement* of $G = (V, E)$ is *co-G* $= (\bar{V}, \bar{E})$ where $\bar{V} = V$, $\bar{E} = \{\{u, v\} : u \neq v, \{u, v\} \notin E\}$.

Lemma 9 *The following statements are equivalent:*

(a) V' *is a vertex cover for* G.

(b) $V - V'$ *is an independent set for* G.

(c) $V - V'$ *is a clique in co-*G.

The proof is straightforward and left to the reader. From this lemma, it is evident that the three problems are inter-reducible problems. So it is sufficient to show one of them *NP*-complete. We begin by showing that Vertex Cover is *NP*-hard; this was shown by Karp.

Theorem 10 3SAT $\leq_m^P$ *Vertex Cover.*

Proof. Let F be a 3CNF formula with m clauses and n variables. We will describe a graph $G = (V, E)$ derived from F such that G has a vertex cover of size at most $2m + n$ iff F is satisfiable. The actual construction of a transducer to accomplish the task is a routine exercise which we omit.

The graph G contains two types of subgraphs: For each variable x in F, we introduce a pair of adjacent nodes labeled by x and $\bar{x}$, as in Figure 3.2(a).

For each clause $\{u_1, u_2, u_3\}$, we introduce a triangle with nodes labeled by the literals in the clause (Figure 3.2(b)). To complete the description of G, we introduce edges connecting a node in any pair with a node in any triangle whenever the two nodes are labeled by the same literal. For example, the graph for the formula $\{\{x_1, x_2, x_3\}, \{\bar{x}_1, x_2, x_3\}, \{\bar{x}_1, \bar{x}_2, x_3\}\}$ is shown in Figure 3.3.

We now claim that F is satisfiable iff G has a vertex cover of size $2m + n$. If F is satisfied by some assignment I, then we choose V' to consist of (a) those nodes in the pairs of G labeled by literals u where $I(u)$=1 and (b) any two nodes from each triangle of G, provided that all the nodes labeled by literals u where $I(u)$=0 (there are at most two such per triangle) are among those chosen. There are n nodes chosen in (a) and $2m$ nodes chosen in (b). Furthermore, one observes that every edge of G is incident to some node in V', and thus V' is a vertex cover. Conversely, if G has a vertex cover V' of size at most $2m + n$, then we easily see that each pair (resp. triangle) must contain at least one (resp. two) nodes in V'. This means that

(a) pair (b) triangle

Figure 3.2: Constructions for vertex cover

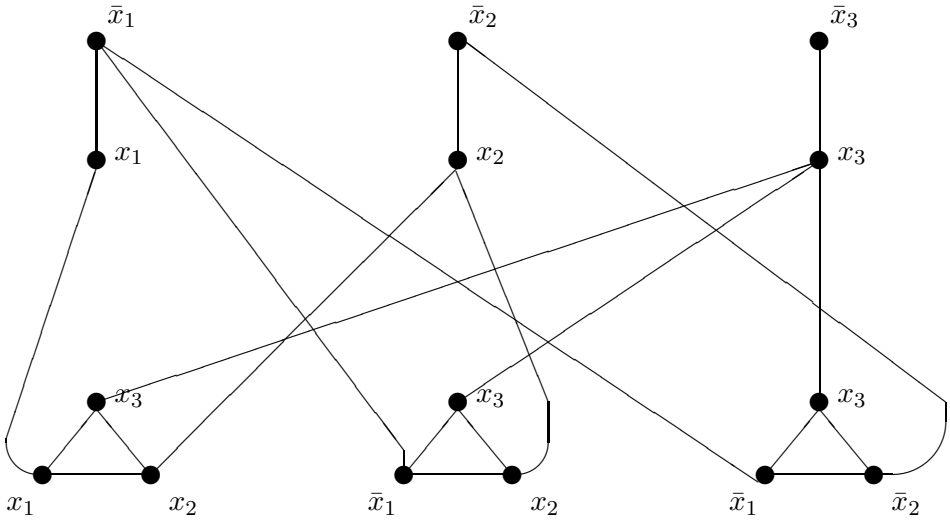
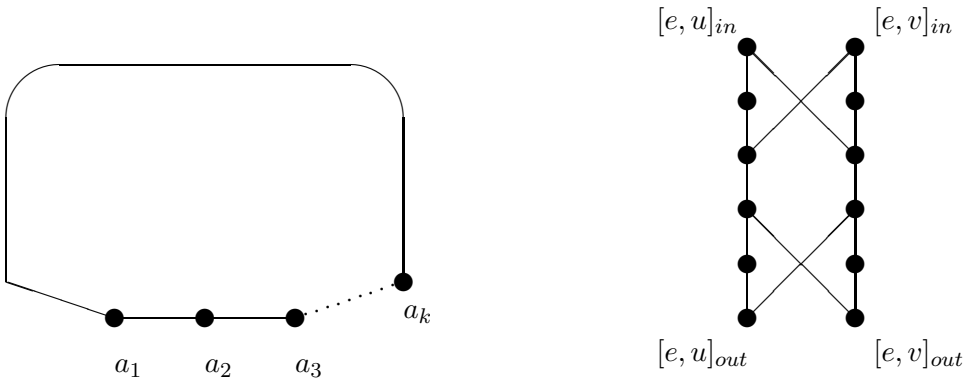


Figure 3.3: Reduction of 3SAT to Vertex Cover



(a) ring (b) gadet

Figure 3.4: Constructions for Hamiltonian circuit

in fact the vertex cover has exactly one vertex from each pair and two vertices from each triangle. The assignment which assigns to each variable x a value of 1 (resp. 0) iff the node in a pair labeled by x (resp. $\bar{x}$) is in V' is seen to satisfy F. **Q.E.D.**

We next show that the Hamiltonian Circuit problem is *NP*-hard, a result of Karp.

Theorem 11 *Vertex Cover* $\leq_m^P$ *Hamiltonian Circuit.*

Proof. Given $G = (V, E)$ and k, we construct $\bar{G} = (\bar{V}, \bar{E})$ such that G has a vertex cover with at most k nodes iff $\bar{G}$ has a Hamiltonian circuit. $\bar{G}$ has two types of vertices. The first type consists of k nodes $a_1, \ldots, a_k$ connected in a ring, i.e., a_i is adjacent to both a_{i+1} and a_{i-1} where subscript arithmetic is modulo k. See Figure 3.4(a). The second type of vertices is introduced by constructing a 'gadget' of 12 nodes for each edge $e = \{u, v\} \in E$ (Figure 3.4(b)). Four nodes in the gadget are specially labeled as

$$[e,u]_{in}, [e,u]_{out}, [e,v]_{in}, [e,v]_{out}.$$

So the total number of vertices in $\bar{V}$ is $k + 12|E|$. In addition to the edges in the ring and the gadgets, other edges are introduced corresponding to each node u in V thus: let $e_1, \ldots, e_m$ be the edges in E which are incident on u. We form a 'u-chain' by introducing the following edges:

$$\{[e_1,u]_{out}, [e_2,u]_{in}\}, \{[e_2,u]_{out}, [e_3,u]_{in}\}, \ldots, \{[e_{m-1},u]_{out}, [e_m,u]_{in}\}. \tag{3.2}$$

These edges string together the gadgets corresponding to the edges $e_1, \ldots, e_m$. This stringing of the gadgets imposes an arbitrary ordering of the e_i's. We also connect

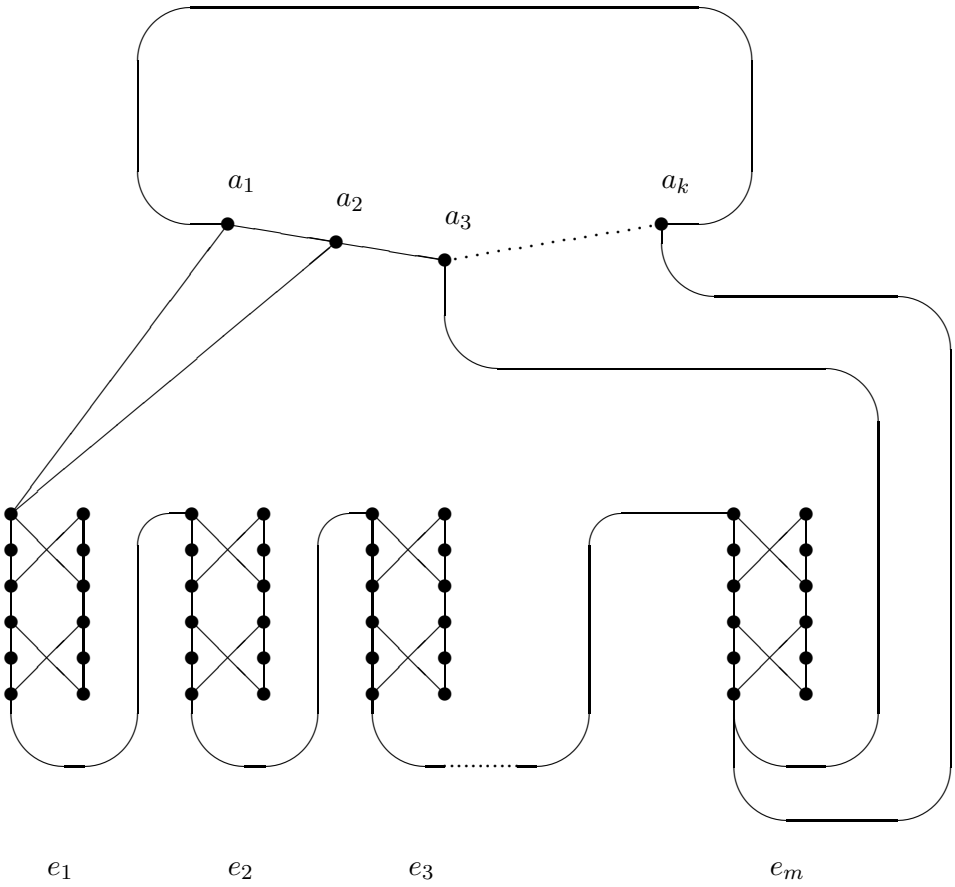


Figure 3.5: A u-chain: $e_1, \ldots, e_m$ are edges incident on node u

$[e_1, u]_{in}$ and $[e_m, u]_{out}$ to each of the nodes $a_i (i = 1, \ldots, k)$ in the ring. See Figure 3.5. Our description of $\bar{G}$ is now complete. It remains to show that G has a vertex cover of size at most k iff $\bar{G}$ has a Hamiltonian circuit. Suppose $U = \{u_1, \ldots, u_h\}$ is a vertex cover of G, $h \leq k$. For each node u_j in U, we define a path p_j from a_j to a_{j+1}. Suppose $e_1, \ldots, e_m$ are the edges in E which are incident on u_j. Let $e_1, \ldots, e_m$ appear in the order as determined by the u_j-chain. The path p_j will include all the edges in (3.2) and the two edges $\{a_j, [e_1, u_j]_{in}\}$ and $\{[e_m, u_j]_{out}, a_{j+1}\}$. The path p_j must also connect the nodes $[e_i, u_j]_{in}$ and $[e_i, u_j]_{out}$ for each $i = 1, \ldots, m$. We consider two ways to do this:

The first way (Figure 3.6(a)) visits every node in the gadget of e_i but the second way (Figure 3.6(b)) visits only half of the nodes. We route p_j through the gadget of e_i using the first way if the other node that e_j is incident upon is not in the vertex cover U; otherwise we use the second way. This completes the definition of the path p_j. The concatenation of $p_1, p_2, \ldots, p_h$ is seen to be a path $P(U)$ from a_1

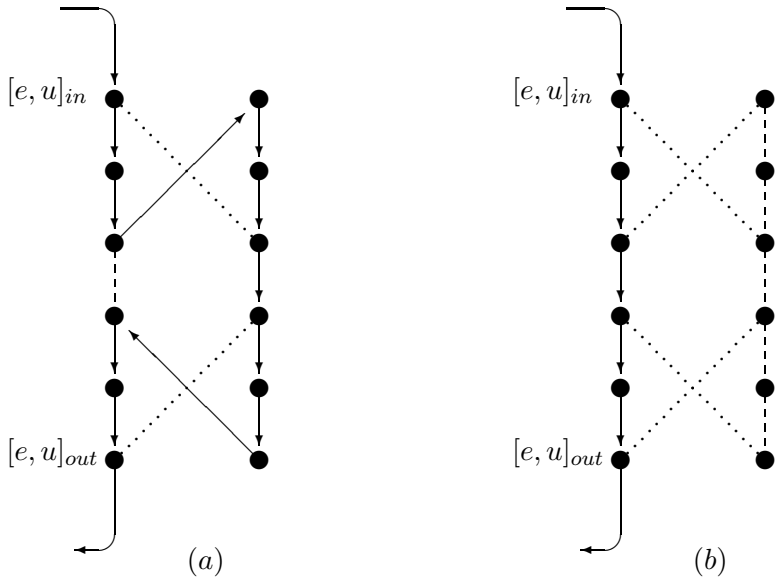


Figure 3.6: The only two ways a Hamiltonian path can route through a gadget

to a_{h+1} which visits every node in each gadget. Finally, this path $P(U)$ is made into a circuit by connecting $a_{h+1}, a_{h+2}, \ldots, a_k$ and a_1 using edges in the ring structure. The result is a Hamiltonian circuit for $\bar{G}$.

Conversely, suppose there is a Hamiltonian circuit C for $\bar{G}$. The circuit is naturally broken up into paths whose end-points (but not other vertices) are in the ring. For each such path p that is non-trivial (i.e., has more than one edge), we claim that there exists a unique vertex $u(p) \in V$, such that p visits all and only the gadgets in the u-chain: this is because if p visits any gadget associated with an edge $e = \{u, v\}$, it must either enter and exit from the nodes $[e,u]_{in}, [e,u]_{out}$ or enter and exit from the nodes $[e,v]_{in}, [e,v]_{out}$. For instance, it is not possible to enter at $[e,v]_{in}$ and exit from $[e,u]_{out}$ (why?). In fact, if p enters and exits from the gadget using the nodes $[e,u]_{in}$ and $[e,u]_{out}$, then there are essentially two ways to do this, as indicated in Figure 3.6. It is easily seen that set of all such vertices $u(p)$ forms a vertex cover $U(C)$ of G. Since there are at most k such paths in C, the vertex cover $U(C)$ has size at most k. **Q.E.D.**

3.6 **Three More Hard Problems

[4]** Optional section. The problems are hard in the sense of being non-trivial to show in *NP*.

In all the problems seen so far, it was rather easy to show that they are in *NP*, but showing them to be *NP*-hard required a bit more work. We now consider two problems for which it is non-trivial to show that they are in *NP*.

PRIMALITY TESTING (*Primes*)
Given: Given a binary number n.
Property: n is a prime.

INTEGER LINEAR PROGRAMMING (ILP)
Given: An integer $m \times n$ matrix A, and an integer m-vector $\mathbf{b}$.
Property: There is an integer n-vector $\mathbf{x}$ such that the system of linear inequalities $A\mathbf{x} \geq \mathbf{b}$ hold.

These problems have practical importance. One important use of primes is in cryptographic techniques that rely on the availability of large prime numbers (several hundred bits in length). We note first that the complement of *Primes* is *Composites*, the problem of testing if a binary number is composite. It is easy to see that *Composites* is in *NP*: to test if n is composite, first guess a factor m less than n and then check if m divides n exactly. Combined with the result that *Primes* is in *NP*, we conclude

$$Primes \in NP \cap co\text{-}NP. \tag{3.3}$$

Now *Primes* is not known to be *NP*-complete and (3.3) gives strong evidence that it is not. This is because if it were *NP*-complete then it would be easy [5] to conclude from (3.3) that *NP*=*co-NP*. This is unlikely, for example, based on extensive experience in the area of mechanical theorem proving.

The ILP problem is also known as the *Diophantine linear programming problem* and is studied in [12]. Let *Rational LP* (or RLP) denote the variant of ILP where we allow the input numbers and output numbers to be rationals. The rational LP problem was shown to be in *P* in 1979 by Khachian [8]; this solved a long standing open problem. See [1] for an account. Previously, all algorithms for the rational linear programming problem had been based on the Simplex Method, and it was shown by Klee and Minty[9] that such an approach can be exponential in the worst case.[6] Alternatives to Khachian's algorithm are now known but they do not seem competitive with the simplex method(s) in practice. However, this situation may have changed recently because of a new polynomial algorithm given by Karmarkar.

[5] See chapter 4.

[6] Recently major progress have been made in understanding the average behavior of the Simplex-based algorithms. They essentially confirm the experimental evidence that, on the average, these methods take time linear in the number m of constraints, and a small polynomial in the dimension n. See [2].

3.6.1 Primality testing

This subsection requires some elementary number theory; Hardy and Wright [6] may serve as an excellent reference. We follow the usual convention that the number 1 does not count as a prime: thus the smallest prime is the number 2. The greatest common divisor of two integers m and n is denoted by $gcd(m,n)$, with the convention $gcd(0,0) = 0$. Note that $gcd(0,n) = gcd(n,0) = |n|$. Two numbers m,n are *relatively prime* if $gcd(m,n) = 1$. Let

$$\mathbb{Z}_n = \{0,1,\ldots,n-1\} \quad \text{and} \quad \mathbb{Z}_n^* = \{m \in \mathbb{Z}_n : gcd(m,n) = 1\}.$$

Euler's *totient function* $\phi(n)$ is defined be the number of integers in $\{1,2,\ldots,n\}$ that are relatively prime to n; thus $\phi(n) = |\mathbb{Z}_n^*|$. For example,

$$\phi(1) = \phi(2) = 1, \phi(3) = \phi(4) = \phi(6) = 2, \phi(5) = 4.$$

The result that *Primes* is in *NP* is due to Pratt [13]. To motivate Pratt's technique, we state a gem of number theory [6, p.63]:

Theorem 12 (Fermat's little theorem) *If p is prime and p does not divide x then $x^{p-1} \equiv 1(\mathrm{mod}p)$.*

For example, if $p = 5$ and $x = 3$ then we have $x^2 \equiv 4, x^4 \equiv 16 \equiv 1$ (all mod5). We have a partial converse to this theorem, due to Lucas [6, p.72]

Theorem 13 *Let n,x be any numbers. If $x^{n-1} \equiv 1(\mathrm{mod}n)$ and $x^m \not\equiv 1(\mathrm{mod}n)$ for all divisors m of $n-1$, then n is prime.*

Proof. We first note two well-known facts: (i) the congruence $ax \equiv b(\mathrm{mod}n)$ has a solution in x iff $gcd(a,n)$ divides b [6, p.51]. (ii) Euler's totient function satisfies $\phi(n) \leq n-1$ with equality iff n is prime.

To prove the theorem, we assume $x^{n-1} \equiv 1(\mathrm{mod}n)$ and $x^m \not\equiv 1(\mathrm{mod}n)$ for all divisors m of $n-1$. We will show that n is prime. Using fact (i), we conclude from $x^{n-1} \equiv 1(\mathrm{mod}n)$ that $gcd(x,n) = 1$. We claim that $x^1, x^2, \ldots, x^{n-1}$ are distinct elements (mod$n$): otherwise if $x^i \equiv x^{i+m}(\mathrm{mod}n)$ for some $1 \leq i < i+m < n$, then $x^m \equiv 1(\mathrm{mod}n)$. The smallest positive m with this property must divide $n-1$: for the sake of contradiction, let us suppose otherwise. So $m > 1$ and $n-1$ may be written as $n-1 = am+b$ where $1 \leq b < m$. Then $1 \equiv x^{n-1} \equiv x^b(\mathrm{mod}n)$, contradicting our choice of m. On the other hand, we have assumed that $x^m \equiv 1(\mathrm{mod}n)$ fails for all divisors m of $n-1$, again a contradiction. Therefore $x^1, x^2, \ldots, x^{n-1}$ (all mod$n$) are distinct members of $\mathbb{Z}_n^*$. We conclude that $\phi(n) \geq n-1$ and the primeness of n follows from fact (ii). **Q.E.D.**

If we directly apply the test of Lucas to see if a given n is prime, we must check each divisor of $n-1$ for the desired property. In the worst case, this requires non-polynomial time since there can be many divisors. To see this we must remember that the size of the input n is only $\log n$. Pratt noticed that we only need to check all primes that divide $n-1$:

Lemma 14 *Let x, n be any numbers and $x^{n-1} \equiv 1(\mathrm{mod} n)$. If for every prime p dividing $n-1$, we have $x^{\frac{n-1}{p}} \not\equiv 1(\mathrm{mod} n)$ then n is prime.*

Proof. Suppose that for every prime p dividing $n-1$, we have $x^{\frac{n-1}{p}} \not\equiv 1(\mathrm{mod} n)$. Our desired result follows from the previous lemma if we show that for every divisor m of $n-1$, $x^m \not\equiv 1(\mathrm{mod} n)$. Suppose to the contrary that $x^m \equiv 1$ for some $m > 0$. If m is the smallest positive number with this property then we have already argued in the last proof that m divides $n-1$. Hence there is a prime p that divides $\frac{n-1}{m}$. Then $x^m \equiv 1$ implies $x^{\frac{(n-1)}{p}} \equiv 1(\mathrm{mod} m)$, a contradiction. **Q.E.D.**

To show that the last two results are not vacuous, we have([6, theorem 111, p.86]):

Theorem 15 *If n is prime then there exists an $x \in \mathbb{Z}_n^*$ satisfying the conditions of the preceding lemma: $x^{n-1} \equiv 1(\mathrm{mod} n)$ and $x^m \not\equiv 1(\mathrm{mod} n)$ for all m that divides $n-1$.*

This result is equivalent to the statement that $\mathbb{Z}_n^*$ is a cyclic multiplicative group, with generator x. The exercises contain a proof. We may now show that *Primes* is in *NP*: on input n, we guess an $x \in \mathbb{Z}_n$ and a prime factorization of $n-1$ (represented as a set of prime numbers together their multiplicities). If $p_1, \ldots, p_k$ are the distinct guessed primes, note that $k \leq \log n$. We then check that each of the following holds:

(a) Verify that $x^{n-1} \equiv 1(\mathrm{mod} n)$.

(b) Verify that the product of the p_i's, taken to the power of their guessed multiplicities, is equal to $n-1$.

(c) Verify that $x^{\frac{n-1}{p_i}} \not\equiv 1(\mathrm{mod} n)$, for each i.

(d) Recursively verify that each of the p_i is a prime.

To verify (a) we must raise x to the $(n-1)$st power. Using the squaring technique, we need use only a linear number (i.e., $O(\log n)$) of multiplications of numbers whose magnitudes are each less than n (this is done by repeatedly reducing the intermediate results modulo n). Hence this takes at most cubic time. To verify (b), there are at most $\log n$ factors (counting multiplicities), and again cubic time suffices. Part (c) is similarly cubic time. Hence, if $t(n)$ is the time of this (non-deterministic) algorithm on input n, then

$$t(n) = O(\log^3 n) + \sum_{i=1}^{k} t(p_i)$$

It is easy to verify inductively that $t(n) = O(\log^4 n)$. Thus *Primes* is in *NP*.

3.6.2 Complexity of ILP

We begin by showing that ILP is *NP*-hard. This is quite easy, via a reduction from 3CNF: given a 3CNF formula F consisting of the clauses

$$C_1, C_2, \ldots, C_m$$

we introduce a set of inequalities. For each variable v, if v or its negation $\bar{v}$ occurs in F, we introduce the inequalities:

$$v \geq 0, \quad \bar{v} \geq 0, \quad v + \bar{v} = 1.$$

For each clause $C = \{\alpha, \beta, \gamma\}$, introduce

$$\alpha + \beta + \gamma \geq 1.$$

It is easy to write all these inequalities in the form $A\mathbf{x} \geq \mathbf{b}$ for a suitable matrix A and vector $\mathbf{b}$. Furthermore, this system of inequalities has an integer solution iff F is satisfiable.

The original proof[7] that ILP is in *NP* is by von zur Gathen and Sieveking[14]. To show that ILP is in *NP*, we want to guess an $\mathbf{x}$ and check if the system $A\mathbf{x} \geq \mathbf{b}$ holds on input $A, \mathbf{b}$. However we cannot do this checking in polynomial time if the smallest size solution $\mathbf{x}$ is too large. The basic issue is to show that if there is a solution then there is one of small size. In this context, the *size* of a number or a matrix is the number of bits sufficient to encode it in standard ways. More precisely, we define the size of a number n as logarithm of its magnitude $|n|$; the size of a $m \times n$ matrix is defined as mn plus the sum of the sizes of its entries; the size of the input to ILP is defined as the sum of the sizes of A and $\mathbf{b}$. The proof which takes up the remainder of this subsection requires some elementary linear algebra. The basis of our estimates is the following simple inequality:

Lemma 16 *Let B be an n square matrix whose entries each have size at most β. Then the determinant $det(B)$ is bounded by $n!\beta^n < s^{2s}$, where s is the size of B.*

This result is seen by summing up the $n!$ terms in the definition of a determinant. In the remainder of this subsection, except noted otherwise, the following notations will be fixed:

$A = (A_{i,j})$	:	an $m \times n$ matrix with integer entries with $m \geq n$.
$\mathbf{b} = (b_i)$	:	an m-vector with integer entries.
α	:	the maximum magnitude (not size) of entries in $A, \mathbf{b}$.
s	:	the size of the A plus the size of $\mathbf{b}$.
$\mathbf{a}_i$	:	the ith row of A.

[7]The present proof is partly based on notes by Ó'Dúnlaing. Kozen points out that the usual attribution of this result to [3] is incorrect. Borosh and Treybig showed a weaker version of the Key lemma above: namely, if $A\mathbf{x} = \mathbf{b}$ has a solution then it has a small solution. It does not appear that the Key lemma can be deduced from this.

Lemma 17 (Key lemma) *Let $A, \mathbf{b}$ be given as above. If there exists an integer solution $\mathbf{x}$ to the system $A\mathbf{x} \geq \mathbf{b}$ then there is a solution $\mathbf{x}$ each of whose entries has magnitude at most $2s^{7s}$.*

It is easy to see that this implies that ILP is in *NP*: First observe that the size s of the input $\langle A, \mathbf{b}\rangle$ satisfies

$$s \geq mn + \log \alpha.$$

On input $\langle A, \mathbf{b}\rangle$, guess some vector $\mathbf{x}$ each of whose entries has magnitude at most $2s^{7s}$. Since each entry has size $O(s^2)$, the guessing takes time $O(ns^2)$. Compute $A\mathbf{x}$ and compare to $\mathbf{b}$, using time $O(mn^2s^3) = O(s^5)$. Accept if and only if $A\mathbf{x} \geq \mathbf{b}$.

We prove some lemmas leading to the Key lemma. The kth *principal submatrix* of any matrix B is the $k \times k$ submatrix of B formed by the first k columns of the first k rows of B; let $B^{(k)}$ denote this submatrix.

Lemma 18 *Suppose the hth principal submatrix $A^{(h)}$ of A is non-singular. If $h < rank(A)$ then there exists an integer n-vector $\mathbf{z}$, $\mathbf{z} \neq \mathbf{0}$, such that*

(i) $\mathbf{a}_i\mathbf{z} = 0$ *for* $i = 1, \ldots, h$,

(ii) each of the first $h+1$ components of $\mathbf{z}$ has magnitude at most s^{2s}, but all the remaining components are zero.

Proof. If the $\mathbf{c}$ is the h-vector composed of the first h entries of the $(h+1)$st column of A then there is an h-vector $\mathbf{v} = (v_1, \ldots, v_h)$ such that $A^{(h)}\mathbf{v} = \mathbf{c}$. In fact, Cramer's rule tells us that the entries of $\mathbf{v}$ are given by $v_i = \pm C_i/\Delta$ where C_i, Δ are determinants of the $h \times h$ submatrices of the $h \times (h+1)$ matrix $(A^{(h)}|\mathbf{c})$. We define the required $\mathbf{z}$ by

$$z_i = \begin{cases} v_i\Delta = \pm C_i & \text{if } 1 \leq i \leq h \\ -\Delta & \text{if } i = h+1 \\ 0 & \text{otherwise.} \end{cases}$$

It is easy to see that $\mathbf{z}$ is a non-zero integer vector and $\mathbf{a}_i\mathbf{z} = \mathbf{0}$ for $i = 1, \ldots, h$. Each z_i has magnitude $\leq s^{2s}$. **Q.E.D.**

Lemma 19 *Assume $0 \leq h < rank(A)$. Suppose that the hth principal submatrix $A^{(h)}$ is non-singular and $A\mathbf{x} \geq \mathbf{b}$ has an integer solution $\mathbf{x}$ such that for $1 \leq i \leq h$, the ith row of A satisfies*

$$b_i \leq \mathbf{a}_i\mathbf{x} < b_i + s^{4s}. \tag{3.4}$$

Then there is a matrix A' obtained by permuting the rows and columns of A such that

(i) the $(h+1)$st principal submatrix $A'^{(h+1)}$ is non-singular, and

(ii) Let $\mathbf{b}' = (b'_1, \ldots, b'_m)$ *be the permutation of* $\mathbf{b}$ *corresponding to the permutation of the columns of* A *to derive* A'*. There is a solution* $\mathbf{x}'$ *to* $A'\mathbf{x}' \geq \mathbf{b}'$ *such that for all* $1 \leq i \leq h+1$*, the following inequality holds:*

$$b'_i \leq \mathbf{a}'_i \mathbf{x}' < b'_i + s^{4s}.$$

Proof. In the statement of this lemma, the principal submatrix $A^{(0)}$ is conventionally taken to be non-singular. By permuting the columns of A if necessary, we can assume that the first $h+1$ columns of A are linearly independent. By the previous lemma, there is a non-zero vector $\mathbf{z}$ such that (i) $\mathbf{a}_i \mathbf{z} = 0$ for all $i = 1, \ldots, h$, and (ii) the first $h+1$ components of $\mathbf{z}$ are at most s^{2s} and the remaining components are 0. Since the first $h+1$ columns of A are linearly independent, there exists $i, (h < i \leq n)$, such that $\mathbf{a}_i \mathbf{z} \neq 0$. Assume that there exists some i such that $\mathbf{a}_i \mathbf{z}$ is positive (the case $\mathbf{a}_i \mathbf{z} < 0$ is similar), and let $J \subseteq \{h+1, \ldots, m-1, m\}$ consist of all those j satisfying

$$\mathbf{a}_j \mathbf{z} > 0.$$

Let δ be the smallest non-negative integer such that if $\mathbf{x}'' = x - \delta \mathbf{z}$ then for some $j_0 \in J$, $b_{j_0} \leq \mathbf{a}_{j_0} \mathbf{x}'' < b_{j_0} + s^{4s}$. We claim that

(a) $\mathbf{a}_i \mathbf{x}'' = \mathbf{a}_i \mathbf{x}$ for $i = 1, \ldots, h$, and

(b) $\mathbf{a}_i \mathbf{x}'' \geq b_i$ for $i = h+1, \ldots, m$.

It is easy to see that (a) follows from our choice of $\mathbf{z}$. As for (b), the case $\delta = 0$ is trivial so assume $\delta > 0$. When $i \notin J$, (b) follows from the fact that $\mathbf{a}_i \mathbf{x}'' \geq \mathbf{a}_i \mathbf{x}$. When $i \in J$, our choice of δ implies that

$$b_i + s^{4s} \leq \mathbf{a}_i \mathbf{x} - (\delta - 1)\mathbf{a}_i \mathbf{z}$$

But $\mathbf{a}_i \mathbf{z} \leq n\alpha s^{2s} \leq s^{4s}$ implies $\mathbf{a}_i \mathbf{x}'' \geq b_i$. This proves (b). By exchanging a pair of rows of $(A|\mathbf{b}|\mathbf{x}'')$ to make the j_0th row of A the new $(h+1)$st row, we get the desired $(A'|\mathbf{b}'|\mathbf{x}')$. **Q.E.D.**

We are finally ready to prove the Key lemma. Let $\mathbf{x} = (x_1, \ldots, x_n)$ be an integer solution to $A\mathbf{x} \geq \mathbf{b}$. For each i, clearly $x_i \geq 0$ or $-x_i \geq 0$. Hence there exists an n square matrix M whose entries are all zero except for the diagonal elements which are all $+1$ or -1 such that $M\mathbf{x} \geq \mathbf{0}$. Let A' be the $(m+n) \times n$ matrix

$$\left(\frac{A}{M}\right)$$

and $\mathbf{b}'$ be the $(m+n)$-vector obtained by appending zeroes to $\mathbf{b}$. Thus $A'\mathbf{x} \geq \mathbf{b}'$ has a solution. But observe that A' has at least as many rows as columns and it has full rank (equal to the number n of columns). Hence the previous lemma is applicable to $A', \mathbf{b}'$ successively, for $h = 0, 1, \ldots, n-1$. This shows the existence of an integer

solution $\mathbf{x}'$ to $A''\mathbf{x}' \geq \mathbf{b}'$, where A'' is a row and column permutation of A' and the individual components of $A''\mathbf{x}'$ are bounded by $s + s^{4s} \leq 2s^{4s}$. Since $rank(A'') = n$, there is an $n \times n$ nonsingular submatrix B of A''. If $B\mathbf{x}' = \mathbf{c}$ then $\mathbf{x}' = B^{-1}\mathbf{c}$.

We now bound the size of the entries of $\mathbf{x}'$. We claim that each entry of B^{-1} has magnitude at most s^{2s}: to see this, each entry of B^{-1} has the form $B_{i,j}/\Delta$ where Δ is the determinant of B and $B_{i,j}$ is the co-factor of the (i,j)th entry of B. Since a co-factor is, up to its sign, equal to a $(n-1) \times (n-1)$ subdeterminant of B, our bound on determinants (lemma 16) provides the claimed upper bound of s^{2s}. Now each entry of $\mathbf{c}$ has magnitude at most $2s^{4s}$. Thus each entry of $\mathbf{x}' = B^{-1}\mathbf{c}$ has magnitude at most $2ns^{6s} < 2s^{7s}$. Finally observe that some permutation of $\mathbf{x}'$ corresponds to a solution to $A\mathbf{x} \geq \mathbf{b}$. This concludes our proof of the Key lemma.

3.7 Final Remarks

This chapter not only introduces the class *NP* but opens the door to the core activities in Complexity Theory: many questions that researchers ask can be traced back to the desire to understand the relationship between *NP* (which encompasses many problems of interest) and *P* (which constitutes the feasably solvable problems in the fundamental mode). For instance, the concepts of reducibilities and complete languages will be extended and serve as subject matters for the next two chapters.

We began this chapter with the traveling salesman problems, TSO and TSD. Now we can easily show the recognition problem TSD is *NP*-complete. It is enough to show how to transform any graph $G = (V, E)$ into an input $\langle D, b\rangle$ for the TSD problem such that G has a Hamiltonian circuit iff $\langle D, b\rangle \in$ TSD. Assume that the vertices of G are $1, 2, \ldots, n$. The $n \times n$ matrix $D = d_{i,j}$ is defined by $d_{i,j} = 1$ if $\{i, j\} \in E$; otherwise $d_{i,j} = 2$. It is easy to see that D has a tour of length $b = n$ iff G has a Hamiltonian circuit. The simplicity of this reduction illustrates an earlier remark that showing a problem L to be *NP*-hard can be facilitated if we have a closely related problem L' already known to be *NP*-hard.

Arithmetic versus Bit Models of complexity In computational problems involving numbers, typically problems in computational geometry, we have two natural models of complexity. One is the Turing machine model where all numbers must ultimately be encoded as finite bit-strings; the other is where we view each number as elementary objects and we count only the number of arithmetic operations. These two complexity models are distinguished by calling them the *bit-model* and *arithmetic-model*, respectively. We warn that despite its name, the arithmetic-model may sometimes allow non-arithmetic basic operations such as extracting radicals.

Complexity in the two models are often closely related. However, we point out that these models may be independent from a complexity viewpoint. More precisely, there may be problems requiring non-polynomial time in one model but which uses only polynomial time in the other. For instance, the linear programming problem

is polynomial time under the bit-model, but unknown to be polynomial time in the arithmetic-model. On the other hand, the problem of shortest paths between two points amidst polygonal obstacles is polynomial time in the arithmetic-model, but not known to be in polynomial time in the bit-model.

Exercises

[3.1] Prove propositions B and C.

[3.2] Construct the Turing transducer N that computes the transformation in the proof of Cook's theorem.

[3.3] * Convert the satisfying assignment problem into an optimization problem by asking for an assignment which satisfies the maximum number of clauses in a given CNF formula F. Is this problem is polynomially equivalent to SAT?

[3.4] Show that 2SAT, the set of satisfiable CNF formulas with exactly two literals per clause can be recognized in polynomial time.

[3.5] Let X be a set of n Boolean variables and k be a positive integer less than n. Construct a CNF formula $T_k(X, Y)$ containing the variables in the set $X \cup Y$ where Y is an auxiliary set of variables depending on your construction satisfying the following property: an assignment $I : X \cup Y \to \{0, 1\}$ satisfies $T_k(X)$ iff I makes at least k variables in X true. Let us call $T_k(X, Y)$ the *k-threshold* formula. Your formula should have size polynomial in n. What is the smallest size you can achieve for this formula? *Hint:* it is easy to construct a formula where Y has kn variables. Can you do better?

[3.6] Show a log-space reduction of the following problems to SAT. Note that there are well-known deterministic polynomial time algorithms for these problems. However, you should give direct solutions, i.e., without going through Cook's theorem or appeal to the known deterministic polynomial time algorithms. Hopefully, your reductions to SAT here are less expensive than solving the problems directly! So your goal should be to give as 'efficient' a reduction as possible. In particular, none of your transformations should take more than quadratic time (you should do better for the sorting problem).

(i) Graph matching: input is a pair $\langle G, k \rangle$ where k is a positive integer and G is n by n Boolean matrix representing an undirected graph; the desired property is that G has a matching of cardinality k. The threshold formulas in the previous problem are useful here.

(ii) Sorting: input is a pair $\langle L, L' \rangle$ of lists where each list consists of a sequence of the form

$$\#k_1\#d_1\#k_2\#d_2\#\cdots\#k_n\#d_n\#$$

where k_i and d_i are binary strings. Regarding k_i as the 'key' and d_i as the corresponding 'data', this language corresponds to the problem of

sorting the data items according to their key values. The 'input' list L is arbitrary but the 'output' list L' is the sorted version of L: L' ought to contain precisely the same (key, data) pairs as L, and the keys in L' are in non-decreasing order.

(iii) Non-planarity testing: input is a graph G and the property is that G is non-planar. (Use the two forbidden subgraphs characterization of Kuratowski.)

(iv) Network flow: input is $\langle G, s, t, C, k\rangle$ where G is a directed graph, s and t are distinct nodes of G (called the source and sink), C assigns non-negative integer values to the nodes of G ($C(u)$ is the *capacity* of node u), and $k \geq 0$ is an integer. The desired property is that the maximum value of a flow from s to t is k. Note that integers are represented in binary, and C can be encoded by a list of pairs of the form $(u, C(u))$ for each node u. (Use the max-flow-min-cut theorem and the fact that we can restrict flows to be integral.)

[3.5] Recall that a 3DNF formula has the form

$$\bigvee_{i=1}^{m} \bigwedge_{j=1}^{3} u_{i,j}$$

where $u_{i,j}$ are literals. We also let 3DNF-SAT denote the set of satisfiable (encoded) 3DNF formulas. Show that 3DNF-SAT can be recognized in polynomial time.

[3.6] (Bauer-Brand-Fisher-Meyer-Paterson) Let F be a Boolean formula. Show a systematic transformation of F to another 3CNF G such that they are co-satisfiable (i.e., both are satisfiable or both are not). Furthermore, $|G| = O(|F|)$ where $|G|$ denotes the size of the formula G. *Hint:* For each subformula H of F, introduce an additional variable α_H. We want to ensure that any assignment I to the variables of H assigns to α_H the value of the subformula H under I (i.e., if I makes H false then $I(\alpha_H) = 0$). For instance, $H = H_1 \vee H_2$. Then we introduce clauses that enforce the equivalence $\alpha_H \equiv \alpha_{H_1} \vee \alpha_{H_2}$.

[3.7] The *parity* function on n variables is $x_1 \oplus x_2 \oplus \cdots \oplus x_n$ where $\oplus$ is exclusive-or. Thus the function is 1 precisely when an odd number of its inputs are 1. Show that the smallest CNF formula equivalent to the parity function is exponential in n. (Note that this shows that we could not strengthen the previous exercise such that F and G become equivalent rather than just co-satisfiable.) *Hint:* Consider the following CNF formula $\bigwedge_{i=1}^{k} \bigvee J_i$ where each J_i is a set of literals over $x_1, \ldots, x_n$. Show that if this is the parity function then $|J_i| = n$.

[3.8] Show that the problem of graph coloring is *NP*-complete.

[3.9] Show that the problem of deciding if a graph contains a pre-specified set of vertex-disjoint paths is *NP*-complete. The input is an undirected graph together with some set $\{(u_i, v_i) : i = 1, \ldots, k\}$ of pairs of vertices. The required property is that there are k pairwise vertex-disjoint paths connecting the given pairs of vertices. *Hint:* Reduce from 3SAT.

[3.10] (Burr) A graph $G = (V, E)$ is *NMT-colorable* ('no monochromatic triangle') if it can be 2-colored such that no triangle (3-cycle) has the same color. Show that the problem of recognizing NMT-colorable graphs is *NP*-complete. *Hint*: Construct three gadgets (I), (II) and (III) with the following properties. Gadget (I) has two distinguished nodes such that any NMT-coloring of (I) must give them the same color. Gadget (II) has two distinguished nodes such that any NMT-coloring of (II) must give them distinct colors. Gadget (III) has four distinguished nodes A, B, C, D such that in any NMT-coloring of this gadget must make at least one of A, B or C the same color as D. When the various gadgets are strung together, the D node of all copies of Gadget (III) should be common.

[3.11] Consider the regular expressions over some alphabet Σ involving the operators of concatenation ($\cdot$), union ($+$), and Kleene-star ($*$). Each regular expression α denotes a subset of Σ^*. We now consider a class of modified regular expressions in which Kleener-star is not used but where we allow intersection ($\cap$). Show that the problem of recognizing those modified regular expressions α where $L(\alpha) \neq \Sigma^*$ is *NP*-hard. *Hint*: imitate Cook's theorem, but use such expressions to denote those strings that *do not* represent accepting computations.

[3.12] (Cook) Show that the problem of recognizing input pairs $\langle G, G' \rangle$ of undirected graphs with the property that G is isomorphic to a subgraph of G' is *NP*-complete.

[3.13] * (Stockmeyer) Show that the problem of recognizing those graphs G that are planar and 3-colorable is *NP*-complete. (Note: we can assume that the input graph is planar because planarity testing can be done in linear time.) *Hint:* Reduce 3SAT to the present problem.

[3.14] * (Kozen) Show that the problem of recognizing valid sentences of the first-order predicate calculus with the equality symbol but *without negation* is *NP*-complete. The language contains the usual logical symbols ($\wedge, \vee, \forall, \exists$) sans negation ($\neg$), individual variables x_i, relation symbols R_i^m and function symbols F_i^m (where $m \geq 0$ denotes the arity of the relation or function symbol, and for $i = 0, 1, \ldots$). We further assume that R_0^m is the standard

equality symbol '='. We are interested in valid sentences, i.e., closed formulas that are true in all models under all interpretations (but equality is standard). Called this the *validity problem for positive first-order logic.*

[3.15] (a) Show that the *tiling problem* is *NP*-complete. The input to this problem has the form $\langle n, k, S\rangle$ where n and k are positive integers and S is a finite set of 'tile patterns'. Each tile pattern in S is a sequence of the form $p = \langle c_1, c_2, c_3, c_4\rangle$ where $c_i \in \{1, \ldots, k\}$. Any oriented unit square whose top, bottom, left and right edges are colored with the colors c_1, c_2, c_3 and c_4, respectively, is said to have pattern p. The problem is to decide whether it is possible to cover an n by n square area using n^2 unit tiles satisfying the following condition: (i) each unit tile has a pattern from S, and (ii) for any two tiles that abut, their two adjacent edges have the same color. *Hint:* simulate a Turing machine computation using tiles.

(b) For fixed k, let the k-tiling problem be the restriction of the tiling problem to k colors. What is the smallest k for which you can show that the k-tiling problem remains *NP*-complete? What is the largest k for which you can show that the problem is in P?

[3.16] (Karp-Held) Give a dynamic programming solution to TSO which has a running time of $O(n2^n)$. Hint: Let $S \subseteq \{2, 3, \ldots, n\}$ be a subset of the n cities, and $k \in S$. Let $C(S, k)$ denote the minimum cost of a route which starts at city 1, visiting all the cities in S exactly once, terminating at k. Give an expression for $C(S, k)$ in terms of $C(T, i)$ for all T where $T \subseteq S$ and $|T| = |S| - 1$.

[3.17] (a) Show that if p is prime and x is an integer, $1 \leq x < p$, then p divides $\left(\begin{array}{c} p \\ x \end{array}\right)$ where $\left(\begin{array}{c} p \\ x \end{array}\right) = \frac{p(p-1)\cdots(p-x+1)}{x!}$.

(b) Conclude (using induction on k) that if p is prime then for all $x_1, \ldots, x_k$,

$$\left(\sum_{i=1}^{k} x_i\right)^p \equiv \left(\sum_{i=1}^{k} x_i^p\right) \pmod{p}.$$

[3.18] Prove Fermat's little theorem. Hint: $x = \sum_{j=i}^{x} y_i$ where each $y_i = 1$ and use the previous exercise.

[3.19] Prove that if G is an Abelian group, and s and t are elements of G of orders m and n (respectively) then G has an element of order $\mathrm{lcm}(m, n)$. Hint: by replacing s with $s^{\gcd(m,n)}$, we may assume that $\mathrm{lcm}(m, n) = mn$. What is the order of st?

[3.20] Prove that $\mathbf{Z}_p^*$ is a cyclic group where p is prime. *Hint:* Let n be the maximal order of an element of $\mathbf{Z}_p^*$. Conclude from the previous exercise that $x^n - 1 \equiv 0 (\mathrm{mod} p)$ has $p - 1$ distinct solutions.

[3.21] We want to demonstrate that if we allow the output tape of transducers to be a 2-way (but still read-only) tape then we have a more powerful model. Recall the non-regular language L_0 in the last section of chapter 2. Consider the problem where, given an integer n in binary, we want to output the string $\bar{1}\#\bar{2}\#\cdots\#\bar{n}$ in the language L_0. The input size is $O(\log n)$ and the output size is $O(n \log n)$. (a) Show that if the output tape is 1-way as in our standard definition of transducers, then the space used is linear, i.e. $O(\log n)$. (b) Show that if we have a 2-way read-only output tape then logarithmic space (i.e. $O(\log \log n)$) suffices.

[3.22] * (Berman) Show that if there is a tally language $L \subseteq \{1\}^*$ that is complete for NP (under $\leq_m^P$) then P=NP.

[3.23] * The *Euclidean Travelling Salesman's Problem* (ETS) is the following: given a set $\{p_1, \ldots, p_n\}$ of points in the plane, find the shortest tour that connects all these points. It is assumed that the points have integer coordinates and the distance between any pair of points p, q is the usual Euclidean distance:

$$d(p,q) = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}$$

where $p = (p_x, p_y), q = (q_x, q_y)$. ETS is known to be NP-hard, but why is it not obviously NP-complete? Give a single-exponential time upper bound on the complexity of ETS.
Hint: Reduce the problem to comparing a sum of square-roots to zero.

Appendix A

Propositional Logic

A *Boolean variable* is a variable that assumes a value of '1' or '0' (alternatively '*true*' or '*false*'). If x is a Boolean variable, then its *negation* is denoted either $\bar{x}$ or $\neg x$. A *literal* is either a Boolean variable or the negation of one. A *(Boolean) formula* is either a variable or recursively has one of the three forms:

conjunction: $(\phi \wedge \psi)$

disjunction: $(\phi \vee \psi)$

negation: $(\neg\phi)$

where ϕ and ψ are Boolean formulas. As an example of a Boolean formula, we have

$$((x_1 \vee x_2) \vee x_3) \wedge (((\bar{x}_1 \vee x_2) \vee x_2) \wedge (\bar{x}_2 \vee \bar{x}_3)). \tag{A.1}$$

It is important to realize that formulas are syntactic objects, i.e., a sequence of marks laid out according to the preceding rules, where the marks comes from $\vee, \wedge, \neg, (,)$, and a unique mark for each variable x_i, $i = 1, 2, 3, \ldots$. When encoding formulas to apply our theory, we cannot assume an infinite number of marks. Hence we must encode each x_i by a sequence from a (finite) alphabet Σ. A simple choice, and the one assumed in this book, is the following: let

$$\Sigma = \{\vee, \wedge, \neg, (,), x, 0, 1\}$$

and each x_i is encoded by the symbol 'x' followed by $b_0 b_1 \cdots b_m \in \{0,1\}^*$ where $b_0 \cdots b_m$ is the integer i in binary. Using this convention, and assuming that no variable x_{i+1} is used unless x_i is also used, it is easy to see that the *size* of a formula ϕ is $O(k + m \log m)$ where k is the number of Boolean operator occurrences and m is the number of variable occurrences.

A variable x *occurs* in a formula F if either x or its negation syntactically appears in F. If all the variables occurring in F are among $x_1, x_2, \ldots, x_n$, we indicate this

by writing $F(x_1, x_2, \ldots, x_n)$. In this notation, some (possibly all) x_i may not occur in F. An *assignment* to a set X of Boolean variables is a function $I : X \to \{0, 1\}$. If X contains all the variables occurring in F, and I is an assignment to X then I (by induction on the size of F) assigns a Boolean value $I(F)$ to F as follows: if F is a variable x, $I(F) = I(x)$; otherwise:

if F is the negation $(\neg\phi)$ then $I(F) = 1 - I(\phi)$;
if F is the conjunct $(\phi \wedge \psi)$ then $I(F) = \min\{I(\phi), I(\psi)\}$;
if F is the disjunct $(\phi \vee \psi)$ then $I(F) = \max\{I(\phi), I(\psi)\}$.

We say I *satisfies* F if $I(F) = 1$. A formula F is *satisfiable* if there exists an assignment which satisfies it. Thus the formula in equation (A.1) is satisfied by I with $I(x_1) = I(x_2) = 1, I(x_3) = 0$.

Two formulas ϕ and ψ are *equivalent*, written $\phi \equiv \psi$, if for any assignment I to the variables occurring in both formulas, $I(\phi) = I(\psi)$. The formulas are *co-satisfiable* if they are either both satisfiable or both unsatisfiable.

As seen in the example (A.1), parentheses in formulas get to be tedious. We can avoid these parenthesis by using the properties that conjunctions and disjunctions are associative, i.e.,

$$\phi_1 \wedge (\phi_2 \wedge \phi_3) \equiv (\phi_1 \wedge \phi_2) \wedge \phi_3; \quad \phi_1 \vee (\phi_2 \vee \phi_3) \equiv (\phi_1 \vee \phi_2) \vee \phi_3,$$

and commutative, i.e.,

$$\phi_1 \wedge \phi_2 \equiv \phi_2 \wedge \phi_1; \quad \phi_1 \vee \phi_2 \equiv \phi_2 \vee \phi_1,$$

and using the fact that

$$\phi \vee \phi \equiv \phi; \quad \phi \wedge \phi \equiv \phi.$$

Thus (A.1) can be simply written as

$$(x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee x_2) \wedge (\bar{x}_2 \vee \bar{x}_3). \tag{A.2}$$

If $S = \{u_1, \ldots, u_n\}$ is a set of literals, we also write

$$\bigvee S \text{ or } \bigvee_{u \in S} u \text{ or } \bigvee_{i=1}^{n} u_i$$

for the conjunction of the literals in S. We use $\bigwedge$ in a similar way for conjunctions. A formula that is a conjunction of disjunctions of literals is said to be in *conjunctive normal form* (CNF). The example (A.2) is such a formula. The *disjunctive normal form* (DNF) is similarly defined.

From the properties noted above, formulas in CNF can be given a convenient alternative form: A CNF formula *in clause form* is a set of clauses, where a *clause* is a set of literals. There is an obvious way to transform a CNF formula in the usual form to one in clause form; for example, the formula (A.2) in the clause form is

$$\{\{x_1, x_2, x_3\}, \{\bar{x}_1, x_2\}, \{\bar{x}_2, \bar{x}_3\}\}.$$

Note that since clauses are defined as sets, repeated literals are removed, as in the case of the second clause above. Satisfiability of CNF formulas in the clause form is particularly simple: let X be the set of variables occurring in the CNF formula F and let I be an assignment to X. Then I *satisfies* a clause C in F iff for some literal u in C we have $I(u) = 1$. I *satisfies* F iff I satisfies each clause in F. For instance, the following CNF formula

$$\{\{x_1, \bar{x}_2\}, \{\bar{x}_1, x_2\}, \{x_1\}, \{\bar{x}_2\}\}$$

is unsatisfiable.

Bibliography

[1] M. Akgül. *Topics in relaxation and ellipsoidal methods*. Pitman Advanced Publishing Program, Boston-London-Melbourne, 1981. (U. Waterloo PhD Thesis).

[2] Karl Heinz Borgwardt. *The Simplex Method: a probabilistic analysis*. Springer-Verlag, 1987.

[3] I. Borosh and L. B. Treybig. Bounds on positive integral solutions of linear Diophantine equations. *Proc. AMS*, 55:299–304, 1976.

[4] Steven A. Cook. The complexity of theorem-proving procedures. *3rd Proc. ACM Symp. Theory of Comp. Sci.*, pages 151–158, 1971.

[5] Michael R. Garey and David S. Johnson. *Computers and Intractibility: A Guide to the Theory of NP-completeness*. Freeman, New York, 1979.

[6] G. H. Hardy and E. M. Wright. *An Introduction to the Theory of Numbers*. Oxford University Press, London, 1938.

[7] Richard M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–104. Plenum Press, New York, 1972.

[8] L. G. Khachian. A polynomial algorithm for linear programming. *Doklady Akad. Nauk USSR*, 244:5:1093–96, 1979. (tr. *Soviet Math. Doklady 20* 191-194).

[9] V. Klee and G. J. Minty. How good is the simplex algorithm? In O. Shisha, editor, *Inequalities III*, pages 159–175. Academic Press, 1972.

[10] Thomas S. Kuhn. *The structure of scientific revolutions*. Chicago Univ. Press, 1970.

[11] L. A. Levin. Universal sorting problems. *Problemi Peredachi Informatsii*, 9:3:265–266, 1973.

[12] C. H. Papadimitriou. On the complexity of integer programming. *Journal of the ACM*, 28:765–768, 1981.

[13] Vaughn R. Pratt. Every prime has a succinct certificate. *SIAM J. Computing*, 4:214–220, 1975.

[14] J. von zur Gathen and M. Sieveking. A bound on solutions of linear integer equalities and inequalities. *Proc. AMS*, 72:155–158, 1978.

Contents

Chapter 4

Reducibilities

March 1, 1999

4.1 Inclusion Questions

Many open problems of Complexity Theory are *inclusion questions* of the form:

$$\text{Is } K \text{ included in } K' \text{ ?} \tag{4.1}$$

Here K, K' are two classes of languages. The oldest such inclusion (or containment) question, dating back to 1960s, is the linear bounded automata (LBA) question: Is $NSPACE(n)$ included in $DSPACE(n)$? As explained in §2.9, these two classes are also known as LBA and $DLBA$, respectively. The P versus NP problem is another instance of the inclusion question. In chapter 3, we introduced Karp reducibility and showed its value for studying this question. In general, reducibilities will be the key tool for investigating inclusion questions. We begin now with a very general notion of reducibility.

Definition 1 A *reducibility* is a reflexive binary relation on languages. Let $\leq$ denote a reducibility. We use an infix notation to reducibility relations: if (L, L') belongs to the relation $\leq$, we write $L \leq L'$. A reducibility is *transitive* if it is transitive as a binary relation; otherwise it is *intransitive*. ■

In the literature, intransitive reducibilities are sometimes called "semi-reducibilities" and "reducibilities" are reserved for transitive ones. Admittedly, our use of "$\leq$" to denote semi-reducibilities can be confusing, since the inequality symbol suggests transitivity. Intransitive reducibilities typically arise when the reducibility concept is defined using nondeterministic machines (see §9).

We say L is $\leq$-*reducible* to L' if the relation $L \leq L'$ holds. Two languages L and L' are $\leq$-*comparable* if either $L \leq L'$ or $L' \leq L$; otherwise they are $\leq$-*incomparable*.

We also write $L \not\leq L'$ if $L \leq L'$ does not hold; and write $L < L'$ if $L \leq L'$ and $L' \not\leq L$.

Next assume that $\leq$ is transitive. If $L \leq L'$ and $L' \leq L$ then we say L and L' are $\leq$-*equivalent*. The $\leq$-*degree* of L is the class of languages $\leq$-equivalent to L.

The following generalizes some concepts from chapter 3.

Definition 2 Let K be a class of languages and $\leq$ a reducibility. Then L is K-*hard* (*under* $\leq$) if for every L' in K, L' is $\leq$-reducible to L. L is K-*complete* (*under* $\leq$) if L is K-hard and L is in K. K is *closed under* $\leq$-reducibility if for any L and L', we have that $L \leq L'$ and $L' \in K$ implies $L \in K$. ■

The importance of complete languages as a tool for studying the question (4.1) comes from the following easy generalization of the corollary to lemma 2 in chapter 3.

Lemma 1 (The Basic Inclusion Lemma) *Let K and K' be language classes and $\leq$ a reducibility. Assume*
(a) K' is closed under $\leq$-reducibility, and
(b) L is a K-complete language (under $\leq$).
Then

$$K \subseteq K' \iff L \in K'.$$

Under hypotheses (a) and (b) of this lemma, the inclusion question (4.1) is equivalent to the membership of a single language in K'. Thus a K-complete language serves as a representative of the entire class K. This lemma is the basis of most applications of reducibilities to the inclusion questions.

Choice of reducibilities (I). Given that we want to study a particular inclusion question, whether $K \subseteq K'$, within the framework of this lemma, how do we choose the reducibility $\leq$? One desirable property is that $\leq$ be transitive. Then, we can show a language $L \in K$ to be K-complete under $\leq$ simply by showing a known K-complete language L_0 is $\leq$-reducible to L. This tack is well-exploited in the field of *NP*-completeness theory as described in chapter 3. More on the choice of reducibilities in the next section. ■

We say a class K is *closed under complementation* if $K = co\text{-}K$. Observe that the question of closure under complement is equivalent to a special kind of inclusion question since

$$K = co\text{-}K \iff K \subseteq co\text{-}K.$$

Of the many open inclusion questions, there is often a weaker conjecture involving the closure of a class under complementation. For instance, it is not known if *NP* is closed under complementation. If $P = NP$ then *NP* must be closed under complementation. The converse is not known to be true. Hence it is conceivably easier to prove that $NP = co\text{-}NP$ than to prove that $NP = P$. Another example is the *LBA*-question. While the *LBA*-question remains open, a general result about

non-deterministic space classes in §2.9 implies that LBA is closed under complementation.

We often wish to compare two reducibilities $\leq_1$ and $\leq_2$. We say $\leq_1$ is *as strong as* $\leq_2$ if $L \leq_2 L'$ implies $L \leq_1 L'$ for all L, L'. It is important to note the direction of implication in this definition since the literature sometimes gives it the opposite sense.[1] If $\leq_1$ is as strong as than $\leq_2$, and but not vice-versa, then $\leq_1$ is *stronger than* $\leq_2$. Thus, the stronger reducibility strictly contains the weaker one, if we regard a reducibility as a set of ordered pairs of languages.

As a historical note, reducibilities are extensively used in recursive function theory. At the subrecursive level, Cook in the proof of his theorem, is an early example of defining a reducibility that takes complexity into account. Meyer and McCreight [13] used such reducibilities in relating complexity of problems. Following Meyer, the reducibilities that take complexity into account may be generically called 'efficient reducibilities'. Systematic study of efficient reducibilities was first undertaken by Ladner, Lynch, and Selman [10, 8, 9]. See also [3, 18].

Karp reducibility is denoted $\leq_m^P$ (§3.3). The superscript P here indicates polynomial-time; similarly, the subscript m indicates Karp reducibility is a form of "many-one reducibility". Generally speaking, efficient reducibilities are classified along two lines, and this is reflected in our notational scheme

$$\leq_\tau^\chi$$

for reducibilities: the symbol τ indicates the type of the reducibility, and the symbol χ indicates the complexity characteristics (see §4.4).

Convention. The following abuse of notation is often convenient. If M (resp. T) is an acceptor (resp. transducer) we sometimes use it to also denote the language (resp. transformation) defined by the indicated machine. Thus if x, y are words, then we may say '$x \in$ M' or '$y = \mathrm{T}(x)$' with the obvious meaning.

4.2 Many-One Reducibility

Many-one reducibility (§3.3) is now treated in a more general setting. The terminology begs the question: is there a one-one or a many-many reducibility? In recursive function theory, and occasionally in complexity theory (e.g. [19]), one-one reducibility has been studied. Nondeterministic many-one reducibility (to be introduced in section 9) may be regarded as a many-many reducibility.

[1] Our rationale is that the strength of the reducibility should be in direct proportion to the power of the machines used in defining it: thus the polynomial time reducibility ought to be as strong as one defined by logarithmic space bounds. The opposite sense is reasonable if we think of "logical strength of implication". Kenneth Regan suggests the term "finer" which seems avoid all ambiguity.

Definition 3 Let Σ, Γ be alphabets, and Φ be a family of transformations. We say a language (Σ, L) is *many-one reducible to* another language (Γ, L') *via* Φ if there exists a transformation $t : \Sigma^* \to \Gamma^*$ in Φ such that for all $x \in \Sigma^*$,

$$x \in L \iff t(x) \in L'.$$

We write $L \leq_m^\Phi L'$ in this case. ■

For any alphabet Σ, the identity transformation is $t : \Sigma^* \to \Sigma^*$ with $t(x) = x$ for all x. The following proposition is immediate.

Lemma 2

(i) *If Φ contains all identity transformations then $\leq_m^\Phi$ is a reducibility.*

(ii) *If Φ is, in addition, closed under functional composition then $\leq_m^\Phi$ is a transitive reducibility.*

(iii) *If Φ, Ψ are families of transformations where $\Phi \subseteq \Psi$ then $\leq_m^\Psi$ is as strong as $\leq_m^\Phi$.*

If Φ is understood or immaterial, we shall simply write '$\leq_m$' instead of '$\leq_m^\Phi$'. Recall that a language L is *trivial* if L or *co-*L is the empty set. It is easy to see that no non-trivial language is many-one reducible to a trivial language and conversely, a trivial language is many-one reducible to any non-trivial one via some constant transformation (i.e., $t(x) = t(y)$ for all x, y). To avoid these special cases, we tacitly restrict all discussion to only non-trivial languages when discussing many-one reducibilities.

In Karp reducibility, we considered the family $\Phi = \mathbf{P}$ of transformations computed by deterministic polynomial-time transducers.[2] We now consider another important family, denoted **DLOG**, consisting of all transformations computed by deterministic transducers running in logarithmic space. The corresponding reducibility is denoted $\leq_m^L$ and called *log-space many-one reducibility*. The proof that $DSPACE(s) \subseteq DTIME(n \cdot O(1)^{s(n)})$ in chapter 2 (§6) can easily be adapted to show that

$$\mathbf{DLOG} \subseteq \mathbf{P}\ .$$

This implies that $\leq_m^P$ is as strong as $\leq_m^L$.

Choice of reducibilities (II). An important instance of (4.1) is whether

$$NLOG \subseteq DLOG. \tag{4.2}$$

Assuming that $P \neq DLOG$ (as is generally conjectured), the class $DLOG$ is not closed under $\leq_m^P$ (see Exercise). Thus hypothesis (a) of the Basic Inclusion Lemma

[2] We use bold letters to denote families of transformations.

fails and we cannot use $\leq_m^P$-reducibility to investigate question (4.2). We say Karp reducibility is too strong for distinguishing *DLOG* from *NLOG*. More generally, hypothesis (a) places an upper limit on the power of the reducibility used in studying inclusion questions. On the other hand, it is easy to see that *DLOG* is closed under $\leq_m^L$, and chapter 5 will show that *NLOG* has complete languages under $\leq_m^L$. This is one of the motivations for introducing $\leq_m^L$. ■

We now show that $\leq_m^L$ is a transitive reducibility. This follows immediately from the following theorem of Jones [7] :

Theorem 3 *The family* **DLOG** *of log-space transformations is closed under functional composition.*

Proof. Let $\Sigma_0, \Sigma_1, \Sigma_2$ be alphabets. For $i = 1, 2$, let $t_i : \Sigma_{i-1}^* \to \Sigma_i^*$ be computed by the transducer N_i that runs in space $\log n$. We shall construct a 2-tape transducer M that computes $t_0 : \Sigma_0^* \to \Sigma_2^*$ that is the functional composition of t_1 and t_2. To motivate the present proof, briefly recall the proof that $\leq_m^P$ is a transitive reducibility. A direct adaptation of that proof to the present problem would yield a machine M′ which on input x, simulates N_1 to produce $t_1(x)$ and then simulates N_2 on $t_1(x)$. In general such an M′ uses more than logarithmic space since $|t_1(x)|$ may have length that is polynomial in $|x|$. To ensure that only logarithmic space is used, we shall avoid storing $t_1(x)$, but rather recompute the symbols in $t_1(x)$ as often as needed.

By our convention on transducers, tape 1 of M is the output tape. Tape 2 has four tracks used as follows:

Track 1:	Work-space of N_1 on input x.
Track 2:	Work-space of N_2 on input $t_1(x)$.
Track 3:	An integer indicating the position of the input head of N_2 on the input $t_1(x)$.
Track 4:	Counter used when simulating N_1.

Initially, track 3 contains the integer 1 indicating that the input head of N_2 is scanning the first symbol of $t_1(x)$. In general, if track 3 contains an integer i, M will call a subroutine R that will determine the ith symbol of $t_1(x)$. This is done as follows: R initializes the counter on track 4 to 0 and begins to simulate N_1 using track 1 as its work-tape. Each time N_1 produces a new output symbol, R increments the count on track 4 and checks if the new count equals the integer i on track 3; if so, R can immediately return with the ith symbol of $t_1(x)$. The operations of M are essentially driven by a direct simulation of N_2: to simulate a step of N_2, M first calls subroutine R to determine the symbol in $t_1(x)$ currently scanned by the input head of N_2. It can then update the work-tape of N_2 represented on track 2, and increment or decrement the integer on track 3 according to the motion of the input head of N_2. Clearly M computes $t_2(t_1(x))$. Finally, we must verify that M

uses $O(\log n)$ space. Track 1 uses $\log |x|$ space. It is not hard to see that track 2, 3 and 4 uses $\log(|t_1(x)|) = O(\log |x|)$ space. **Q.E.D.**

Choice of reducibilitites (III). For most of the open inclusion questions, the consensus opinion generally favor a negative answer. Using smaller families of transformations (such as **DLOG** instead of **P**) to define reducibilities gives us sharper tools for proving negative answers to these questions. For instance, suppose L is NP-complete under some $\leq$-reducibility and P is closed under $\leq$ (thus the premises of the Basic Inclusion Lemma are satisfied). To show that NP is not contained in P, it is sufficient to prove that L is not $\leq$-reducible to some P-complete language L' (in the next chapter we shall show the P does have complete languages). Clearly, it would be easier to prove this result if $\leq$ were in fact $\leq_m^L$ rather than $\leq_m^P$. However, there are inherent limits to this tact of tool sharpening, as illustrated by the next result from [6]. ■

Lemma 4 *Let Φ be the families of transformations computed by transducers running in space $f(n)$. If $f(n)$=$o(\log n)$ then there does not exist NP-complete languages under $\leq_m^\Phi$-reducibility.*

Proof. If t is a transformation computed by some transducer using $f(n) = o(\log n)$ space then for all x, $|t(x)| = |x|2^{O(f(|x|))} = o(|x|^2)$. Let $L_0 \in NTIME(n^k)$ for some $k \geq 1$. If $L' \leq_m^\Phi L_0$ via t then it is easy to see that $L' \in NTIME(n^{2k})$. In particular, if L_0 is NP-complete under $\leq_m^\Phi$-reducibility then $NP = NTIME(n^{2k})$. This contradicts the fact that $NP \neq NTIME(n^k)$ for any k (this result is shown in chapter 6). **Q.E.D.**

Choice of reducibilities, conclusion (IV). This lemma says if the reducibility is weakened below logarithmic space then it is useless for distinguishing P from NP. Combined with preceding discussions, we draw the lesson that the strength of the reducibility should chosen appropriately for the inclusion question of interest. It turns out that because the major questions we ask center around the canonical list, many-one log-space reducibility $\leq_m^L$ is suitable for most of our needs. ■

Despite lemma 4, there are subfamilies of the logarithmic-space transformations **DLOG** that are useful for a variety of purposes. We briefly describe three such subfamilies of **DLOG**.

(a) Consider what is essentially the smallest non-trivial family of transformations, namely, those computed by transducers that use no space (these are called *finite state transducers*). Denote this family by **FST**. This family is closed under functional composition [1]. If the input head of the transducer is constrained to be one-way, we denote the corresponding subfamily by **1FST**.[3] We will show in chapter 6 that there are complete languages for $DTIME(n^k)$ under such $\leq_m^{1FST}$-reducibilities.

[3]The 1-way finite state transducers are also called *generalized sequential machines* or gsm.

(b) Let **Llin** denote the subfamily of **DLOG** where $t \in$ **Llin** implies that for all x, $|t(x)| = O(|x|)$. Meyer, Lind and Stockmeyer first considered these 'log-linear' transformations and corresponding $\leq_m^{Llin}$-reducibilities. In chapter 6, we will see applications of such transformations in lower bound proofs.

(c) Let **1DLOG** denote the family of transformations computable by transducers that use logarithmic space but where the input tapes are one-way. The corresponding reducibility $\leq_m^{1L}$ has been studied by Hartmanis and his collaborators. In particular, they point out that many of the well-known complete languages for $DLOG, NLOG, P$ and NP remain complete under $\leq_m^{1L}$ reducibilities. It is easy to see that such reducibilities are transitive.

We next prove some useful properties about the canonical list of complexity classes of chapter 2 (§3):

$$DLOG,\ NLOG,\ PLOG,\ P,\ NP,\ PSPACE,$$
$$DEXPT,\ NEXPT,\ DEXPTIME,\ NEXPTIME,$$
$$EXPS,\ EXPSPACE.$$

Theorem 5

(i) *All the classes in the canonical list are closed under $\leq_m^{Llin}$-reducibility.*

(ii) *Each class in the canonical list, except for DEXPT and NEXPT, is closed under $\leq_m^{L}$-reducibility.*

Proof. (i) We prove this for the case $DEXPT$. Say $L \leq_m^{Llin} L'$ via some $t \in$ **Llin** and L' is accepted by some M in deterministic time $2^{O(n)}$. To accept L, we operate as follows: on input x compute $t(x)$ and then simulate M on $t(x)$. The time taken is clearly $O(1)^{|t(x)|} = O(1)^{|x|}$.
(ii) Proofs similar to the above can be used except that, with log-space reducibilities, the transformed output $t(x)$ could have length polynomial in $|x|$. If M accepts in time $O(1)^n$ then to simulate M on $t(x)$ takes time $O(1)^{|t(x)|}$, which is $\neq O(1)^{|x|}$. This accounts for the exclusion of the classes $DEXPT$ and $NEXPT$. **Q.E.D.**

4.3 Turing Reducibility

We introduce the idea of computing relative to oracles. Such a computation is done by a multitape Turing acceptor M equipped with a special one-way *oracle tape* and with three distinguished states called QUERY, YES and NO. Such an M is called an *oracle* or *query machine*. We allow M to be nondeterministic. Let L be any language. A computation of M *relative to the oracle set L* proceeds in the usual way until the machine enters the QUERY state. Then, depending on whether the word written on the oracle tape is in L or not, M next enters the YES or NO state,

respectively. This transition is called an oracle query. The oracle tape is erased (in an instant) after each oracle query, and we proceed with the computation. We require that there are no transitions into the YES and NO states except from the QUERY state. The machine M with oracle set L is denoted by $\mathrm{M}^{(L)}$. Since other oracle sets could have been used with M to produce different behavior, we denote the query machine M without reference to any specific oracle by $\mathrm{M}^{(\cdot)}$. The acceptance time, space and reversal of $\mathrm{M}^{(L)}$ are defined as for ordinary Turing machines with the provision that *space used on the oracle tape is not counted*. We do not feel justified in charging for the space on the oracle tape since the oracle tape does not allow space to be reused or even to be reexamined (so this 'space' has properties more akin to time complexity). Clearly the time, space and reversal of $\mathrm{M}^{(\cdot)}$ depend on the oracle. We say that the acceptance time of $\mathrm{M}^{(\cdot)}$ is f if for all oracles L, $\mathrm{M}^{(L)}$ accepts in time f. Similar definitions for acceptance space/reversals as well as for running complexity can be made: we leave this to the reader. Query machines define language operators (see appendix of chapter 2): the *oracle operator* $\phi = \phi_{\mathrm{M}}$ corresponding to a query machine $\mathrm{M}^{(\cdot)}$ is defined by $\phi(L)=L'$ iff $\mathrm{M}^{(L)}$ accepts L'. (The alphabet of L' is taken to be the input alphabet of $\mathrm{M}^{(\cdot)}$.)

Example 1 Recall the traveling salesman decision problem TSD in chapter 3. Consider the following variation, called here the *traveling salesman minimum tour problem* (abbr. TSM):

Given: a pair $\langle D, \pi \rangle$ where D is a $n \times n$ matrix of non-negative integers and π is a permutation on $\{1, \ldots, n\}$.

Property: π represents a minimum cost tour.

It is easy to construct a query machine $\mathrm{M}^{(\cdot)}$ that solves TSM relative to an oracle for TSD: on input $\langle D, \pi \rangle$, M begins by checking the input has the correct format and then computing the cost c of the tour π. Then it asks the oracle two questions (by writing these words on the tape): $\langle D, c \rangle$ and $\langle D, c-1 \rangle$. Our machine M will accept if and only if the first answer is yes, and the second answer is no. Clearly this oracle machine runs in polynomial time and reduces the problem TSM to the problem TSD. ■

Extending our convention for acceptors and transducers, it is sometimes convenient to say '$x \in \mathrm{M}^{(L)}$' to mean that x is accepted by the query machine $\mathrm{M}^{(L)}$.

Definition 4 Let L' and L be languages and Ω a family of oracle operators. L' is *Turing-reducible* to L *via* Ω, denoted $L' \leq_T^{\Omega} L$, if there exists an oracle operator $\phi \in \Omega$ such that $\phi(L)=L'$. If M is an oracle machine computing ϕ, we also write $L' \leq_T L$ via ϕ or via M. If K any class of languages, then we let Ω^K denote the class of languages that are Turing-reducible to some language in K via some operator in Ω. If K consists of a single language A, we also write Ω^A. ■

Notation. Resource bounded complexity classes are defined by suitable resource bounded acceptors. If we replace these acceptors by oracle machines with corresponding resource bounds, where the oracles come from some class K, then the class of languages so defined is denoted in the original manner except that we add K as a superscript. For instance, if F any family of complexity functions then $DTIME^K(F)$ denotes the class of languages accepted by oracle machines $\mathrm{M}^{(A)}$ in time $t \in F$ with $A \in K$. Or again, $N\text{-}TIME\text{-}SPACE^K(n^{O(1)}, \log n)$ is the class of languages that are Turing-reducible to languages in K in simultaneous time-space $(n^{O(1)}, \log n)$. We call the classes defined in this way *relativized classes*. The classes in the canonical list can likewise be 'relativized', and we have:

$$DLOG^K, NLOG^K, P^K, NP^K, PSPACE^K, etc.$$

Any question involving complexity classes can now be asked of relativized classes. In particular, the inclusion questions can be relativized; for instance, the NP versus P problem can be relativized as follows:

Does there exist an oracle A such that $NP^A \subseteq P^A$?

The treatment of relative classes is given in chapter 9.

For the time being we consider the deterministic oracle operators; in section 9 we will return to the nondeterministic case. The two major families of deterministic oracle operators are $\mathcal{P}$ and $\mathcal{DLOG}$, consisting of those oracle operators computable by some deterministic query machine that runs in polynomial time and logarithmic space (respectively).[4] If $\Omega = \mathcal{P}$, the corresponding polynomial-time reducibility is denoted $\leq_T^P$. Similarly, if $\Omega = \mathcal{DLOG}$, we denote the reducibility by $\leq_T^L$. Cook's Theorem was originally stated using $\leq_T^P$ instead of $\leq_m^P$. In the literature, $\leq_T^P$ is also known as *Cook reducibility*.

Lemma 6

(i) *$\leq_T^P$ is a transitive reducibility.*

(ii) *$\leq_T^L$ is a transitive reducibility.*

Lemma 7

(i) *P is closed under $\leq_T^P$.*

(ii) *$DLOG$, $NLOG$ and $PLOG$ are closed under $\leq_T^L$.*

[4]We use script letters for families of oracle operators.

The proofs of lemma 6(i) and 7(i) are straightforward, and the proofs of lemma 6(ii) and 7(ii) are similar to the proof of theorem 3. The nondeterministic version of lemma 7(i) is not known to be true: it is not clear that NP is closed under $\leq_T^P$ (see Exercises).

Turing reducibility is the strongest notion of reducibility known. For example, it is stronger than many-one reducibility if we view a transducer as a special type of oracle machine in which the oracle is asked only one question at the end of the computation (and furthermore, this oracle machine accepts if and only if the oracle says yes).

Alternative oracle models. The oracle machines as defined above are called *one-way oracle machines*. Although such machines will serve as the standard model of oracle computation in this book, we will point out some unexpected properties which points to other possibly better definitions. These properties have to do with space-bounded oracle computation. Note that the oracle tape is not counted among the work-tapes. The alternative is to allow the oracle tape to behave like an ordinary work-tape until the machine enters the query state: then, as in the one-way model, the machine next enters the YES or NO state, and the oracle tape is blanked in an instant. We call this the *two-way oracle machine*. The amount of space used by the oracle tape is now included when considering space usage. We give one motivation for preferring the one-way model: we generally expect Turing reducibility to be as strong as the corresponding many-one reducibility. For instance, it seems that $\leq_T^L$ ought to be as strong as the many-one reducibility $\leq_m^L$. Note that a one-way oracle machine using log-space may make queries of polynomial length. Under the two-way oracle model the machine can only make queries of logarithmic length, and hence it is no longer evident that $\leq_T^L$ is as strong as $\leq_m^L$. In fact, $\leq_T^L$ is not even a reducibility (but $\leq_m^L$ clearly is a reducibility) (Exercises). The next two results illustrate further differences between the two oracle models.

Theorem 8 *Under the two-way oracle model, for any oracle A and any complexity function s:*

(i) *(Relativized Savitch's theorem) $NSPACE^A(s) \subseteq DSPACE^A(s^2)$.*

(ii) $NSPACE^A(s) \subseteq DTIME^A(n^2 \log n O(1)^{s(n)})$.

The relativized Savitch's theorem was noted in [18]. The proof of this theorem is an exercise but it is essentially the same as in the unrelativized cases. In contrast, with respect to the one-way machine model, there are oracles that belie the above result. We state such a result of Ladner and Lynch [8] here, but defer its proof till chapter 9.

Theorem 9 *Let s be a complexity function that is space-constructible. Then there is an oracle A such that $NSPACE^A(s) \not\subseteq DSPACE^A(O(1)^s)$.*

This theorem contradicts the relativized Savitch's theorem in a very strong sense: there is an oracle A such $NSPACE^A(s) \not\subseteq DSPACE^A(s^k)$ for every integer k. It also contradicts the unrelativized result that $NSPACE(s)$ is included in $DTIME(O(1)^s)$, for $s(n) \geq \log n$. Such properties have suggested to researchers that our definitions of oracle space may be "pathological". To obtain a more reasonable notion, Gasarch suggests the following:

Definition 5 A *tame* oracle machine is a one-way oracle machine that has a distinguished set of states called *oracle states*. The machine only writes on the oracle tape when in these states, and transitions from oracle states are deterministic and can only go into other oracle states or enter the QUERY state. ■

Tameness implies that the oracle machine precedes an oracle query by entering the oracle states. If the machine uses s space, then it has $n2^{O(s(n))}$ configurations, where a configuration here does not include the contents on the oracle tape. Hence the length of the query word in a tame oracle machine using space $s(n)$ is $n2^{O(s(n))}$. It is now easy to prove the analog of theorem 8:

Theorem 10 *Theorem 8 holds under the tame oracle model.*

From now on, whenever space complexity is considered, we will assume that the one-way oracles are tame.

4.4 Universal Machines and Efficient Presentation

We frequently want to restrict a class K to a subclass consisting of those languages in K that have a common alphabet Σ: let $K|\Sigma$ denote this subclass of K. Thus "$K = K|\Sigma$" is an idiomatic way of saying that the all languages in K have alphabet Σ. It turns out that all natural classes K have the property that K and $K|\Sigma$ have essentially the same complexity properties, provided $|\Sigma| \geq 2$ (see Exercises). Indeed, the literature sometimes restricts languages to have the binary alphabet $\Sigma = \{0, 1\}$.

Definition 6 A k-tape *universal Turing acceptor* U is a Turing acceptor with k work-tapes and two read-only input tapes taking inputs of the form $\langle i, x\rangle$, i.e. with i on the first input tape and x on the second input tape. We usually inteprete i as an natural number (represented in a m-adic notation). Let K be any class and for each $i \geq 0$, let L_i be the language

$$L_i = \{x : U \text{ accepts } \langle i, x\rangle\}$$

If $K = K|\Sigma$ then we say that U is a *universal acceptor* for K (alternatively, either U *accepts* K or U *presents* K) if

(i) For all i, $L_i \in K$.

(ii) For all $L \in K$, there are infinitely many indices i such that $L_i = L$. We call this the *recurrence property*.

Finally, if K is an arbitrary class of languages, we say K *has universal acceptors* (or is *presentable*) if for each Σ, $K|\Sigma$ has a universal acceptor. ■

With any i fixed on the first input tape, we can regard U as a Turing acceptor U_i that accepts $L(U_i) = L_i$. The number i is also called the *code* or *index* of the machine U_i. We sometimes identify U with the family $\{U_i : i = 0, 1, 2, \ldots\}$ of Turing acceptors. (However not every set of Turing acceptors qualifies to be called a universal acceptor.) Note that U can be deterministic or nondeterministic.

Example 2 (A standard universal machine construction) Recall that RE is the class of recursively enumerable languages. We construct a universal acceptor U^{RE} for the class $RE|\Sigma$ where Σ is any alphabet. Let $k \geq 1$ be fixed. Let $\mathcal{M}_0$ be the family of all nondeterministic k-tape Turing acceptors whose input alphabet is Σ. It is easy to see that the class of languages accepted by the machines in $\mathcal{M}_0$ is precisely the class $RE|\Sigma$. We represent each acceptor in $\mathcal{M}_0$ by its finite transition table, listing in some arbitrary order the tuples in the table. Our universal Turing machine U^{RE} will simulate each machine from this representation. Since the tape alphabets of machines in $\mathcal{M}_0$ are unrestricted while U^{RE} must use a fixed alphabet, we must encode each symbol in the universal symbol set Σ_∞, say, as a word in $\{0,1\}^*$. Once this is done, each machine M in $\mathcal{M}_0$ is naturally represented by binary strings i which we call *codes* of M. (Codes are not unique since the tuples the transition table of M can be ordered in orbitrarily.) The string i can be interchangeably regarded as an integer (in such contexts, when we write $|i|$ we mean the length of the binary string i, never the absolute value of the integer i). In case a binary string i does not encode a valid machine description, we say it encodes the *null machine* that accepts no inputs. The preceding conventions amount to an enumeration of the machines in $\mathcal{M}_0$:

$$\phi_1, \phi_2, \phi_3, \ldots \tag{4.3}$$

where each machine in $\mathcal{M}_0$ occurs at least once in the list. Now it is easy to construct U^{RE} that upon input $\langle i, x\rangle$ simulates the ith acceptor ϕ_i on input x in a step by step fashion; we leave the details to the reader. Note that U^{RE} must be assumed to be nondeterministic if it is to simulate nondeterministic steps of the simulated machine. We can show that the recurrence property holds: for each r.e. language L, L is accepted by U_i^{RE} for infinitely many i's. This follows from the following simple observation: for each transition table δ, there are infinitely many other tables δ' that are obtained from δ by adding extra instructions (i.e., tuples) that can never be executed. Hence if U_i^{RE} accepts a language L, then there must in fact be infinitely many other indices j such that U_j^{RE} also accepts L. This completes our description of the universal machine U^{RE}.

It is easy to see that our construction has the following fundamental property:

(4) For each machine ϕ_i in $\mathcal{M}_0$, if ϕ_i accepts in simultaneous time-space-reversal (t, s, r) then there are infinitely many indices j such that U_j^{RE} accepts $L(\phi_i)$ in time-space-reversal $(O_i(t), O_i(s), O_i(r)) = O_i(t, s, r)$.

Property (4) follows from the observation that U_j^{RE} uses $O_i(1)$ units of its resource for each unit of the corresponding resource of the ϕ_i. The reader should understand how this is so.

To continue this example, we consider the family $\mathcal{M}_1$ of all k-tape deterministic Turing machines. We want a universal simulator U^D for $\mathcal{M}_1$. One way to do this is to use the same enumeration of the machines in $\mathcal{M}_0$, but to define the ith machine ϕ_i as the null machine whenever i does not encode a deterministic acceptor. U^D can easily check if ϕ_i is deterministic, in $O(|i|^2) = O_i(1)$ time. Again, a statement similar to (4) can be made. An alternative is to regard each representation of a nondeterministic acceptor as a representation of a deterministic one simply by assuming that the first choice is to be taken whenever there are several choices for a transition. U^D is deterministic, naturally. ■

The preceding example illustrates the basic principle that a reasonable family of machines can often be systematically enumerated, leading to a universal machine. The principle is also called the "arithmetization or Gödel numbering of machines". In Complexity Theory, it is not enough to say that a class K has a universal acceptor, for we also want the universal machine to be 'efficient'. Intuitively, the machine U^{RE} in the above example is efficient because (4) holds. On the other hand, it is easy to construct a universal acceptor U for the class NP such that each U_i is a very inefficient simulator of ϕ_i (the ith nondeterministic polynomial time machine), say, U_i runs exponentially slower than ϕ_i.

Efficient Presentation. We would like to say that a universal machine is an "efficient" presentation of a class of languages K. To be concrete, in what sense can a universal machine U be said to "efficiently" present the class NP? We would like this to mean that for each i, U_i accepts a language in NP in non-deterministic polynomial time. Unfortunately, NP is simply a class of languages: despite the fact that the definition NP involves the complexity characterization "non-deterministic polynomial time", there is no reason why this characterization is inherent to NP. We therefore proceed as follows: we explicitly attach the complexity characterization $\chi =$"non-deterministic polynomial time" to the class NP and call the pair (NP, χ) a "characteristic class" based on NP. Of course, we can obtain other characteristic classes based on NP if we wish. For instance, if $P = NP$, then it is reasonable to consider (NP, χ') where $\chi' =$"deterministic polynomial time". Now, the notion of efficient presentation of a characteristic class is perfectly meaningful.

Definition 7 *A* resource bound *is a pair $\beta = (F, \rho)$ where F is a family of complexity functions and ρ is a computational resource such as time or space. A* complexity

characteristic χ *is a pair of the form*

$$(\mu, \beta)$$

where μ *is a computational mode, and* β *is a resource bound.* ■

We usually write F-ρ instead of (F, ρ) for resource bounds. For instance,

$$n^{O(1)}\text{-}time, \log^{O(1)} n\text{-}space$$

are resource bounds. This may be read as *poly-time* and *polylog-space*. This notation extends naturally. As further examples, the complexity characteristics

$$(nondeterministic, log(n)\text{-}reversals), (deterministic, O(1)^n\text{-}time).$$

are called *nondeterministic log-reversals* and *deterministic exponential-time*, respectively.

Definition 8 *Let* $\chi = (\mu, \beta)$, $\beta = (F, \rho)$. *Let* $MACHINES(\chi)$ *denote the family of acceptors that have the characteristic* χ, *i.e., each* M *in* $MACHINES(\chi)$ *computes in mode* μ *and accepts in resource bounds* $\beta = (F, \rho)$ *(i.e., in* f *units of resource* ρ, *for some* $f \in F$*). A characteristic class (of languages) is a pair* (K, χ) *where* K *is a language class and* χ *a complexity characteristic such that for each* $L \in K$, *there is some* M *in* $MACHINES(\chi)$ *that accepts* L. *We say* (K, χ) *is* based *on* K. ■

It is easy to generalize the notion of complexity characteristics χ to more than one resource bound: in general let $\chi = (\mu, \{\beta_1, \ldots, \beta_k\})$. Then a machine in the class $MACHINES(\chi)$ computes in mode μ and accepts in simultaneous resource bounds of β_i for each $i = 1, \ldots, k$. For our present purposes, $k = 1$ is sufficient. Although we have defined a complexity characteristic χ as a two parameter object, it is conceivable to introduce further parameters such as: (i) the choice between running and acceptance complexity, (ii) the number of work-tapes, and even (iii) the computational model (we implicitly assumed Turing acceptors), etc.

Example 3 For each of the classes in the canonical list, there is a natural complexity characteristic associated with its definition. Thus the class $DLOG$ of deterministic log-space languages is associated with the characteristic

$$(deterministic, \log n\text{-}space).$$

Indeed, the names for these classes implicitly refer to their characteristics. Henceforth, when we refer to a characteristic class K in the canonical list, we imply the complexity characteristic of K used in its definition. As another example, 'the characteristic class NP' refers to

$$(NP, (nondeterministic, n^{O(1)}\text{-}time)).$$

More generally, for any characteristic class (K, χ), if χ is understood then we may simply refer to K as the characteristic class itself.

Different complexity characteristic attached to a given class K. Thus, Savitch's theorem (chapter 2) tells us that the following are two characteristic classes based on $PSPACE$:

$$(PSPACE, (deterministic, n^{O(1)}\text{-}space)),$$
$$(PSPACE, (nondeterministic, n^{O(1)}\text{-}space)).$$

■

In the literature, the distinction between a characteristic class and its underlying abstract abstract class of languages is often only implicit.

Example 4 Note that in the definition of the characteristic class (K, χ), we allow the possibility of some M in $MACHINES(\chi)$ that accepts a language not belonging to K. This freedom gives rise to trivial ways of obtaining characteristic classes based on any language class K, simply by choosing the complexity family F in χ to be the family of all complexity functions. We next give a non-trivial example. Below we will prove a result about the following characteristic class:

$$(NPC, (nondeterministic, n^{O(1)}\text{-}time))$$

where $NPC \subseteq NP$ is the class of NP-complete languages (under Karp reducibility). If $P \neq NP$ then we see in section 7 that there exist languages accepted by machines in $MACHINES(nondeterministic, n^{O(1)}\text{-}time)$ that are not in NPC. ■

We come to our main definition.

Definition 9 Let (K, χ) be a characteristic class, $K = K|\Sigma$, and U a universal machine. We say U is *an efficient universal acceptor for* (or is *an efficient presentation of*) the characteristic class (K, χ) if

(a) For each i, $L(U_i) \in K$.

(b) Each U_i has complexity characteristic χ. Here, reversals on the first input tape (containing i) are not counted, and space on both input tapes is not counted.

(c) Let $\chi = (\mu, (F, \rho))$. For each $L' \in K$, there is a $f \in F$ such that for infinitely many $i \geq 0$, $L' = L(U_i)$ and each U_i accepts in resource bound (f, ρ). This is the *efficient recurrence property* of U.

In general, when K is not necessarily of the form $K = K|\Sigma$, we say (K, χ) *has efficient universal acceptors* if for each alphabet Σ, there is an efficient universal machine for $(K|\Sigma, \chi)$. ■

Example 5 (Efficient universal acceptors for P) Fix an alphabet Σ. We will construct a universal acceptor U^P for the characteristic class $(P|\Sigma, \chi)$ where $\chi = (deterministic, n^{O(1)}\text{-}time)$. In example 2, we described a universal acceptor U^D for simulating the family $\mathcal{M}_1$ of k-tape deterministic Turing acceptors with input alphabet Σ. We obtain U^P from U^D as follows: U^P, upon input $\langle i, x\rangle$ simulates U^D on input $\langle i, x\rangle$ for at most $|x|^{|i|}$ steps (these are steps of U_i^D).[5] If U_i^D does not accept x within $|x|^{|i|}$ steps, then U^P rejects. U^P can keep track of the number of simulated steps since $f(n) = n^{|i|}$ is time-constructible and U^P can in parallel (i.e., on separate tapes) run for exactly $f(n)$ steps while simulating U_i^D. Furthermore, we need the fact that the family of polynomials $\{n^k : k = 1, 2, \ldots\}$ are uniformly constructible (i.e., the two argument 'universal-polynomial' function $p(n, k) := n^k$ is time-constructible). (Exercises) We call U^P a 'clocked' version of U^D.

We claim that U^P is an efficient presentation of the characteristic class $P|\Sigma$. Clearly U^P accepts only languages in $P|\Sigma$. Each language L in $P|\Sigma$ is accepted by some M in time $O(n^h)$ for some $h \geq 1$. By a tape reduction result in chapter 2, there is a 2-tape machine in $\mathcal{M}_1$ accepting L in time $O(n^h \log n) = O(n^{h+1})$. Then property (4) in example 2 implies that there exists an i such that U_i^P accepts L in time $O(n^{h+1})$. Finally, we must show the efficient recurrence property of U^P: for each $L' \in P$, there is a $c > 0$, and there are infinitely many i's such that $L(U_i) = L$ and U_i accepts in time $c \cdot n^{h+1}$. This follows from our standard coding of machines. (it is insufficient to appeal to the recurrence property of U^D). ■

We now show efficient universal machines for the characteristic classes in the canonical list. The technique is simple and generally applicable; it involves attaching a 'clock' to each Turing machine defined by a suitable universal machine. But this is precisely the technique illustrated in the last example.

Definition 10 A universal machine U is *total* if for all inputs $\langle i, x\rangle$, U eventually halts on that input. A class is *recursively presentable* if it has a total universal machine. ■

Theorem 11 *Each characteristic class in the canonical list is efficiently presentable by a total universal machine.*

Proof. The construction for P in the above example can be generalized to all the other classes. Here we use the fact that the families of complexity functions $(n^{O(1)}, \log n,$ etc) are all constructible in the corresponding complexity resource (time or space). **Q.E.D.**

We extend the preceding definitions for acceptors to transducers and oracle machines. This can be done in a rather straightforward way by imitating the definitions for acceptors. Hence we are contented with a very quick description. We define

[5] More precisely, if U_i^D simulates the ith machine ϕ_i in the standard listing of $\mathcal{M}_1$, then each step of ϕ_i is simulated by $O(|i|) = O_i(1)$ steps of U^D.

universal transducers and *universal oracle machines*, simply by giving an ordinary transducer or oracle machine an extra read-only input tape, so that input is now a pair $\langle i, x\rangle$. Let Φ and Ψ be families of transformations and oracle operators, respectively. Again we need the concept of *characteristic families* of transducers or oracles, denoted (Φ, χ) and (Ψ, χ) where χ is a complexity characteristic. For each alphabet Σ, we may define $\Phi|\Sigma$ (resp. $\Psi|\Sigma$) to be the subfamily of transformations (resp. oracle operators) whose input and output alphabets are Σ.[6] For characteristic families of transducers or oracles, there is a corresponding notion of efficient presentation for the family. Using the idea of clocks, the reader may prove as an exercise:

Lemma 12

(i) *The characteristic families* **DLOG** *of log-space transformations and* **P** *of polynomial-time transformations are each efficiently presentable.*

(ii) *The characteristic families* $\mathcal{DLOG}$ *of log-space oracle operators and* $\mathcal{P}$ *of polynomial-time oracle operators are each efficiently presentable.*

Definition 11 Two languages $L, L' \subseteq \Sigma^*$ are said to be *finite variants* of each other if $(L - L') \cup (L' - L)$ is a finite set. ■

We extend this definition by saying that two classes of languages $K = K|\Sigma$ and $K' = K'|\Sigma$ are finite variants of each other if for each $L \in K$, there is a finite variant of L in K' and vice-versa. An arbitrary class K is *closed under finite variations* if for each Σ,

(a) K contains the trivial languages $\emptyset$ and Σ^* and

(b) for all $L, L' \subseteq \Sigma^*$ that are finite variants of each other, $L \in K$ if and only if $L' \in K$.

It is easy to see that all the complexity classes defined in this book are closed under finite variation.

We next prove an interesting theorem by Landweber, Lipton and Robertson [11]. It concerns the characteristic class *NPC* of example 4 and depends on the efficient presentability of the characteristic classes *NP* and **P**.

Theorem 13 *The characteristic class NPC of NP-complete languages under Karp reducibility is efficiently presentable.*

Proof.

Let $U = \{U_i\}$ be an efficient universal acceptor for the characteristic class *NP* and let $T = \{T_i\}$ be an efficient universal transducer for the polynomial time transformations. (The notation $\simeq$ next refers to the pairing function defined in the

[6] We do not need the more general case where the input and output alphabets are different.

appendix.) We construct $U' = \{U'_i\}$ such that if $i \simeq \langle j,k \rangle$ then U'_i accepts $L(U_j)$ if SAT is Karp reducible to $L(U_j)$ via T_k; otherwise U'_i accepts some finite variant of SAT. We prove that U' is indeed the desired universal machine.

We show how U' operates: on input x of length n, U'_i *deterministically* verifies for each y of length $\leq \log\log n$ that

$$y \in \text{SAT} \iff T_k(y) \in L(U_j). \tag{4.4}$$

We say the verification for a given y *succeeds* if and only if (4.4) can be verified (using the procedure given below) within n steps. We say that x is a *success* if and only if the verification succeeds for each y of length $m \leq \log\log n$; otherwise x is a *failure*. U' first marks out $\lceil \log\log n \rceil$ tape-cells in $O(n)$ time. (We leave the details for this to the reader – it is similar to the proof in the appendix that $\log^* n$ can be computed in linear time.) To verify a particular y of length m, U' proceeds as follows:

(i) Determine if $y \in$ SAT. Fix any nondeterministic acceptor for SAT that takes time m^c on y, $m = |y|$, for some constant c that does not depend on the input. By results in chapter 2, we can simulate this acceptor deterministically in time $O(1)^{m^c}$.

(ii) Compute $T_k(y)$, using time m^d for some constant d that depends only on k.

(iii) Determine if $T_k(y) \in L(U_j)$. Suppose U_j on an input of length n takes nondeterministic time n^e for some e depending on j. Since $|T_k(y)| \leq m^d$, a deterministic simulation of the nondeterministic U_j takes time $O(1)^{(m^d)^e} = O(1)^{m^{de}}$.

For n sufficiently large, we see that (i–iii) can be done in n steps. Thus deciding if x is a success or failure takes at most n^2 time since there are at most $O(1)^{\log\log n} \leq n$ values of y to check. If x is a success then U_i next simulates U_j on $T_k(x)$; otherwise it simulates the fixed nondeterministic acceptor for SAT on input x. This completes our description of U'.

We now show that U' is an efficient universal acceptor for *NPC*. Suppose $i \simeq \langle j,k \rangle$. If SAT is Karp-reducible to $L(U_j)$ via T_k then we see that for *all* inputs, U'_i behaves exactly like U_j. Conversely, if SAT is not Karp-reducible to $L(U_j)$ via T_k then for *large enough* inputs, U'_i behaves like a particular acceptor for SAT. Since *NPC* is closed under finite variation, the accepted language is still in *NPC*. For the efficient recurrence property, it is not hard to show that for any $L' \in NPC$, there are infinitely many i's such that U'_i accepts L' using $O_i(n^c)$ time, for some c. **Q.E.D.**

The above proof is rather formal, but it is really quite simple in outline: On input x, $U_{\langle j,k \rangle}$ spends a small amount of time, say $f(|x|)$ steps, looking for a counter example to the assertion that SAT is reduced to $L(U_j)$ via T_k. If it fails within $f(n)$ steps, it simulates an algorithm for SAT on input x; if it succeeds, it simulates U_j on input x. For j,k fixed, $U_{\langle j,k \rangle}$ is NP provided $f(n)$ is polynomial-bounded (but

clearly any unbounded non-decreasing function). It is easy to see that this machine accepts U_j or some finite variant of SAT. The point is that if the assertion is false, we will find the counterexample in finite time.

In contrast, we have the result of Chew and Machtey [4] that if $P \neq NP$ then $NP - P$ has no recursive presentation.

4.5 Truth-table Reducibilities

This section introduces a reducibility based on the idea of 'truth-tables'; it is intermediate in strength compared to Turing-reducibility and many-one reducibility. We may view many-one reducibility via a transformation t as a special case of Turing reducibility where the query machine on input x asks only one question

"Is $t(x)$ in the oracle language?"

at the end of the computation, with the decision to accept identical to the answer from the oracle. In truth-table reducibility, this is generalized to the asking more than one question with the decision to accept determined by some fixed function of the answers to these questions. The difference between this and general Turing reducibility is that the set of questions asked is solely a function of x, independent of the oracle. Thus a query machine can be decomposed into two parts: a 'query generator' which formulates a sequence of questions based on the input x, and a 'query evaluator' which evaluates a predicate on the sequence of answers. A simple example is a query generator which on any input x produces two queries (strings) $f(x)$ and $g(x)$, where f and g are transformations; the query evaluator accepts if $f(x)$ but not $g(x)$ is in the oracle.

For any language (Σ, L), define its *indicator function* as the function $\chi_L : \Sigma^* \to \{0,1\}$ such that $\chi_L(x) = 1$ iff $x \in L$.

Definition 12 Let Σ and Γ be alphabets.
(a) A *query generator* is a transformation $g : \Sigma^* \to (\Gamma \cup \{\#\})^*$, where $\#$ is a symbol not in Γ. If $g(x) = y_1\#y_2\#\cdots\#y_k$, we view the y_i's as queries to an oracle.
(b) A *query evaluator* is a language $e \subseteq \Sigma^*\#\{0,1\}^*$ (i.e., each word in e has the form $w\#x$ where $w \in \Sigma^*$ and $x \in \{0,1\}^*$).
(c) A *truth-table* is a pair (g, e) where g and e have the forms (a) and (b) respectively.
(d) Say (Σ, L) is *truth-table reducible* to (Γ, L') *via* (g, e), written

$$L \leq_{tt} L' \text{ via } (g, e),$$

if for all x in Σ^* where $g(x) = y_1\#y_2\#\cdots\#y_k$, we have

$$x \in L \iff x\#\chi_L(y_1)\chi_L(y_2)\cdots\chi_L(y_k) \in e.$$

■

Definition 13 A (deterministic) *truth-table machine* is a pair (G, E) where G is a deterministic transducer computing g and E is a deterministic acceptor for some language e, for some truth table (g, e). We say (g, e) is *polynomial-time* if it is computed by some truth-table machine (G, E) where the generator G and the evaluator E both compute in polynomial-time. In general, to say that a truth-table machine operates within some complexity bounds means that both the generator and evaluator have the same complexity bounds. ■

We defer the definition of nondeterministic truth-table machines to section 9. Let 'tt' abbreviate 'truth-table', as in 'tt-machine' or 'tt-reducibility'. We now collect the various types of reducibilities obtained by restrictions on the query generator g and evaluator e. Suppose $(\Sigma, L) \leq_{tt} (\Gamma, L')$ via (g, e).

(i) The truth-table (g, e) is *positive* if e satisfies the restriction that for all $x \in \Gamma^*$ and $b_i, b_i' \in \{0, 1\}$,

$$x\#b_1b_2\cdots b_k \in e \text{ and } \bigwedge_{i=1}^{k}(b_i \leq b_i') \text{ implies } x\#b_1'b_2'\cdots b_k' \in e.$$

The corresponding truth-table reducibility is described as *positive* and denoted $\leq_{ptt}$.

(ii) The truth-table is *conjunctive* if e satisfies the restriction that for all $x \in \Gamma^*$ and $b_i \in \{0, 1\}$,

$$x\#b_1\cdots b_k \in e \iff \bigwedge_{i=1}^{k}(b_i = 1).$$

Similarly, *disjunctive* truth tables are defined by using disjuncts '$\bigvee$' instead of the conjuncts '$\bigwedge$' above. The corresponding truth-table reducibilities are denoted $\leq_{ctt}$ and $\leq_{dtt}$, respectively.

(iii) Let k be a positive integer. If g makes k queries on every input x then (g, e) is called a *k-truth-table*. We then say L is k-truth-table reducible to L' via (g, e) and the corresponding notation is $L \leq_{k\text{-}tt} L'$. Similarly, any of the above restrictions on $\leq_{tt}^{P}$ can be modified by the parameter k:

$$\leq_{k-ptt}, \leq_{k-ctt}, \leq_{k-dtt}.$$

(iv) A truth-table (g, e) is *bounded* if it is a k-truth-table for some k. We then say L is *bounded truth-table reducible* to L' via (g, e), denoted $L \leq_{*\text{-}tt} L'$. Similarly, the other restrictions on $\leq_{tt}^{P}$ can be modified: $\leq_{*-ptt}, \leq_{*-ctt}, \leq_{*-dtt}$.

Clearly a many-one reducibility is simultaneously a positive, conjunctive and a disjunctive truth-table as well as a 1-truth-table.

Notations. If (g, e) is computed by a (deterministic) polynomial-time tt-machine and L is truth-table reducible to L' via (g, e) then we write $L \leq^P_{tt} L'$; more generally, if (g, e) satisfies any of the restrictions in (i-iv) above, we append a superscript P to the corresponding reducibility notation. Thus $\leq^P_{k\text{-}tt}$ denotes polynomial-time k-truth-table reducibility. Similarly, if we restrict the space usage of the query generator and query evaluator to logarithmic space, then we append the superscript L. Thus we have $\leq^L_{tt}, \leq^L_{ptt}, etc.$

Lemma 14 *The following truth-table reducibilities are transitive:*

$$\leq^P_{tt}, \leq^P_{*\text{-}tt}, \leq^P_{ptt}, \leq^P_{ctt}, \leq^P_{dtt}, \leq^P_{1\text{-}tt}.$$

Among the special tt-reducibilities we have introduced, the list in this lemma omits just the k-tt-reducibilities where $k > 1$. We prove the lemma for the case of $\leq^P_{1\text{-}tt}$, leaving the rest as exercise.

Proof. Let A, B, C be languages. Suppose $A \leq^P_{1\text{-}tt} B$ via (g, e) and $B \leq^P_{1\text{-}tt} C$ via (g', e'). We will show that $A \leq^P_{1\text{-}tt} C$. Now, $x \in A$ iff $x\#\chi_B(g(x)) \in e$ and $y \in B$ iff $y\#\chi_C(g'(y)) \in e'$. Substituting $g(x)$ for y, we see that $x \in A$ holds iff either

$$x\#1 \in e \text{ and } g(x)\#\chi_C(g'(g(x))) \in e'.$$

or

$$x\#0 \in e \text{ and } g(x)\#\chi_C(g'(g(x))) \notin e'.$$

holds. That is, iff one of the following holds:

(a) $x\#1 \in e$ and $g(x)\#1 \in e'$ and $g'(g(x)) \in C$.

(b) $x\#1 \in e$ and $g(x)\#0 \in e'$ and $g'(g(x)) \notin C$.

(c) $x\#0 \in e$ and $g(x)\#1 \in e'$ and $g'(g(x)) \notin C$.

(d) $x\#0 \in e$ and $g(x)\#0 \in e'$ and $g'(g(x)) \in C$.

Now define the truth-table (g'', e'') where $g''(x) = g'(g(x))$ and e'' is given by the following two rules. A word $x\#1$ is in e'' iff either

$$x\#1 \in e \text{ and } g(x)\#1 \in e'$$

or

$$x\#0 \in e \text{ and } g(x)\#0 \in e'$$

holds. Again, a word $x\#0$ is in e'' iff either

$$x\#1 \in e \text{ and } g(x)\#0 \in e'$$

or

$$x\#0 \in e \text{ and } g(x)\#1 \in e'$$

holds.
The reader can verify that $A \leq_{1\text{-}tt} C$ via (g'', e'') and that (g'', e'') can be computed in polynomial time. **Q.E.D.**

It is also easy to show

Lemma 15 *The following lists of reducibilities (modified by any given complexity bound β such as polynomial-time or log-space) are in order of non-decreasing strength:*

(i) $\leq_m^\beta, \leq_{k\text{-}tt}^\beta, \leq_{(k+1)\text{-}tt}^\beta, \leq_{*\text{-}tt}^\beta, \leq_{tt}^\beta, \leq_T^\beta$

(ii) $\leq_m^\beta, \leq_{ctt}^\beta, \leq_{ptt}^\beta, \leq_{tt}^\beta$

In (ii), we could have substituted $\leq_{dtt}^\beta$ for $\leq_{ctt}^\beta$.

Alternative definition of tt-reducibilities. Often the truth-table (g, e) embodies 'Boolean functions' in the sense that on any input x where $g(x) = y_1\# \cdots \#y_k$ then there is a Boolean function f_x on k variables such that $x\#b_1b_2 \cdots b_k \in e$ if and only if $f_x(b_1, \ldots, b_k) = 1$. This leads to the following alternative definition of tt-reducibilities. Fix a representation of Boolean functions as binary strings, and let f_w denote the Boolean function f_w represented by the binary string w. With query generators g defined as before, we now define an *evaluator* e as a transformation $e : \Sigma^* \to \{0,1\}^*$; intuitively, $e(x)$ represents the Boolean function $f_{e(x)}$ used to determine the final answer. We say (Σ, A) is tt-reducible to (Γ, B) via (g, e) if for all $x \in \Sigma^*$ the following is satisfied: $x \in A$ iff

$$f_{e(x)}(\chi_B(y_1), \chi_B(y_2), \ldots, \chi_B(y_k)) = 1$$

where $g(x) = y_1\# \cdots \#y_k$ for some $k \geq 1$. The particular method of representing Boolean functions could affect the strength of the reducibility. We consider three increasingly more succinct representations:

(a) A k-variable function can be represented by a Boolean 2^k-vector. In some literature, such a vector is also called a 'truth-table' which in turn lends its name to the entire family of reducibilities in this section.

(b) By Boolean formulas over the basis $\{\wedge, \vee, \neg\}$.

(c) By Boolean circuits over the same basis.

In the Exercises, we see that with respect to polynomial-time bounds, these three alternative definitions of tt-reducibilities do not lead to a different notion of tt-reducibility. We also give another definition of tt-reducibility in the Exercises that is again equivalent to our standard choice. These results lend credibility to the claim that our standard choice is robust. Unfortunately, with respect to log-space bounds, the situation seems to be different.

4.6 Strength of Reducibilities

Among the reducibilities, there are pairs $(\leq_1, \leq_2)$ where it is obvious that $\leq_1$ is as strong as $\leq_2$. For instance, $(\leq_1, \leq_2) = (\leq_T^P, \leq_m^P)$. A natural question is whether $\leq_1$ is, in fact, stronger than $\leq_2$. In this section, we illustrate some of these results. The notion of partial sets will be useful in our proofs:

Definition 14 *A partial set (relative to a set X) is a partial function f from X to $\{0,1\}$. The* universe *of f is X, and its* domain, $\mathrm{dom}(f)$, *consists of elements $x \in X$ such that f is defined at x. If f' is also a partial set with universe X, we say f' and f* agrees *if they are identical on $\mathrm{dom}(f) \cap \mathrm{dom}(f')$. Call f' a* partial subset *of f (or, f extends f') if $\mathrm{dom}(f') \subseteq \mathrm{dom}(f)$ and f' and f agrees.* ■

A basic diagonalization outline. The proofs in this section are based on the idea of 'diagonalization'. The reader will better understand these proofs if she keeps in mind the following outline: Suppose the result in question follows from the existence of a language L that satisfies a countably infinite list of conditions

$$C_0, C_1, C_2, \ldots$$

These conditions are finite in the sense that each condition only refers to a finite subset of L and they are 'existential' in that if L is a partial set that satisfies any condition then any extension of L also satisfies that condition. The proof involves constructing the language in stages where in the ith stage we have built a partial set L_i with finite domain such that each of the conditions $C_0, C_1, \ldots, C_i$ is satisfied by L_i. Typically, $\mathrm{dom}(L_i) \subseteq \mathrm{dom}(L_{i+1})$.

There are many variations to this basic outline. Perhaps there are more than one infinite list of conditions to satisfy. Perhaps we cannot guarantee satisfying a given condition C during any stage; so we may have to try to satisfy the condition C for infinitely many stages, and prove that our attempts will succeed in *some* stage. Instead using the conditions C_i to define the ith stage, we can use an "inverted diagonalization" by considering all strings in the domain: fix some total ordering of the set of input words. Usually, we use[7] the **lexicographic ordering**:

$$x_0, x_1, x_2, x_3, \ldots \tag{4.5}$$

That is, we list the strings in order of non-decreasing lengths, and among those of the same length we use the usual dictionary order. We decide whether x_j ought to belong to the language L in stage j. This is done by attempting to use x_j to satisfy some condition C_i in our infinite list. We may need to return a particular C_i in many stages (but hopefully C_i will be satisfied in some finite state). The advantage

[7] Despite the terminology, this is NOT the same as dictionary order which does not have strings listed in order of non-decreasing length.

of this approach is that our description of the diagonalization process amounts to specifying an acceptor for the constructed language L.

We proceed with the first result that shows that 1-tt-reducibility is stronger than Karp reducibility. Note that for any language L, we have L is 1-tt-reducible to *co-L*. Therefore our desired result is an immediate consequence of the following from Ladner, Lynch and Selman [9]:

Theorem 16 *There exists a non-trivial language L in $DEXPT$ such that L is not Karp-reducible to co-L.*

Before proving this theorem, it is instructive to give a simple proof of a similar result that does not require L to be in $DEXPT$. The simpler proof will follow the basic diagonalization outline above: we construct $L \subseteq \{0,1\}^*$ in stages. Let

$$T = \{T_0, T_1, T_2, \ldots\}$$

be an efficient universal transducer for the polynomial-time transducers used in Karp reducibility. The desired result follows if L satisfies the following infinite list of conditions:

$$C_i : L \text{ is not many-one reducible to } \textit{co-}L \text{ via } T_i$$

for $i = 0, 1, 2, \ldots$. Note that condition C_i is equivalent to the existence of a word x satisfying

$$x \in L \iff T_i(x) \in L.$$

Such an x is said to "cancel" T_i and x is also called a "witness" for condition C_i. Let $x_0, x_1, \ldots$, be the lexicographic ordering of all binary strings. At stage i, assume inductively that we have a partial set L_i that has a finite domain; L_i is a partial subset of the language L that we are trying to construct. Furthermore, for each $h < i$, the condition C_h is satisfied. We now wish to define L_{i+1} whose domain contains the domain of L_i. Suppose x_j is the next word (in the lexicographical ordering) not in the domain of L_i. Let $y = T_i(x_j)$ and we consider two possibilities: if y is already decided, then put x_j into the domain of L_{i+1} in such a way that x_j cancels T_i; if y is not yet decided, let us put both x_i and y into L (again cancelling T_i). It is not hard to show that condition C_i is satisfied in stage i: one only need to see that x_j exists so that the plan to cancel T_j can be carried out. One should also verify that our arguments is valid even if $x_j = y$.

This simple proof tells us little about the complexity of L (although L is evidently recursive). We need a technique due to Machtey and others to "slow down the diagonalization". To do this, we use the slow growing function $\log^* n$ and its inverse $\mathrm{Exp}(n)$ defined in the appendix.

Proof of theorem 16. We will use the "inverted diagonalization outline" in which the ith stage examines the ith word in a lexicographic listing of all words. We

define a Turing acceptor M that accepts a language L over the alphabet $\Sigma = \{0,1\}$ in *DEXPT* where L is not Karp-reducible to *co-L*. On input $x \in \{0,1\}^*$ where $|x| = n$:

(a) If x is the empty word ϵ, accept.

(b) If x contains a '1' or if $n \neq \text{Exp}(m)$ for some m then reject. Note that this step takes linear time since we can check if $n = \text{Exp}(\log^* n)$ in linear time, as shown in the appendix.

(c) Hence assume $x = 0^n$ (a string of zeroes) such that $n = \text{Exp}(\log^* n))$. Let $\log^* n = m \simeq \langle i, j \rangle$ for some m, i, j. The appendix shows how m, i, j can be extracted in polynomial time (actually, we will not need j). Simulate the universal transducer T on input $\langle i, x \rangle$ for 2^n steps of T. If T_i does not halt within 2^n steps, reject.

(d) Hence assume that T_i produces the output $y = T_i(x)$ within 2^n steps. If $|y| > \text{Exp}(m-1)$ then reject. If $|y| \leq \text{Exp}(m-1)$ then accept iff $y \in L$ (which we determine by a recursive call to M on y).

Thus step (d) uses x to explicitly[8] cancel the condition C_i; the first three steps (a-c) ensure that x is used in step (d) then x has the correct form: $x = 0^n$ where $n = \text{Exp}(\log^* n)$. The construction of M is a 'slow down' of the diagonalization process since condition C_i is (explicitly) cancelled at the $\text{Exp}(i)$th stage or later.

Correctness: We will show that L is non-trivial and is not Karp-reducible to *co-L*. To see non-triviality, L contains the empty word ϵ and does not have words containing 1's. Aiming for a contradiction, assume that L is Karp-reducible to *co-L* via some T_i. We may assume that T_i takes $|x|^c$ time, for some $c = c(i)$, on input $\langle i, x \rangle$. Choose j large enough so that if $m \simeq \langle i, j \rangle$ and $n = \text{Exp}(m)$ then

$$2^n > n^c.$$

Then it is easy to see that on input $x = 0^n$, the acceptor M will proceed to step (d) after obtaining an output $y = T_i(x)$. Hence, $|y| \leq n^c < 2^n = 2^{\text{Exp}(m)} = \text{Exp}(m+1)$. There are two cases:

$$|y| \leq \text{Exp}(m-1)$$

and

$$\text{Exp}(m-1) < |y| < \text{Exp}(m+1). \tag{4.6}$$

In the first case, by the definition of M, x is in L iff y is in L. In the second case, x is rejected so we must show that y is also rejected. In fact, (4.6) implies that unless y has the form $0^{\text{Exp}(m)}$, y would be rejected. But if $y = 0^{\text{Exp}(m)}$ then $x = y$, again showing that y is rejected. Thus the algorithm is correct.

[8]There may be "accidental" cancellation of some conditions.

Timing Analysis: We show that M runs in time $O(2^n)$ time. Steps (a-c) and the non-recursive part of step (d) clearly takes $O(2^n)$ steps. The recursive call to M has argument y with length $\leq \text{Exp}(m-1) = \log_2 n$. If $t(n)$ is the time required by M then we have $t(n) \leq t(\log n) + O(2^n)$. The solution to this recurrence gives $t(n) = O(2^n)$. **Q.E.D.**

It should be clear that this proof works with any superpolynomial time bound f in place of 2^n, provided that both f and its 'inverse' are both easy to compute.

The fact that there is a language in *DEXPT* that distinguishes Karp- from tt-reducibility prompts the question: does there exist a language in *NP* that distinguishes them? Of course, if we can prove an affirmative answer then it is easy to see that $NP \neq P$. But even assuming that $NP \neq P$, this is an open question.

Suppose $\leq_1$ is as strong as $\leq_2$. To show that the former is in fact stronger, we have to show the existence of languages A and B such that $A \leq_1 B$ but not $A \leq_2 B$. This is usually quite simple to show, especially if we can carry out the basic diagonalization outline above. But often we would like to impose *side restrictions* on A, B. For instance, we may insist (as in the last theorem) that B does not have very high complexity.

Example 6 One reason for imposing side restrictions is that they yield more information about the relationships between two reducibilities. Ladner, Lynch and Selman define the relation 'stratifies' where $\leq_1$ is said to *stratify* $\leq_2$ if there exists L, L' such that L and L' are of the same $\leq_2$-degree but L and L' are $\leq_1$-incomparable. Clearly $\leq_1$ stratifies $\leq_2$ implies that $\leq_2$ is stronger than $\leq_1$, but intuitively, it is more than 'just' stronger: this is because there is a pair of languages that are similar from the viewpoint of $\leq_2$, while being very different from the viewpoint of $\leq_1$. ■

To illustrate another kind of side restriction, we make the following definition: let A be a language and let $\leq_1, \leq_2$ be reducibilities where $\leq_1$ is as strong as $\leq_2$. Then $\leq_1$ and $\leq_2$ are said to be *distinguishable over* A if there exists a language B such that

$$A \leq_1 B \text{ but not } A \leq_2 B.$$

The next result is from I. Simon [18]:

Theorem 17 *For all $A \notin P$, $\leq^P_{(k+1)\text{-}tt}$ and $\leq^P_{k\text{-}tt}$ are distinguishable over A.*

Before embarking on the proof, we need the notion of characteristic families of truth-tables, and efficient universal truth-tables. By now, this can be done routinely as follows: Let Ω be any family of truth-tables, and $\chi = (\mu, F - \rho)$ any complexity characteristic. Then (Ω, χ) is a characteristic family provided that each truth-table $(g, e) \in \Omega$ is computed by some tt-machine (G, E) where G and E both have complexity characteristic χ. A *universal tt-machine* is a pair (T, U) where $T = \{T_i\}$ and $U = \{U_i\}$ are a universal transducer and a universal acceptor, respectively. For any alphabet Σ, $\# \notin \Sigma$, we again have the restriction $\Omega|\Sigma$ of Ω to Σ, consisting of

$(g, e) \in \Omega$ where $g : \Sigma \to \Sigma \cup \{\#\}$ and $e \subseteq \Sigma \cup \{\#, 0, 1\}$. The characteristic family (Ω, χ) is efficiently presentable if for each Σ, there exists a universal (T, U) such that (i) for each $i \geq 0$, (T_i, U_i) computes a truth-table in $\Omega|\Sigma$, and (ii) each truth-table in $\Omega|\Sigma$ is computed by (T_i, U_i) for infinitely many $i \geq 0$ where each T_i and U_i has complexity characteristic in χ. It is an easy exercise to show that the family Ω^P_{tt} (resp. Ω^P_{k-tt}, for each k) of truth-tables computed by tt-machines with complexity characteristic

$$(deterministic, n^{O(1)}\text{-}time)$$

is efficiently presentable.

Proof of theorem 17. Let A be any language not in P; without loss of generality, assume A is over the alphabet $\{0, 1\}$. The proof follows the basic diagonalization outline above. Let (T, U) be an efficient universal truth-table with deterministic polynomial-time characteristic. We will construct a language $B \subseteq \{0, 1\}^*$ such that

$$x \in A \iff |B \cap S_k(x)| \text{ is odd} \tag{4.7}$$

where

$$S_k(x) := \{x0^i1^{k-i} : i = 0, 1, \ldots, k\}$$

Clearly $A \leq^P_{(k+1)\text{-}tt} B$; to show the theorem, we will show that $A \leq^P_{k\text{-}tt} B$ fails. We construct B in stages. Let B_s ($s \geq 0$) be the partial subset constructed at the end of the $(s-1)$st stage. In stage s, we extend B_s to B_{s+1} in such a way that we 'cancel' the truth-table machine (T_s, U_s), i.e., we include a word x in the domain of B_{s+1} such that $x \in A$ if and only if

$$x\#\chi_B(y_1)\cdots\chi_B(y_k) \notin U_s \tag{4.8}$$

where $T_s(x) = y_1\#\cdots\#y_k$ (clearly we may assume that T_s always makes exactly k queries). We will maintain the inductive hypothesis that the domain of B_s consists of all words of length $\leq h(s)$ for some $h(s) \geq 0$, and such that B_s satisfies (4.7) in the sense that for all x of length $\leq h(s) - k$, we have $x \in A$ iff $B_s \cap |S_k(x)|$ is odd. We describe the construction of B_{s+1} in four steps:

(a) Choose any x_0 of length $h(s)+1$; note that x_0 is not in the domain of B_s. Let $T_s(x_0) = y_1\#\cdots\#y_k$. We first ensure that each y_i is in the domain of B_{s+1}: if y_i is not in the domain of B_s then we put it in the domain of B_{s+1}, arbitrarily (say $y_i \notin B_{s+1}$).

(b) We next put x_0 in the domain of B_{s+1}, making $x_0 \in B_{s+1}$ or $x_0 \notin B_{s+1}$ so as to satisfy (4.8). The choices in (a) completely determine this step.

(c) Next include into the domain of B_{s+1} all the remaining words in $S_k(x_0)$ so as to ensure (4.7). Note that this is always achievable because at least one word in $S_k(x_0)$ has not yet been put into the domain of B_{s+1} by step (a) (note that $x_0 \notin S_k(x_0)$).

(d) To clean-up, set $h(s+1) = \max\{|x_0| + k, |y_1|, \ldots, |y_k|\}$. Let $w_1, w_2, \ldots, w_r$ order all the words not yet in the domain of B_{s+1} of length at most $h(s+1)$. We put w_i into the domain of B_{s+1} to satisfy (4.7). To see that this is achievable, we use two simple observations: first, for all words $w \neq w'$, the intersection $S_k(w) \cap S_k(w')$ is empty. Second, setting $Y = \{x_0, y_1, \ldots, y_k\} \cup S_k(x_0)$, we see that for each word $w \neq x_0$ of length $\leq h(s+1)$, $|S_k(w) \cap Y| \leq k$. Note that Y includes all words of length $> h(s)$ that have been put in the domain of B_{s+1} by steps (a-c).

This completes the proof. **Q.E.D.**

4.7 The Polynomial Analogue of Post's Problem

Suppose that $P \neq NP$. Then we may ask if there exists a language in $NP - P$ that is not NP-complete under Cook reducibility. This question is analogous to a famous question posed by Post in 1944: Does there exist a recursively enumerable language that is not recursive and not RE-complete under arbitrary[9] Turing reducibility? An affirmative answer was independently found in 1956 by Friedberg [5] and Muchnik [14]. In this section we show Ladner's result [10] that the polynomial analogue of Post's problem also has an affirmative solution (provided, of course, that $P \neq NP$). We remark that there are very few natural problems that are conjectured to be in $NP - P$ and not NP-complete. Essentially the only well-known candidates are primes[10] and graph isomorphism.

Theorem 18 (Ladner) *If $P \neq NP$ then there exists a language in $NP - P$ that is not NP-complete under Cook reducibility.*

This theorem is a consequence of the next result.

Theorem 19 *Let K a class of languages over the alphabet $\Sigma = \{0, 1\}$ and P denote as usual the class $DTIME(n^{O(1)})$. Suppose K has a total universal acceptor and is closed under finite variation. Let (Σ, L) be a recursive language not in $K \cup P$. Then there exists a language (Σ, L') such that*

(i) *L' is not in K,*

(ii) *L' is Karp-reducible to L,*

(iii) *L is not Cook-reducible to L'.*

[9] That is, Turing reducibility with no resource bounds on the query machines.

[10] A prime candidate indeed. Linear Programming used to be another such candidate until Khacian showed it to be in P. Lesser-known candidates include Minimum Weight Triangulation and Discrete Logarithm.

To see that theorem 19 implies Ladner's theorem, let $K = P|\Sigma$ and $L \in (NP|\Sigma)\backslash P$. Since K has a total universal acceptor, it follows from theorem 19 that there exists an L' such that L' is not in P (by (i)), L' is in NP (by (ii)) and L' is not NP-complete under Cook reducibility (by (iii)).

Proof of theorem 19. Let

$$U = \{U_0, U_1, U_2, \ldots\}.$$

and

$$Q = \{Q_0^{(\cdot)}, Q_1^{(\cdot)}, Q_2^{(\cdot)}, \ldots\}.$$

denote (respectively) a universal acceptor for K and a universal query machine for the family of polynomial-time oracle operators whose input and output alphabet is Σ. (Note: we do not require U or Q to be efficient.)

The requirement that L' is not in K (respectively, L is not Cook-reducible to L') is equivalent to the set of even (resp. odd) numbered conditions in the following infinite list of conditions C_i $(i = 0, 1, \ldots)$ where

$$\begin{aligned} C_{2j}: &\quad L' \neq U_j \\ C_{2j+1}: &\quad L \neq Q_j^{(L')} \end{aligned}$$

Let

$$x_0, x_1, x_2, \ldots \tag{4.9}$$

be the usual lexicographic enumeration of words in Σ^*. We shall construct a transducer T that runs in linear time such that T: $\Sigma^* \to \{0\}^*$. Then define

$$L' := \{x : x \in L \text{ and } |\mathrm{T}(x)| \text{ is even}\}. \tag{4.10}$$

This clearly shows that L' is Karp-reducible to L.

To prove the theorem it remains to describe T such that the resulting L' satisfies the list of conditions above. We first explain in an intuitive way the operation of T on input x. We say that a word x is *in stage* i if $|\mathrm{T}(x)| = i$. T has the property that $|T(x_{i+1})|$ is equal to either $|T(x_i)|$ or $|T(x_i)| + 1$. Hence the words in the ordering (4.9) have non-decreasing stage numbers. We maintain the following invariant:

> (*) If x is in stage i, then condition C_i is not yet satisfied but each condition C_j $(j < i)$ is witnessed by some word y_j that precedes x in the lexicographical order (4.9).

The first thing that T does on input x is to try to determine the stage of x. It can only do this approximately in the sense that it can determine an integer i such that x is either in stage i or stage $i + 1$. This uncertainty turns out not to be crucial because T will try to satisfy condition C_i and if it succeeds, then $|\mathrm{T}(x)|$ is equal to $i + 1$ otherwise i. Given i, consider how T can try to satisfy C_i. There are 2 cases.

(i) Suppose that $i = 2j$. Then T systematically attempts to find a word w such that $w \in L'$ iff $w \notin U_j$. Such a w is a 'witness' for the condition C_{2j}. (Note the self-reference here: we are constructing L' and we want to check if w belongs to L'.) If T is successful in finding a witness, T outputs the word 0^{i+1}; otherwise it outputs the word 0^i. The output 0^{i+1} serves a double purpose: to inform subsequent computations that condition C_i is satisfied and to ensure that x is not in the set L' (recall the definition (4.10) of L'). Likewise for an output of 0^i.

(ii) Suppose $i = 2j + 1$. T systematically attempts to find a witness for condition C_{2j+1}, i.e., a word w such that $w \in L$ iff $w \notin Q_j^{(L')}$. (Again note the self-reference.) If successful in finding the witness w, x is put in stage $i+1$ by the output $T(x) = 0^{i+1}$; else it stays in stage i with the output $T(x) = 0^i$.

Now we fill in the details. Let $|x| = n$. First, to determine the approximate stage of x as follows. T allocates a total of n steps to compute (by simulating itself) the values $\mathrm{T}(0^1), \mathrm{T}(0^2), \ldots, \mathrm{T}(0^m)$ in succession, where m is such that the allocated n steps are used up while in the middle of computing $\mathrm{T}(0^{m+1})$. Then x is in the approximate stage i where $i = |\mathrm{T}(0^m)|$. We next attempt to find a witness for the ith condition C_i by testing successive words in the ordering (4.9) until a total of n steps are used up. If a witness is found within n steps then output $T(x) = 0^{i+1}$; otherwise output $T(x) = 0^i$. Consider how we test for the witness property:

(I) Suppose $i = 2j$. To test whether any word w is a witness, T checks (a) whether $w \in U_j$ (by simulating the universal acceptor U) and (b) whether $w \in L'$ (by checking if $|\mathrm{T}(w)|$ is even and if $w \in L$ – the latter is possible since L is recursive). Note that w is a witness if the outcomes for (a) and (b) differ.

(II) Suppose $i = 2j + 1$. To test whether w is a witness, we check (c) whether $w \in L$ as before and (d) whether w is accepted by $Q_j^{(\cdot)}$ using oracle L'. To do (d), we simulate the universal query machine Q using input $\langle j, w\rangle$, and whenever a query "Is z in L'?" is made, we can check this as in (b) above.

Correctness: By construction T runs in linear time. It is also easy to see that our invariant (*) holds. In particular, if T goes through stage i for all i then each condition would be satisfied and we are done. Therefore we only need to ensure that T does not remain in some stage i forever. For the sake of contradiction, we assume that for some i and some n_0, we have $|\mathrm{T}(x)| = i$ for all $|x| \geq n_0$, In other words, suppose T never leaves stage i.

(I) Suppose $i = 2j$. Since we never leave stage $2j$, by the definition of L', L' is equal to a finite variant of L. Now $L \notin K$ and K is closed under finite variation imply that L (and hence L') must differ from U_j for infinitely many inputs. Choose the first (with respect to the lexicographical ordering (4.9)) witness

w_0 to the fact that $U_j \neq L'$. Now choose n large enough: precisely, $n > n_0$ and n exceeds the number of steps needed by T to compute $\mathrm{T}(0^1), \ldots, \mathrm{T}(0^m)$, where $|\mathrm{T}(0^m)| = 2j$, and exceeds the number of steps to test if x is a witness to condition C_{2j} for all x preceding w_0 in the ordering (4.9). Observe that the smallest such number n is well-defined. Then clearly $|\mathrm{T}(0^n)| = 2j + 1$, a contradiction.

(II) Suppose $i = 2j + 1$. Then by definition of L', L' is a finite set. Now observe that L must differ from $Q_j^{(L')}$ on infinitely many words, for otherwise $L \in P$ which would be contrary to our assumption. Let w_0 to be the first (with respect to the ordering (4.9)) witness word to the fact that $L \neq Q_j^{(L')}$. Again choose $n > n_0$ such that it exceeds the time to determine that 0^n is in stage $2j + 1$ and exceeds the time to test if x is a witness to the condition C_{2j+1} for all x preceding w_0. Then it is easy to see that $|\mathrm{T}(0^n)| = i + 1$. **Q.E.D.**

An interesting observation from the proof is that L' is the intersection of L with the polynomial time computable set $G = \{x : |\mathrm{T}(x)| \text{ is even}\}$.

Corollary 20 *If $NP \neq P$ then for every NP-complete language L (under $\leq_T^P$) there exists a language $G \in P$ such that $G \cap L$ is in $NP - P$ but is not NP-complete.*

There is another way to view Ladner's proof: G is nothing but a 'gappy' set determined by some recursive function $r(n)$ such that $x \in G$ iff $r(2i) \leq |x| < r(2i + 1)$. Here $r(n)$ is the number of steps needed to find witnesses for all the conditions up to C_n. The size of such gaps is studied in Lipton, Landweber and Robertson [11] and Chew and Machtey [4]. Schöning [17], Schmidt [16] and Regan [15] have greatly simplified and generalized these techniques. An interesting result [6] is the following: Assuming that $P \neq NP$, there is a language in $NP - P$ that is not P-hard under one-way log-space many-one reducibility $\leq_m^{1L}$.

4.8 The Structure of Polynomial Degrees

The reducibilities $\leq$ in this section are assumed to be transitive, so that the term $\leq$-degrees is meaningful. For any two languages (Σ_0, L_0) and (Σ_1, L_1), define their *join* $L_0 \oplus L_1$ to be the language

$$\{0x : x \in L_0\} \cup \{1x : x \in L_1\}.$$

over the alphabet $\Sigma_0 \cup \Sigma_1 \cup \{0, 1\}$. Note that L_1 and L_2 are each $\leq_m^P$-reducible to $L_0 \oplus L_1$. Furthermore, for any L, if $L_i \leq_m^P L$ holds for $i = 0, 1$ then it is easy to see that $L_0 \oplus L_1 \leq_m^P L$. In other words, the $\leq_m^P$-degree of $L_0 \oplus L_1$ is the least upper bound of the $\leq_m^P$-degrees of L_0 and of L_1. The same holds for Cook-reducibility. In the terminology of lattice theory[11] we have just shown:

[11] A partial order $(X, \leq)$ is an upper semi-lattice if for all $x, y \in X$, there is a least upper bound denoted $x \vee y$ (the least upper bound is unique if it exists).

Theorem 21 *The $\leq_m^P$-degrees form an upper semi-lattice. The same holds for $\leq_T^P$-degrees.*

Definition 15 Let K be a class of languages, and $\leq$ a transitive reducibility. We say that the $\leq$-degrees of K are *dense* if the following holds. If $L_0 < L_2$ $(L_0, L_2 \in K)$ then there exists $L_1 \in K$ such that

$$L_0 < L_1 < L_2.$$

■

We now address questions about the density of the above semi-lattices. The theorem of Ladner shows that if $NP \neq P$ then the Karp-degrees in $NP - P$ is dense. Similarly if $P \neq NP$ then the Cook-degrees between P and NP are dense. More generally, we have:

Theorem 22 *If $L <_T^P L'$ and L and L' are recursive then there exist a recursive L'' such that $L <_T^P L'' <_T^P L'$.*

There have been considerable developments in such questions (e.g., see [2]).

4.9 Nondeterministic reducibilities

In this section, we consider reducibilities defined by nondeterministic machines (acceptors, transducers, query machines, tt-machines). Of these, nondeterministic tt-machines have not been defined. But first, we extend the notion of many-one reducibility to multivalued transformations.

Definition 16 *Let $(\Sigma, A), (\Gamma, B)$ be languages and f be the multivalued transformation from Σ to Γ computed by some transducer T. We say that A is* many-one (nondeterministically) reducible *to B via f (or via T) if for all $x \in \Sigma^*$,*

$$x \in A \iff f(x) \cap B \neq \emptyset.$$

■

A nondeterministic tt-machine (G, E) consists of a nondeterministic transducer G and a nondeterministic evaluator E. It turns out that, at least for polynomial-time bounds, we can restrict E to be deterministic (see Exercises). The corresponding pair (g, e) computed by a nondeterministic (G, E) is called a *multivalued truth-table*.

We say A is *nondeterministically tt-reducible* to B via (g, e) (or via (G, E)) if for all $x \in \Sigma^*$, there exists $y_1 \# \cdots \# y_k \in g(x)$ such that

$$x \in A \iff x \# \chi_B(y_1) \cdots \chi_B(y_k) \in e.$$

Recall that $\chi_B(x)$ is the indicator function for the set B. For nondeterministic polynomial-time bounded reducibilities, we use the superscript 'NP' in the usual way. For instance, $\leq_T^{NP}$, $\leq_{tt}^{NP}$ and $\leq_m^{NP}$ are the nondeterministic counterparts of $\leq_T^P$, $\leq_{tt}^P$ and $\leq_m^P$. These new reducibilities have rather different properties from their deterministic counterparts. The next result is by Ladner, Lynch and Selman.

Theorem 23 *The following pairs of reducibilities form pairs of identical binary relations over the non-trivial languages:*

(i) $\leq_T^{NP}$ *and* $\leq_{tt}^{NP}$

(ii) $\leq_{ptt}^{NP}$ *and* $\leq_{ctt}^{NP}$

(iii) $\leq_{dtt}^{NP}$ *and* $\leq_m^{NP}$.

Proof. In each case, the first member of a pair is as strong as the second. Hence it is sufficient to show that the second is as strong as the first.

(i) Suppose $A \leq_T^{NP} B$ via a nondeterministic polynomial-time oracle machine M and we want to show $A \leq_{tt}^{NP} B$. Choose any $x_0 \notin B$ and $x_1 \in B$ (this is possible since we assume non-trivial languages). We will construct a nondeterministic tt-machine (G, E) such that $A \leq_{tt}^{NP} B$ via (G, E). On any input x, the query generator G simulates M on input x. G also maintains on a special work-tape T a list of values, initially empty. If the ith query ($i \geq 0$) of M to the oracle is the word y_i then G guesses the oracle's answer. If the guessed answer is yes, then G records in the tape T the pair (y_i, x_1); otherwise it records in T the pair (y_i, x_0). If M accepts, then G outputs the contents of tape T:

$$y_1 \# z_1 \# y_2 \# z_2 \# \cdots \# y_k \# z_k$$

where $z_i \in \{x_0, x_1\}$. If M rejects, G outputs $x_0 \# x_1$. We now have to describe the acceptor E. On input $x \# b_1 c_1 \cdots b_k c_k$, ($b_i, c_i \in \{0, 1\}$), E accepts iff $b_i = c_i$ for all $i = 1, \ldots, k$. It is easy to see that $A \leq_{tt} B$ via (G, E).

(ii) Let $A \leq_{ptt}^{NP} B$ via (G, E). We will construct a nondeterministic tt-machine (G′, E′). G′, on input x, simulates G on x. If G outputs $y_1 \# \cdots \# y_k$, then G′ guesses a subset $I \subseteq \{1, 2, \ldots, k\}$ and checks if $x \# b_1 \cdots b_k$ is accepted by E, where $b_i = 1$ iff $i \in I$. If it is accepted then G′ outputs $y_{i_1} \# y_{i_2} \# \cdots \# y_{i_m}$ where $I = \{i_1, i_2, \ldots, i_m\}$; otherwise G′ outputs x_0 where x_0 is some fixed word not in B. The evaluator E′ is essentially determined since we use conjunctive reducibility. The reader can easily verify that $A \leq_{ctt}^{NP} B$ via (G′, E′).

(iii) This follows the proof for (ii) closely. **Q.E.D.**

We now show that some, though not all, of the nondeterministic reducibilities are intransitive:

Theorem 24 (i) $\leq_m^{NP}$ *and* $\leq_{ctt}^{NP}$ *are transitive.*

(ii) $\leq_{tt}^{NP}$, $\leq_{*\text{-}tt}^{NP}$ *and* $\leq_{k\text{-}tt}^{NP}$ $(k \geq 1)$ *are intransitive.*

Proof. The proof of (i) is straightforward, and is omitted. To show (ii), it is sufficient to construct non-trivial languages A, B, C over $\{0,1\}$ satisfying

(a) $A \leq_{1\text{-}tt}^{NP} B \leq_{1\text{-}tt}^{NP} C$

(b) $A \not\leq_{tt}^{NP} C$.

Note that (a) follows from the following two conditions:

$$x \in A \iff (\exists y)[\ |y| = |x| \text{ and } y \notin B\ ] \tag{4.11}$$

$$x \in B \iff (\exists y)[\ |y| = |x| \text{ and } y \in C\ ] \tag{4.12}$$

Let $\mathrm{Q} = \{\mathrm{Q}_i\}$ be an efficient universal oracle machine for the characteristic family of nondeterministic polynomial-time oracle machines (it is easy to see that Q exists). By definition each Q_i runs in polynomial time; we may further assume that each Q_i uses less than 2^n steps if $n \geq i$. Since $\leq_T^{NP}$ and $\leq_{tt}^{NP}$ are identical relations, (b) corresponds to an infinite list of conditions where the ith condition states:

$$A \text{ is not Turing-reducible to } C \text{ via } \mathrm{Q}_i.$$

We proceed in stages where we satisfy the sth condition in stage s, $s \geq 0$. Let that the partial sets $A_{s-1}, B_{s-1}, C_{s-1}$ have been defined at the end of the $s-1$st stage. Also, inductively assume that each of the sets $A_{s-1}, B_{s-1}, C_{s-1}$ has a common domain. Moreover, for each n, all words of length n are in the common domain or all or not.

To initialize, each of the sets A_0, B_0, C_0 has domain consisting of just the empty string ϵ where $\epsilon \in A_0, \epsilon \notin B_0 \cup C_0$. In stage s pick any word x of length n where n is the smallest integer such that no word of length n is in the current common domain. We will make x a witness to condition s by considering two cases:

Case (1): x is accepted by Q_s using oracle C_s (where queries on words not in the domain of C_s are given NO answers). Let Y be the set of words queried in some arbitrary accepting computation path of $\mathrm{Q}_s^{(C_s)}$ on x. Then we extend the domains of A_s, B_s and C_s by omitting all words of length $|x|$ from A_{s+1}, including all words of length $|x|$ into B_{s+1} and including exactly one string $y \notin Y$ of length $|x|$ into C_{s+1}. The existence of y follows from the fact that $|Y| < 2^{|x|}$ since we assume that Q_s runs in time $< 2^n$ for $n \geq s$. Observe that condition s as well as (4.11) and (4.12) are satisfied.

Case (2): x is not accepted by $\mathrm{Q}_s^{(C_s)}$. Extend the respective domains by including all words of length $|x|$ into A_{s+1} but omitting them all from B_{s+1} and from C_{s+1}. Again, condition s as well as properties (4.11) and (4.12) are satisfied.

We now clean-up for stage s. For each m such that some word of length m is queried in some computation path of $\mathrm{Q}_s^{(C_s)}$ on input x, if such words are not already in the common domain of $A_{s+1}, B_{s+1}, C_{s+1}$, we put every word w of length m domains of $A_{s+1}, B_{s+1}, C_{s+1}$ so as to satisfy properties (4.11) and (4.12): more precisely, we exclude w from C_{s+1} and from B_{s+1} but include w in A_{s+1}. Note that omitting w from C_{s+1} is consistent with the replies that $\mathrm{Q}_s^{(C_s)}$ received on input x. This completes our description of stage s. **Q.E.D.**

Before concluding this section, we describe a weaker notion of nondeterministic reducibility investigated by Long [12]:[12]

Definition 17 *Let $(\Sigma, A), (\Gamma, B)$ be languages and f a multivalued function from Σ^* to Γ^*. We say A is unequivocal many-one reducible to B via f if for all $x \in \Sigma^*$, we have*

$$\begin{aligned} x \in A &\Rightarrow f(x) \subseteq B, \\ x \notin A &\Rightarrow f(x) \cap B = \emptyset. \end{aligned}$$

We write $A \leq_{um} B$ via *f in this case.* ■

Recall (chapter 2, section 9) that a nondeterministic acceptor is unequivocal if for all inputs, there is at least one terminating computation path, and terminating computation paths are all accepting or all rejecting. We say A is *unequivocally Turing-reducible to B via* a query machine M if A is Turing-reducible to B via M and $\mathrm{M}^{(B)}$ is unequivocal. We then write $A \leq_{uT} B$ *via* M.

Remark: All reducibilities seen until now in this book could equally have been made in terms of the 'abstract' concept of transformations, oracle operators or truth-tables as well as in terms of their 'concrete' counterparts of transducers, oracle machines or tt-machines. In contrast, this last reducibility apparently cannot be defined without explicit reference to the concrete concept of machines. There will be other examples.

In the usual way, we append superscripts P, NP, etc to the various reducibilities to indicate complexity bounds. Unequivocal many-one reducibility has proven to be useful in classifying the complexity of number-theoretic problems. The polynomial-time versions of these are called *gamma-reducibilities* (γ-reducibilities) by Adleman and Manders who introduced them. Gamma reducibilities are valuable for studying closure under complements because of the following result: there are *NP*-complete languages under γ-reducibilities and furthermore if L is *NP*-complete under γ-reducibilities then $NP = co\text{-}NP$ iff $L \in co\text{-}NP$. In general, unequivocal reducibilities seem to have nice properties, as seen in the following result by Long.

Lemma 25 *Unequivocal many-one reducibilities and unequivocal Turing-reducibilities with polynomial-time bounds are transitive reducibilities.*

[12] Long qualifies these reducibilities as 'strong' because his notion of strength of reducibilities is opposite to ours.

4.10 Conclusion

In this chapter we introduced the important reducibilities and their basic properties. Many other forms of reducibility have been studied but here we are contented to give a few examples.

(1) Lynch has investigated the use of multiple oracles (by endowing machines with more than one oracle tape).

(2) *Reductions based on logical constructs.* For any complexity function f, Jones defines the *f-bounded rudimentary reducibility* based on certain simple logical predicates and quantifications bounded by f. For instance, the $\log n$-bounded rudimentary reducibility is weaker than $\leq_m^L$.

(3) An interesting concept due to J. Simon is similar to many-one reductions except for an extra 'parsimony' constraint. Roughly speaking, parsimony insists that the reducibility preserves the number of solutions. In illustration of this idea, consider the reduction of Hamiltonian circuit problem to SAT. If we have a transformation t that takes any graph G to a corresponding CNF formula F such that the number of Hamiltonian circuits in G is equal to the number of satisfying assignments to F, then t is parsimonious. To formalize this notion, we must reformulate[13] a 'problem' to be a binary relation over Σ^* for some alphabet Σ: this would allow us to count the number of solutions to a problem, and thus define parsimonious reductions among such problems. Once this is done, it can be shown that all known transformations of *NP*-complete problems can be modified to be parsimonious.

[13]Naturally this takes us outside the 'standard' complexity theory in the sense of departing from some basic decisions made in chapter one.

Exercises

[4.1] Consider the following definition[14] of *NP*-completeness: a language L is *NP*-complete if (a) L is in *NP* and (b) if L is in P then $NP = P$. Clearly SAT is *NP*-complete under this definition. This definition appears attractive because it does not appeal to any notion of reducibility. What is unusual about it? How does the polynomial analogue of Post's problem fare under this definition?

[4.2] (Aho-Hopcroft-Ullman) Consider the following informal notion of K-hardness: say L is *effectively K-hard* if there is an effective (i.e. recursive) procedure F such that for each $L' \in K$, $F(L')$ is a transformation such that L' is many-one reducible to L via $F(L')$. Make this into a precise definition. Verify that SAT is effectively *NP*-hard under your definition. Most notions of *NP*-hardness are based on a concept of reducibility. Is there a corresponding reducibility here?

[4.3] (i) By modifying the proof in chapter 3, show that SAT is *NP*-complete under many-one log-space reducibility.
(ii) In fact, show that all the *NP*-complete problems in chapter 3 are still *NP*-complete under the one-way log-space $\leq_m^{1L}$ reducibility of Hartmanis.

[4.4] Show that $LBA = NSPACE(n)$ is precisely the class of context sensitive languages.

[4.5] Prove that *DLOG* is not closed under $\leq_m^P$-reducibility if $DLOG \neq P$.

[4.6] For any K and any family of transformations Ω, show that L is K-complete iff *co-L* is (*co-K*)-complete under $\leq_m^\Omega$-reducibility. Conclude that if L is P-complete under Karp reducibility then *co-L* is also P-complete.

[4.7] (Meyer) Show that $NP =$ *co-NP* iff there exists an *NP*-complete language L in *co-NP*.

[4.8] (Jones, Lien, Laaser) Show that t is a deterministic log-space transformations iff the following language is in *DLOG*:

$$\{x\#i\#b : x, i \in \{0,1\}^* \text{ and } b \text{ is the } i\text{th symbol of } t(x)\}$$

for some transformation $|t(x)| = |x|^{O(1)}$.

[4.9] Show that the familiar functions $x + y$, $x - y$, $\max(x, y)$ and $x \cdot y$ are in **DLOG** (the family of deterministic log-space transformations). It is not known if $\lfloor x/y \rfloor$ is in **DLOG**; but obtain a small space complexity s such that division can be computed by a transducer using space s.

[14] we may call this the 'student definition' because it often appears in oral or written exams when students are asked for a definition. Of course, teachers do it too. See discussion in [20].

[4.10] What is the flaw in the following proof that NP is closed under Cook reducibility: let $L \in NP$ and L' is accepted by some deterministic polynomial time oracle machine $\mathrm{M}^{(\cdot)}$ with oracle L. To show that L' is in NP we define a machine M' which on input x simulates $\mathrm{M}^{(L)}$ until M enters the QUERY state. Then M' temporarily suspends the simulation of M and begins simulating a nondeterministic polynomial time acceptor N of L using the query word. If N accepts then M' continues the simulation of M from the YES state, otherwise it continues from the NO state. Clearly M' computes in nondeterministic polynomial time.

[4.11] Let $|\Sigma| \geq 2$. Show that $P = NP$ iff $P|\Sigma = NP|\Sigma$

[4.12] Say the language (Σ, L) is a *coded image* of (Γ, L') if there is a homomorphism $h : \Gamma \to \Sigma^*$ such that $h(a)$ has the same length for all a in Γ and the natural extension of h, $h^* : \Gamma^* \to \Sigma^*$, maps L' isomorphically into L. A class K is *coded in* K' if every language in K has a coded image in K'. K and K' are *codably equivalent* if they are each coded in the other. Show that P and $P|\Sigma$ are codably equivalent iff $|\Sigma| \geq 2$. Show the same for the other standard classes. Conclude that with respect to the $P = NP$, $DLOG = NLOG$ and $P = NLOG$ questions, our theory might as well be restricted to languages over $\{0, 1\}$.

[4.13] (infinitely often reducibility) Let t be a transformation $t : \Sigma^* \to \Gamma^*$. A language (Σ, L) is *infinitely often (i.o.) many-one reducible* to (Γ, L') if for infinitely many $x \in \Sigma^*$, we have $x \in L$ iff $t(x) \in L'$. We extend this definition to incorporate complexity in the obvious way: e.g. 'i.o. Karp reducibility' refers to i.o. many-one reducibility in polynomial time, denoted $\leq^P_{i.o.m}$. Investigate the properties of such efficient reducibilities.

[4.14] In this exercise we assume the two-way oracle model.
(i) Show that the relationship $\leq^L_T$ is not reflexive (and hence it is not a reducibility). *Hint:* Consider any in $DLBA - DL$.
(ii) Show that the reducibilities $\leq^{DLBA}_m$ and $\leq^{DLBA}_T$ are incomparable (here the superscript $DLBA$ denotes linear space bounds.) *Hint:* To show that $\leq^{DLBA}_T$ is not stronger than $\leq^{DLBA}_m$ pick any language $L \notin DLBA$ and show that L is not $\leq^{DLBA}_T$-reducible to where $L' = \{ww : w \in L\}$.

[4.15] * For any language L, define the language $L' = \{\mathrm{bin}(|x|) : x \in L\} \subseteq \{0, 1\}^* \subseteq \{0, 1\}^*$. Give an efficient reduction of L' to L. If L is accepted in time-space-reversal (t, s, r), what are the obvious complexity bounds on L'? Find non-trivial improvements on these bounds.

[4.16] Show that if f is space-constructible then $DSPACE(f)|\Sigma$ (for any alphabet Σ) is efficiently presentable.

[4.17] * (i) Show that the two-argument *universal-polynomial* complexity function $p(n,k) := n^k$ is constructible, for integer values of the inputs. *Hint:* First show that $p'(n,k) = \Theta(n^k)$ is constructible. Actually, this is sufficient for all purposes.
(ii) Fill in the details for the construction of an efficient universal acceptor for the characteristic class P.
(iii) More generally, show that if a family F of complexity functions has a time-constructible universal-function $F^*(i,n)$, then $NTIME(F)$ is efficiently presentable. That is, for each fixed i, $F^*(i,n) \in F$ and conversely each $f(n) \in F$ is equal of $F^*(i,n)$ for some n.

[4.18] Define a *super universal machine* to be a Turing acceptor U with three input tapes. If the inputs are $\langle i,j,x\rangle$ then if i and j are fixed, we regard the machine as an ordinary Turing acceptor on input x, and denote it by $U_j^{(i)}$. If i is fixed we regard the family

$$U^{(i)} = \{U_0^{(i)}, U_1^{(i)}, \ldots\}$$

as a universal machine. We say that U efficiently presents a characteristic class K if (a) for each i, $U^{(i)}$ is an efficient presentation of K, and (b) for each efficient universal acceptor V of K, there exists an i such that $U^{(i)}$ has the same enumeration of K as V: for each j sufficiently large, $L(U_j^{(i)}) = L(V_j)$. Does the characteristic class NP have a super universal acceptor?

[4.19] Prove that for any reasonable resource bound β, say $\beta = n^{O(1)}$-*time*, $\leq_m^\beta$ and $\leq_{1\text{-}ptt}^\beta$ are identical.

[4.20] (Ladner, Lynch and Selman) Prove that (i) $\leq_{1\text{-}tt}^P$ stratifies $\leq_m^P$, and (ii) $\leq_{(k+1)\text{-}tt}^P$ stratifies $\leq_{k\text{-}tt}^P$. (see section 6 for definition of stratifies)

[4.21] Show that $\leq_{k\text{-}tt}^P$ for $k > 1$ is an intransitive reducibility.

[4.22] (Ladner, Lynch and Selman) Define the reducibility $\leq_0$ thus: $L \leq_0 L'$ if there is a query machine and a polynomial-time transformation $t : \{0,1\} \to \{\#\{0,1\}^*\}^*$ such that $M^{(L')}$ accept L and for input x, M only asks questions from the list $t(x)$ (ie. $\#x_1\#x_2\#\cdots\#x_k$ where $x_i \in \{0,1\}$ is viewed as the list $(x_1, x_2, \ldots, x_k)$). Show that $\leq_0$ is the same as polynomial-time truth-table reducibility.

[4.23] Show that if $A \leq_T^P B$ then A is tt-reducible to B in deterministic time 2^{n^k} for some $k \geq 1$.

[4.24] (Ladner and Lynch) Let L, L' be languages.
(i) Show that $L \cup L' \leq_{tt}^L L \oplus L'$. In fact, the $\leq_{2-tt}^{FST}$-reducibility is sufficient.
(ii) Show the existence of languages L, L' such that $L \cup L' \not\leq_m^L L \oplus L'$.

[4.25] (i) Prove theorem 8 (which contains the relativized Savitch's theorem) under the two-way oracle model. Why does the proof break down with the one-way oracle model?
(ii) Prove theorem 8 again, but under the tame one-way oracle model.

[4.26] Complete the proof of lemma 14.

[4.27] (Ladner, Lynch and Selman) Show that $L \leq_m^{NP} L'$ iff there exists a polynomial p and a polynomial-time transformation t such that $x \in L$ iff

$$(\exists y)[|y| \leq p(|x|) \wedge t(x,y) \in L']$$

[4.28] Prove that if $PSPACE \neq P$ then there exists L, L' in $PSPACE$ such that L is Cook-reducible, but not Karp-reducible to L'. (Of course, if we can prove this result without the hypothesis that $PSPACE \neq P$ then we would have proven that $PSPACE \neq P$.)

[4.29] (Selman) A language (Σ, L) is P-selective if there is a polynomial time transformation $f : \Sigma^* \times \Sigma^* \to \Sigma^*$ such that $f(x,y) \in \{x,y\}$, and furthermore, if either x or y is in L then $f(x,y) \in L$.
(i) Show that for all non-trivial languages L, L is in P iff *co*-$L \leq_m^P L$ and L is P-selective.
(ii) If L is P-selective and $L' \leq_m^P L$ then L' is P-selective.
(iii) If SAT is P-selective then $NP = P$.

[4.30] Show that the following function is a pairing function:

$$\langle x,y \rangle = a^2 - 2(a-b) + \delta(x < y)$$

where $a = \max(x,y)$, $b = \min(x,y)$ and $\delta(x < y) = 1$ if $x < y$ holds and 0 otherwise. Show that this, and its associated projection functions π_1, π_2 are polynomial-time computable.

[4.31] Imitate Ladner's proof technique to obtain a language in $NEXPT - DEXPT$ (assuming this is non-empty) that is not complete.

[4.32] (I. Simon) Show that for all $A \notin P$, $\leq_T^P$ and $\leq_{tt}^P$ are distinguishable over A.

[4.33] (I. Simon) Let $\leq_1$ be stronger than $\leq_2$. We say that $\leq_1$ and $\leq_2$ are *downward distinguishable* over a language L if there exists a language L' such that $L' \leq_1 L$ but not $L' \leq_2 L$. (Note that the definition of distinguishable in the text may be qualified as 'upward'.) Let $\leq_T^{PS}$ denote Turing reducibility where the complexity bound is deterministic polynomial space. Show there exists a language $L \notin P$ such that $\leq_T^{PS}$ and $\leq_m^P$ are not downward distinguishable over L.

[4.34] (Ladner and Lynch) Show that $\leq_T^P$ is stronger than $\leq_T^L$.

[4.35] (Ladner and Lynch, open) Is $\leq_m^P$ stronger than $\leq_m^L$? Is $\leq_{tt}^P$ stronger than $\leq_{tt}^L$?

[4.36] Consider the alternative definition of tt-reducibility described at the end of section 5. Compare the relative strengths of the usual tt-reducibility with this definition, respect to each of the three choices of representing Boolean functions: (a) 'truth-tables' in the original sense of the term, (b) Boolean formulas, and (c) Boolean circuits.
(i) Show that with respect to polynomial-time bounds, there is no difference.
(ii) Show a difference (assuming $DLOG \neq P$) with respect to log-space bounds. What is the significance of the fact that the circuit value problem (CVP) is P-complete under log-space reducibility? (Note: the CVP problem, described in chapter 5, is the problem of deciding if a distinguished output of a Boolean circuit under a given input is 0 or 1.)

[4.37] Show that with respect to polynomial time, we could restrict nondeterministic tt-machines (G, E) such that E is deterministic.

[4.38] Prove the last theorem of section 8 which generalizes the Ladner's answer to the polynomial analogue of Post's problem.

[4.39] (Lind, Meyer) Show that the log-space transformations **DLOG** is closed under explicit transformation (viz., substitution by constants, renaming and identification of variables) and two-sided recursion of concatenation. The latter is defined as follows: A transformation $f : (\Sigma^*)^n + 1 \to \Delta^*$ on $n+1$ variables is defined by *two-sided recursion of concatenation* from $g : (\Sigma^*)^n \to \Delta^*$ and $h_1, h_2 : (\Sigma^*)^{n+2} \to \Delta^*$ if

$$f(\bar{w}, \epsilon) = g(\bar{w})$$
$$f(\bar{w}, xa) = h_1(\bar{w}, x, a) \cdot f(\bar{w}, x) \cdot h_2(\bar{w}, x, a)$$

for all $\bar{w} \in (\Sigma^*)^n, w \in \Sigma^*$ and $a \in \Sigma$. Note that in this context, we can regard a multi-variable transformation such as f as an ordinary one variable transformation if we encode a sequence (w_1, w_n) of words as one word $w_1 \# \cdots \# w_n$ where $\#$ is a new symbol.

[4.40] * Following the outline in the concluding section, formalize the notion of parsimony by defining problems as binary relations over Σ^*. How much of standard complexity theory as used in this book carry over?

[4.41] (J. Simon) Show that SAT can be many-one reduced to Hamiltonian circuits using a parsimonious polynomial-time transformation t, i.e., for any CNF $F \in$ SAT, the number of satisfying assignments to F is equal to the number of Hamiltonian circuits in $t(F)$.

[4.42] * Investigate the basic structure of the diagonalization proofs by formalizing the concept of 'finite, existential conditions'. Refer to the basic diagonalization outline of section 6.

Appendix A

Useful number-theoretic functions

Pairing functions. A *pairing function* is any bijection between $\mathbb{N} \times \mathbb{N}$ and $\mathbb{N} = \{0, 1, 2, \ldots\}$. We define a particular pairing function $\langle \cdot, \cdot \rangle : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ in which the correspondence between $(i, j) \in \mathbb{N} \times \mathbb{N}$ and $n \in \mathbb{N}$ is given by the relation

$$n = i + \binom{i + j + 1}{2}$$

where $\binom{x}{2} = x(x-1)/2$. Unless stated otherwise, by 'the pairing function' we refer to this definition. Using $\langle \cdot, \cdot \rangle$ to denote a pairing function is a deliberate confusion of notation since in general we use the same notation for ordered pairs. This confusion is usually without harm. Whenever a distinction is necessary, we will write $\langle i, j \rangle \simeq n$ to say that the value of the pairing function on the values i, j is n. The best way to see that this is a bijection is through the picture of figure A.1.

The bijection induces a total ordering $\preceq$ of $\mathbb{N} \times \mathbb{N}$ defined as follows: the ordered pairs $(i', j'), (i, j)$ are related as $(i', j') \preceq (i, j)$ iff $n' \leq n$ where $\langle i, j \rangle \simeq n$ and $\langle i', j' \rangle \simeq n'$. If, in fact $n < n'$ holds, then we write $\langle i', j' \rangle \prec \langle i, j \rangle$. The (first and second) *projection functions* π_1 and π_2 are defined by

$$n \simeq \langle \pi_1(n), \pi_2(n) \rangle$$

for all n. A simple property of the pairing function is that it is increasing in each of its arguments.

Lemma 26 *The function $\langle \cdot, \cdot \rangle$, π_1 and π_2 are computable in polynomial time.*

Proof. The fact that the pairing function is polynomial-time computable follows from the fact that multiplication is polynomial-time computable. To see that π_1 is polynomial time, note that on input n, in time polynomial in the size of the input,

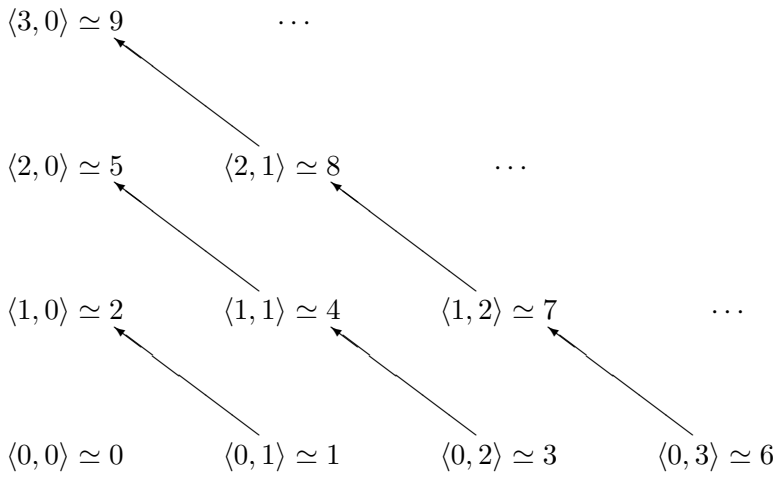


Figure A.1: The pairing function $\langle \cdot, \cdot \rangle$

we can determine the k such that $k^2 \leq 2n \leq (k+1)^2$. (Use a binary search for k in the range between 1 and n.) We want the m such that $m(m-1) \leq 2n \leq m(m+1)$. Clearly $k = m$ or $m-1$, and we can easily determine which is the case. Finally, $\pi_1(n) = n - \binom{m+1}{2}$. It is similarly easy to compute $\pi_2(n)$. **Q.E.D.**

Suppose $k \geq 2$ is fixed. We can use the pairing function to encode k-tuples as follows: a k-tuple $(x_1, \ldots, x_k)$, where $x_i \in \{0,1\}^*$, is encoded by

$$\langle x_1, \langle x_2, \cdots, \langle x_{k-2}, \langle x_{k-1}, x_k \rangle \rangle \cdots \rangle \rangle.$$

The projection functions $\pi_1, \ldots, \pi_k$ are easily defined. This mapping from $\mathbb{N}^k$ to $\mathbb{N}$ is a bijection. If we are willing (this is a matter of taste) to introduce extra symbols for encoding k-tuples, then it is easy to get an alternative encoding: encode $(x_1, \ldots, x_k)$ as $x_1 \# x_2 \# \cdots \# x_k$ where $\#$ is a new symbol. This poorman's encoding has the obvious simple linear-time decoding. However, it is not a bijection. But in all our applications, bijections were nice but unessential.

Finally, we could use the pairing function to encode finite sequences $\langle x_1, \ldots, x_k \rangle$ (of arbitrary length k) by explicitly encoding the length of the sequence:

$$\langle k, \langle x_1, \langle x_2, \cdots, \langle x_{k-2}, \langle x_{k-1}, x_k \rangle \rangle \cdots \rangle \rangle \rangle.$$

Note that this encoding is no longer a bijection between $\mathbb{N}^*$ and $\mathbb{N}$.

A fast growing function and its inverse.

Define the *super-exponential function* $\mathrm{Exp} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ by $\mathrm{Exp}(0) = 0$ and $\mathrm{Exp}(n+1) = 2^{\mathrm{Exp}(n)}$. So $\mathrm{Exp}(n)$ consists of a stack of n two's.[1] In the remainder of this appendix, Exp will refer to the one argument function.

The *log-star function* $\log^* n$ is defined to be the largest integer m such that $\mathrm{Exp}(m) \leq n$. This function is one of the possible the 'inverses' of $\mathrm{Exp}(n)$.

Lemma 27

(i) Exp *can be computed in time linear in its output size* $|\mathrm{Exp}(n)|$

(ii) $\log^* n$ *can be computed in time linear in its input size* $|n|$.

Proof. Note that $|n|$ here refers to the length of the binary representation $bin(n)$ of n, not the absolute value of n. Note that for $n > 0$, $|\mathrm{Exp}(n)|$ is $1 + \mathrm{Exp}(n-1)$ since $bin(\mathrm{Exp}(n))$ is '1' followed by $\mathrm{Exp}(n-1)$ 0's. To show (i), suppose $bin(\mathrm{Exp}(n-1))$ is available, and we want to compute $bin(\mathrm{Exp}(n))$. This amounts to writing out exactly $\mathrm{Exp}(n-1)$ zeroes. To do this, we treat $bin(\mathrm{Exp}(n-1))$ as a binary counter and we want to decrement it to zero. If $m > 0$ is the value of the binary counter and $g(m)$ denote the maximal string of zeroes that form a suffix $bin(m)$ (i.e. the suffix of $bin(m)$ has the form '$\cdots 10^{g(m)}$'), then a single decrement of the counter takes $O(g(m))$ steps. Hence the time to decrement the counter from $\mathrm{Exp}(n-1)$ to zero is order of

$$\sum_{m=1}^{\mathrm{Exp}(n-1)} g(m) = \sum_{k=1}^{\mathrm{Exp}(n-2)} k \cdot 2^{\mathrm{Exp}(n-2)-k} = O(2^{\mathrm{Exp}(n-2)}) = O(\mathrm{Exp}(n-1)).$$

Thus computing $\mathrm{Exp}(n)$ from $\mathrm{Exp}(n-1)$ is linear in $|\mathrm{Exp}(n)|$. Summing this work for $m = 1, 2, \ldots, n$ we get $\sum_{m=1}^{n} |\mathrm{Exp}(m)|$ which is $O(|\mathrm{Exp}(n)|)$, as required. To show (ii), we compute the increasing values

$$\mathrm{Exp}(0), \mathrm{Exp}(1), \mathrm{Exp}(2), .$$

until for some $m \geq 0$, $\mathrm{Exp}(m+1)$ has length greater than $|n|$. Then $\log^* n = m$, and we can convert to binary if desired. The linearity of this procedure is similarly shown as in (i). **Q.E.D.**

[1] $\mathrm{Exp}(n)$ is to be distinguished from the usual exponential function $\exp(n) := e^n$, where e is the base of the natural logarithm.

Bibliography

[1] A. V. Aho and J. D. Ullman. A characterization of two-way deterministic classes of languages. *Journal of Computers and Systems Science*, 4(6):523–538, 1970.

[2] Klaus Ambos-Spies. Sublattices of the polynomial time degrees. *Information and Computation*, 65:63–84, 1985.

[3] T. Baker, J. Gill, and R. Solovay. Relativizations of the $P =? NP$ question. *SIAM J. Computing*, 4:431–442, 1975.

[4] P. Chew and M. Machtey. A note on structure and looking back applied to the relative complexity of computable functions. *Journal of Computers and Systems Science*, 22:53–59, 1981.

[5] R. M. Friedberg. Two recursively enumerable sets of incomparable degrees of unsolvability. *Proceed. Nat. Acad. of Sciences*, 43:236–238, 1957.

[6] Juris N. Hartmanis, Neil Immerman, and Steven Mahaney. One-way log-tape reductions. *19th Symposium FOCS*, pages 65–71, 1978.

[7] Neil D. Jones. Space-bounded reducibility among combinatorial problems. *Journal of Computers and Systems Science*, 11:68–85, 1975.

[8] R. E. Ladner and N. A. Lynch. Relativization of questions about log space computability. *Math. Systems Theory*, 10:19–32, 1976.

[9] R. E. Ladner, N. A. Lynch, and A. L. Selman. A comparison of polynomial time reducibilities. *Theor. Comp. Sci.*, 1:103–123, 1975.

[10] Richard E. Ladner. On the structure of polynomial time reducibility. *JACM*, 22:1:155–171, 1975.

[11] L. H. Landweber, R. J. Lipton, and E. L. Robertson. On the structure of sets in *NP* and other complexity classes. *Theoretical Computer Science*, 15:181–200, 1981.

[12] Timothy J. Long. Strong nondeterministic polynomial-time reducibilities. *Theoretical Computer Science*, 21:1–25, 1982.

[13] A. R. Meyer and E. M. McCreight. Computationally complex and pseudo-random zero-one valued functions. In Z. Kohavi and A. Paz, editors, *Theory of machines and computations*, pages 19–42. Academic Press, 1971.

[14] A. A. Muchnik. On the unsolvability of the problem of reducibility in the theory of algorithms (in russian). *Doklady Akademii Nauk SSSR*, 108:194–197, 1956.

[15] Kenneth W. Regan. The topology of provability in complexity theory. *Journal of Computers and Systems Science*, 38:384–432, 1988.

[16] Diana Schmidt. The recursion-theoretic structure of complexity classes. *Theoretical Computer Science*, 38:143–156, 1985.

[17] Uwe Schöning. A uniform approach to obtain diagonal sets in complexity classes. *Theoretical Computer Science*, 18:95–103, 1982.

[18] Istvan Simon. On some subrecursive reducibilities. Technical Report Tech. Rep. STAN-CS-77-608, Computer Sci. Dept., Stanford Univ., April, 1977. (PhD Thesis).

[19] Osamu Watanabe. On one-one polynomial time equivalence relations. *Theoretical Computer Science*, 38:157–165, 1985.

[20] C. K. Yap. Logical curiosities: on definitions of *NP*-completeness. manuscript, 1985.

Contents

Chapter 5

Complete Languages

March 1, 1999

5.1 Existence of complete languages

The class *NP* has a long list of complete problems derived from almost every area of the computational literature. The theoretical unity that this fact provides for such diverse problems is one of the striking accomplishments of the field. Naturally, one is led to ask if other complexity classes such as *NLOG* and *PSPACE* have complete languages. The framework of the Basic Inclusion Lemma also motivates our search for complete languages. The answer is yes for these and many other classes; moreover, there is a systematic way to show this (e.g., [13]). The method depends on the existence of efficient universal machines; universal machines in turn depend on the existence of suitable tape-reduction theorems. In this chapter, for simplicity, the reducibility is assumed to be $\leq_m^L$ unless otherwise specified.

Convention: Let i denote a natural number. In contexts where a string is expected, we use the same symbol 'i' to denote the binary string representing the natural number i. The notation '$|i|$' always denotes the length of the binary representation of i, never the absolute value of i as a number. Also, if # is a symbol, then $\#^i$ denotes a string of i copies of #.

Theorem 1 *Each complexity class in the canonical list, with the exception of PLOG, has complete languages under log-space many-one reducibility,* $\leq_m^L$.

Proof for the class P: Fix $\Sigma = \{0, 1\}$. We have shown in chapter 4 (example 5) that the characteristic class $P|\Sigma$ has an efficient universal acceptor $U^P = \{U_i^P : i = 0, 1, \ldots\}$ where each U_i^P accepts in time $n^{|i|}$. Define over the alphabet $\{0, 1, \#\}$ the language

$$L^P = \{i\#^m x : i, x \in \Sigma^*, m = |i| \cdot |x|^{|i|}, x \in L(U_i^P)\}.$$

We claim that L^P is P-complete under $\leq_m^L$. First we show that L^P is in P. This is witnessed by the acceptor M that, on input w, first checks that w has the form $i\#^m x$ for some i, x, m. Next it checks that $m = |i| \cdot |x|^{|i|}$, and if so, simulates U_i^P on x. The simulation takes time $O(m) = O(|w|)$. To show that L^P is P-hard, consider any language $(\Gamma, L) \in P$. Let h be the homomorphism that encodes each symbol of Γ as a unique binary string in Σ^* of length $\lceil \log |\Gamma| \rceil$. So $h(L)$ is a language over Σ; furthermore, it is easy to see that

$$h(L) \in P \iff L \in P.$$

So let $h(L)$ be accepted by U_i^P for some i. Consider the transducer T that on input x outputs $i\#^m h(x)$ where $m = |i| \cdot |h(x)|^{|i|}$. It is not hard to make T use $\log(|x|^{|i|}) = O_i(\log |x|)$ space to produce this output. Now x is in L iff $\mathrm{T}(x)$ is in L^P. Thus $L \leq_m^L L^P$. This shows that L^P is P-hard.

Sketch of the other cases: The proof for the class $NLOG$ uses the same technique as above: we use the existence of an efficient universal acceptor U^{NLOG} for $NLOG|\Sigma$ such that for each i, U_i^{NLOG} accepts in space $|i| \log n$. The complete language is

$$L^{NLOG} = \{i\#^m x : i, x \in \Sigma^*, m = |i| \cdot |x|^{|i|}, x \in L(U_i^{NLOG})\}.$$

The proof for the class $DEXPT$ uses the existence of an efficient universal acceptor U^{DXT} such that for each i, U_i^{DXT} accepts in time $|i|^n$ time. The complete language is

$$L^{DXT} = \{i\#^m x : i, x \in \Sigma^*, m = |i| \cdot |x|, x \in L(U_i^{DXT})\}.$$

The proofs for the other classes are similar. **Q.E.D.**

We make two remarks. First, the method unfortunately does not extend to $PLOG$. No complete problems for this class (under the usual reducibilities) are known. Second, the result is trivial for $DLOG$ because unrestricted log-space many-one reducibility is too powerful: it is easy to see that $DLOG$ is $\leq_m^L$-reducible to any non-trivial language. Following Hartmanis, we consider the one-way log-space many-one reducibilities, $\leq_m^{1L}$ (see section 2, chapter 4); the reader may verify that the construction for L^{NLOG} in the above proof carries over to give us a $DLOG$-complete language L^{DLOG} under $\leq_m^{1L}$-reducibility.

The next question we ask is whether classes such as

$$XTIME(n^k), XSPACE(\log^k n), XSPACE(n^k)$$

$(X = D, N)$ for each $k \geq 1$ have complete languages. The answer is yes, using a simple variation of the above proof. In particular, the characteristic class $XSPACE(\log^k n)$ has an efficient universal acceptor U' and hence it is not surprising to find complete languages for it. More precisely, we may assume that the universal acceptor U_i' accepts in space at most $|i| \log^k n$ for each i. Then a complete language for $XSPACE(\log^k n)$ is given by

$$L' = \{i\#^m x : x \in L(U_i'), m = |x|^{|i|}\}.$$

We make the observation that the language L^P in the proof of theorem 1 is actually in $DTIME(O(n))$, so it is in fact $DTIME(O(n))$-complete. To see this as a more general phenomenon, we next state a definition and lemma.

Definition. Let (Σ, L) be a language, $\#$ a symbol not in Σ, and f a number-theoretic function. Then the *f-padded version* (*with padding symbol $\#$*) of L is the language $L' = \{x\#^{f(|x|)} : x \in L\}$ over $\Sigma \cup \{\#\}$.

Lemma 2 *Let $k \geq 1$ be any fixed integer. Under the $\leq_m^L$-reducibility, we have:*

(i) *If a language L is $DSPACE(n^k)$-complete then L is $PSPACE$-complete.*

(ii) *If L is $PSPACE$-complete then the f-padded version of L is $DSPACE(n^k)$-complete, where $f(n) = n^h$ for some $h \geq 1$.*

Proof.

(i) It is sufficient to show that every language $L' \in PSPACE$ can be reduced to L. Assume L' is accepted by some U_i in space $f(n) = n^{|i|}$, where U is an efficient universal acceptor for the characteristic class $PSPACE$. To reduce L' to L, we construct the transducer that on input x outputs $t(x) = i\#x\#^{f(|x|)}$. Clearly the language $t(L') = \{t(x) : x \in L'\}$ is in $DSPACE(n) \subseteq DSPACE(n^k)$. Since L is $DSPACE(n^k)$-complete, we have $t(L')$ is many-one reducible to L via some log-space transformation r. Hence L' is many-one reducible to L via the functional composition $r \circ t$. Since the log-space transformations are closed under functional composition (chapter 4), we conclude that $r \circ t$ is log-space computable.

(ii) Since L is $PSPACE$-complete, let L be accepted by some M in space n^h for some $h \geq 1$. Let L' be the n^k-padded version of L for some $k \geq 1$. Clearly $L' \in DSPACE(n) \subseteq DSPACE(n^k)$. To show that L' is $DSPACE(n^k)$-hard, let $L'' \in DSPACE(n^k)$ be many-one reducible to L via some log-space transformation t. The reader may verify that L'' is many-one reducible to L' via the transformation r where $r(x) = t(x)\#^{|t(x)|^h}$. Clearly r is log-space computable. **Q.E.D.**

This tells us that the classes LBA, $DSPACE(n^k)$ and $PSPACE$ have essentially the same complete languages under log-space many-one reducibility.

The analog of the above lemma for $DTIME(n^k)$ can be proved in exactly the same way. Unfortunately, $DTIME(n^k)$ is not closed under log-space many-one reducibility so that we cannot apply the Basic Inclusion Lemma (chapter 4). The next result (from [4]) remedies this situation by considering the family **1FST** of transformations computed by 1-way finite state transducers (chapter 4, section 2):

Theorem 3 *Let $k \geq 1$, and $X = D$ or N. Let K be the class $XTIME(n^k)$ or $XSPACE(n^k)$.*

(i) *K is closed under $\leq_m^{1FST}$-reducibility.*

(ii) *There exists K-complete languages under $\leq_m^{1FST}$ reducibilities.*

Proof. (i) This is a simple exercise in machine construction. (ii) We just show the result for $K = XTIME(n^k)$. Consider the language L^k consisting of all words of the form

$$w = x_1\#y\#x_2\#\cdots\#y\#x_n \tag{5.1}$$

where $x_i, y \in \{0,1\}^*$, each x_i has the fixed length k and y is the encoding of a Turing acceptor M_y that is clocked to run for n^k steps (with the appropriate mode X) and M_y accepts $x_1x_2\cdots x_n$. We first show that L^k is hard for $XTIME(n^k)$. Let (Σ, L) be accepted by some Turing acceptor M_y in time n^k. Let $h : \Sigma \to \{0,1\}^*$ be any encoding of the symbols of Σ using binary strings with a fixed length, and define the transformation t, depending on h and M_y, such that for any $x = a_1a_2\cdots a_n$, $t(x) = h(a_1)\#y\#h(a_2)\#y\#\cdots\#y\#h(a_n)$ where y is an encoding of M_y that is clocked to run for at most n^k steps. Clearly $t \in \mathbf{1FST}$ and L is many-one reducible to L^k via t. It remains to show that L^k is in $XTIME(n^k)$. On input w, we can verify in linear time that w has the form given by (5.1). Then we simulate M_y. Because there is a copy of the machine encoding y next to each 'real input symbol' $h(a_i) = x_i$, it takes only $O(|y| + |x_1|)$ steps to simulate one step of M_y. This gives $O((|y| + |x_1|) \cdot n^k) = O(|w|^k)$ steps overall. By the linear speedup theorem, we can make this exactly $|w|^k$ steps. **Q.E.D.**

Similarly, we can show that reversal classes defined by polynomial or exponential complexity functions have complete languages [9]:

Theorem 4 *The classes $DREVERSAL(n^{O(1)})$ and $DREVERSAL(O(1)^n)$ have complete languages under log-space many-one reducibility.*

Proof. We show this for the case of $DREVERSAL(n^{O(1)})$. Using the existence of a efficient universal machine $\{U_1, U_2, \ldots\}$ for this characteristic class, we see that the language

$$\{i\#^m x : m = |x|^{|i|}, x \in L(U_i)\}$$

is complete. **Q.E.D.**

Natural Complete Languages. The complete languages generated in the above manner are artificial and it is of interest to obtain 'natural' complete problems. By natural problems we mean those arising in contexts that have independent interest, not just concocted for the present purpose. (Of course, naturalness is a matter of degree.) The advantage of natural complete problems is that they are an invaluable guide as to the inherent complexity of related natural problems. [1] We examine such languages in the remainder of this chapter. For each complexity class K studied

[1] Cf. comments in footnote 18, in chapter 1.

below, we first give a direct proof that a language L_0 is complete for K under $\leq_m^L$-reducibility. Subsequently, we may show any other languages L to be K-complete by showing $L_0 \leq_m^L L$, using the fact that $\leq_m^L$-reducibility is transitive.

Notation. We will conveniently use $[i..j]$ to denote the set $\{i, i+1, \ldots, j\}$ where $i \leq j$ are integers.

5.2 Complete Problems for Logarithmic Space

5.2.1 Graph Accessibility

The first problem shown to be complete for *NLOG* is the following. It was originally introduced as 'threadable mazes' by Savitch [28] but the present form is due to Jones [19].

Graph Accessibility Problem (GAP)

Given:	A pair $\langle n, G\rangle$ where G is an $n \times n$ adjacency matrix of a directed graph on the node set $[1..n]$.
Property:	There is a path from node 1 to node n.

To show that GAP is in *NLOG*, we describe a nondeterministic acceptor M that guesses a path through the graph as follows: on input x, M first verifies that x has the correct format representing the pair $\langle n, G\rangle$. Then it writes down '1' on tape 1 and on tape 2 makes a nondeterministic guess of some j, $1 \leq j \leq n$. In general, suppose tapes 1 and 2 contain the integers i and j (respectively). Then it verifies that $\langle i, j\rangle$ is an edge of G. If not, it rejects at once. It next checks if $j = n$. If so, it accepts; otherwise it copies j to tape 1 (overwriting the value i) and makes another guess k on tape 2. Now we have the pair $\langle j, k\rangle$ represented on tapes 1 and 2, and we may repeat the above verification and guessing process. Clearly the input is in GAP iff some sequence of guesses will lead to acceptance. The space required by M is $\log n$ where the input size is $O(n^2)$.

To show that GAP is *NLOG*-hard, let L be accepted by some nondeterministic machine N in space $\log n$. We reduce L to GAP using a transducer T that on input x computes the transformation $t(x)$ as follows: let $|x| = n$ and we may assume that each configuration of N that uses at most $\log n$ space on input x. is represented by integers in the range $[2..n^c - 1]$ for some integer $c > 0$. (Some integers in this range may not encode any configuration.) Then $t(x)$ is the encoding of $\langle n^c, G\rangle$ where the entries $G_{i,j}$ of the matrix G (representing a graph) is defined as follows: $G_{i,j} = 1$ iff one of the following holds:

(1) i and j encode configurations C_i and C_j (respectively) of M and $C_i \vdash C_j$.

(2) $i = 1$ and j represents the initial configuration.

(3) i represents an accepting configuration and $j = n^c$.

It is not hard to see that T can output the successive rows of G using only space $\log n$. Furthermore, x is accepted by N iff $t(x)$ is in GAP. This completes the proof.

5.2.2 Unsatisfiability of 2CNF formulas

For the next *NLOG*-complete problem, recall from chapter 3 that the kCNF formulas ($k \in \mathrm{N}$) are those with exactly k literals per clause.

Unsatisfiability of 2CNF Formulas (2UNSAT)

Given: A 2CNF formula F.

Property: F is unsatisfiable.

With respect to a 2CNF formula F, let us write $u \to v$ if $\{\bar{u}, v\} \in F$. Thus "$\to$" here is just "logical implication". Clearly, $u \to v$ iff $\bar{v} \to \bar{u}$. The reflexive, transitive closure of $\to$ is denoted $\stackrel{*}{\to}$. Thus $u \stackrel{*}{\to} v$ iff there is a sequence $u_1, u_2, \ldots, u_k$ ($k \geq 1$) of literals such that $u_1 = u$, $u_k = v$ and $u_i \to u_{i+1}$ for $i = 1, \ldots, k-1$. The literals u_i in the sequence need not be distinct. We note that *if I is a satisfying assignment for F and $u \stackrel{*}{\to} v$ then $I(u) = 1$ implies $I(v) = 1$.* It follows that if $u \stackrel{*}{\to} \bar{u}$ and I satisfies F then $I(u) = 0$. For any set of literals X, let the closure of X be $cl(X) = \{v : (\exists u \in X) u \stackrel{*}{\to} v\}$. Hence if I satisfies F and $I(u) = 1$ for all $u \in X$ then $I(v) = 1$ for all $v \in cl(X)$. We begin with the following characterization:

Theorem 5 *A 2CNF formula F is unsatisfiable iff there exists a literal u such that $u \stackrel{*}{\to} \bar{u}$ and $\bar{u} \stackrel{*}{\to} u$.*

Proof. If there is such a literal u. the above remarks makes it clear that F is unsatisfiable. Conversely, suppose there are no such literal. We will show that F is satisfiable. A set of literals is *consistent* if it does not contain both x and $\bar{x}$ for any variable x. We claim that for each variable x, either the set $cl(\{x\})$ is consistent or the set $cl(\{\bar{x}\})$ is consistent: otherwise let y and z be variables such that

$$\{y, \bar{y}\} \subseteq cl(\{x\}) \quad \text{and} \quad \{z, \bar{z}\} \subseteq cl(\{\bar{x}\})$$

Thus $x \stackrel{*}{\to} y$ and $x \stackrel{*}{\to} \bar{y}$. But $x \stackrel{*}{\to} \bar{y}$ implies $y \stackrel{*}{\to} \bar{x}$ so that transitivity implies $x \stackrel{*}{\to} \bar{x}$. A similar argument using $z, \bar{z}$ shows that $\bar{x} \stackrel{*}{\to} x$, contradicting our assumption that there are no such x.

We now define a sequence $U_0 \subseteq U_1 \subseteq \cdots \subseteq U_m$ (for some $m \geq 1$) of sets of literals: Let $U_0 = \emptyset$ (the empty set). Supposed U_i ($i \geq 0$) is defined. If for every variable x, either x or $\bar{x}$ is in U_i then we stop (i.e., set m to be i). Otherwise choose such a variable x and by the above observation, either $cl(\{x\})$ or $cl(\{\bar{x}\})$ is consistent. If $cl(\{x\})$ is consistent, set $U_{i+1} := U_i \cup cl(\{x\})$. Else, set $U_{i+1} := U_i \cup cl(\{\bar{x}\})$.

It is immediate that each U_i is closed, i.e., $cl(U_i) = U_i$. Suppose U_m is consistent. Then the assignment that makes each literal in U_m true is a satisfying assignment

for F. To see this, suppose $\{u, v\}$ is a clause in F and u is not in U_m. This means $\bar{u}$ is in U_m. But $\bar{u} \xrightarrow{*} v$, so v is in U_m. This shows that every clause is satisfied, as desired.

It remains to show the consistency of U_m. We show inductively that each U_i, $i = 0, \ldots, m$, is consistent. U_0 is clearly consistent. Next assume that U_i $(i \geq 0)$ is consistent but $U_{i+1} = U_i \cup cl(\{u\})$ is inconsistent. Say, $\{v, \bar{v}\} \subseteq U_{i+1}$. We may assume that $\bar{v} \in U_i$ and $v \in cl(\{u\})$. But $v \in cl(\{u\})$ implies $u \xrightarrow{*} v$, or equivalently, $\bar{v} \xrightarrow{*} \bar{u}$. Then $\bar{v} \in U_i$ implies $\bar{u} \in U_i$, contradicting our choice of u when defining U_{i+1}. **Q.E.D.**

From this lemma, we see that 2UNSAT is in *NLOG*: on input F, guess a literal u such that $u \xrightarrow{*} u$ witnesses the unsatisfiability of F. We then guess the sequence $u \to u_1 \to u_2 \to \cdots \to u_k \to u$. We need not store the entire sequence, just the first literal u and a current literal u_i. If we assume that each literal is encoded by a string of length $O(\log n)$ when F has n distinct variables, the space used is clearly $\log n$. We could impose this restriction on the encoding. However, it is not hard to see that even if no such restrictions were made, logarithmic space is sufficient. The idea is that any variable in F can be represented by a counter of size $\log n$ that points to some occurrence of that variable in F. The details are left as an exercise.

We now prove that 2UNSAT is *NLOG*-hard by reducing GAP to it. Let $\langle n, G\rangle$ be the input to GAP. We can construct the following CNF formula whose variables are denoted by the integers $1, \ldots, n$:

$$1 \wedge \bar{n} \wedge \bigwedge_{\langle i,j\rangle} (\bar{i} \vee j)$$

where $\langle i, j\rangle$ range over edges in G. Note that this is not quite a 2CNF formula because the first two clauses have only one literal each.[2] To turn a one variable clause $\{u\}$ to two variables, it is easy to introduce a new variable z so that $\bar{u} \to z \to u$. If we interpret the variable i to mean "node i is reachable from node 1", then this formula says that node 1 but not node n is reachable, and if $\langle i, j\rangle$ is an edge and node i is reachable then so is node j. Clearly, if there is a path from 1 to n then the formula is unsatisfiable. Conversely, if there are no paths from 1 to n then (exercise) it is easy to see that the formula is satisfiable with this assignment: (a) assign to false all those nodes that can reach node n, (b) assign to true all the remaining nodes. Finally, the fact that this formula can be constructed from $\langle n, G\rangle$ in logarithmic space completes the proof.

Based on our intuition from the satisfiability problem SAT, it is perhaps curious that it is 2UNSAT rather than 2SAT that is *NLOG*-complete.[3]

[2]Since we regard clauses as sets of literals, it is no good saying that the clause with one literal can be made into a 2-literal clause by repeating the single literal.

[3]Nevertheless, this switch is typical when we change from a time class to space class.

5.2.3 Associative Generator Problem

The next problem concerns a simple algebraic system with one associative binary operation denoted $\oplus$. Without loss of generality, the elements of the system are $[1..n]$ and let the multiplication table of $\oplus$ be given by an $n \times n$ matrix $T = (T_{i,j})$. Thus $T_{i,j} = k$ means $i \oplus j = k$. For any set $S \subseteq [1..n]$, the $\oplus$-closure of S is the smallest set containing S and closed under $\oplus$. We should keep in mind that the elements in $[1..n]$ represents abstract elements, not integers.

Associative Generator Problem (AGEN)

Given:	$\langle n, T, S, w\rangle$ where T is a multiplication table for a binary operation $\oplus$ over $[1..n]$, $S \subseteq [1..n]$ and $w \in [1..n]$.
Property:	The operation $\oplus$ is associative and w is in the $\oplus$-closure of S.

We sketch the proof. To show that AGEN is in *NLOG*, we note that it is easy to check in logarithmic space that an input has the form $\langle n, T, S, w\rangle$ and that T is associative. Now w is in the $\oplus$-closure of S iff it can be expressed as $w = x_1 \oplus x_2 \oplus \cdots \oplus x_k$. It is not hard to see that the shortest such expression has length of k at most n. Hence it is easy to nondeterministically accept AGEN by guessing successive values of x_i, for $i = 1, \ldots, n$, and only keeping track of the partial sum $x_1 \oplus x_2 \oplus \cdots \oplus x_i$.

Next, we show that GAP is reducible to AGEN. Let $\langle n, G\rangle$ be the input to GAP. We describe the AGEN instance $\langle m, T, S, w\rangle$ as follows: Choose $m = 1 + n + n^2$. Instead of describing the table T in terms of $[1..m]$, it is more intuitive to interpret the elements of $[1..m]$ as the set

$$X = [1..n] \cup ([1..n] \times [1..n]) \cup \{\infty\}$$

where ∞ is a new symbol. Hence $|X| = m$. We can make a correspondence between X and $[1..m]$ as follows: Each $i \in [1..n]$ corresponds to itself in $[1..m]$; the pair $\langle i, j\rangle \in [1..n] \times [1..n]$ corresponds to $i + jn$ in $[1..m]$ and finally ∞ corresponds to m. The table T is now described by the following rules:

For all $i, j, k, u, v \in [1 \ldots n]$,

$$i \oplus \langle j, k\rangle = \begin{cases} k & \text{if } i = j \text{ and } \langle j, k\rangle \text{ is an edge of } G \\ \infty & \text{else.} \end{cases}$$

$$\langle i, j\rangle \oplus \langle u, v\rangle = \begin{cases} \langle i, v\rangle & \text{if } j = u \\ \infty & \text{else.} \end{cases}$$

In all other cases, the value of $x \oplus y$ is ∞. It is easy to see that $\oplus$ is associative and there is a path $(1, x_1, \ldots, x_k, n)$ from 1 to n in G iff $1 \oplus \langle 1, x_1\rangle \oplus \cdots \oplus \langle x_k, n\rangle = n$ holds. The desired AGEN instance $\langle m, T, S, w\rangle$ is now evident: m and T has been

described; S is the set consisting of 1 and all pairs $\langle i, j\rangle$ representing edges of G; w is chosen to be n. Finally, one verifies that this AGEN instance can be computed in logarithmic space from $\langle n, G\rangle$.

5.2.4 Deterministic Logarithmic Space and below

To get a meaningful complete language for $DLOG$, we weaken our reducibility as in section 1. We restrict the input head on the transducers to be one-way. A natural restriction of GAP that is $DLOG$-complete is to make the outdegree of each node at most 1. However, we must also be careful about the actual encoding of the problem. In particular, we can no longer assume the adjacency matrix representation of graphs:

Deterministic Graph Accessibility Problem (1GAP)

Given: $\langle n, G\rangle$ as in GAP except that each vertex of G has outdegree at most 1 and G is encoded by a sequence of its edges.

Property: There is a path from node 1 to n.

It is easy to see that this problem is in $DLOG$. We leave it as an exercise to reduce an arbitrary language in $DLOG$ to 1GAP: essentially we cycle through all configurations C and for each C, we output the 'edge' pair (C, C') where C' is the successor of C, $C \vdash C'$. Indeed, this proof shows that the following variant of 1GAP remains complete: where we insist that the inputs $\langle n, G\rangle$ satisfy the property that the list of edges of G be topologically sorted (i.e., all edges entering a node must be listed before edges exiting from that node).

Let us briefly consider the power of one-way log-space many-one reducibility [16]. It turns out that many natural problems which are complete for canonical classes (under $\leq_m^L$) remain complete under $\leq_m^{1L}$. This is fortuitous because it is easily shown that there are languages that are complete for P (for instance) under $\leq_m^L$ but which are not P-complete under $\leq_m^{1L}$. See Exercises.

While there is less interest sublogarithmic space complexity, it should be noted that complete problems can be defined for any nondeterministic space bound between $\log\log n$ and $\log n$ space. Monien and Sudborough define the so-called 'bandwidth limited' versions of the GAP problem [23]. To understand this, we say the *bandwidth* of a graph G is the smallest $m \geq 0$ such that each edge (i, j) satisfies $|i - j| \leq m$. We must also be careful about how to encode such graphs. Let us encode graphs on the vertices $\{1, 2, \ldots, n\}$ by a sequence of the adjacency lists of each vertex. The adjacency list for vertex i has the form $(i, m(i, 1), m(i, 2), \ldots, m(i, d_i))$ where d_i is the out-degree and the edges exiting from i go to $i + m(i, j)$ for each j. Thus $m(i, j)$ is an incremental positive or negative value, relative to i. Naturally the bandwidth of G is an upper bound on their absolute value. Now for any complexity function $f(n) \leq \log n$ define the language $GAP(f)$ to encode the family of graphs G such that if the vertices are $\{1, \ldots, n\}$ then the bandwith of G is at

most $f(n)$, and such that there is a path from node 1 to node n. Then it can be shown that $GAP(f)$ is complete for $NSPACE(f)$ under $\leq_m^L$-reducibility. We may further weaken the reducibility so that the transducers use only $f(n)$ space, provided we encode the languages even more compactly: omit the first number i in the list $(i, m(i,1), \ldots, m(i,d_i))$, and so i is implicit from the order of occurrence of its list.

5.3 Complete Problems for P

5.3.1 Unit Resolution

We begin by showing a problem based on the *resolution proof system* to be P-complete. This is actually part of a theme started in the previous section where we show that 2UNSAT is *NLOG*-complete problem. That is, we will show a family of related problems arising from theorem proving, each of which is complete for some class in our canonical. The famous SAT problem (chapter 3) is part of this family.

The basic concept of resolution proofs is simple: Two clauses C and C' are *resolvable* if there is a literal u such that $u \in C$ and $\bar{u} \in C'$. Their *resolvent* is the clause $C'' = (C \cup C') - \{u, \bar{u}\}$. This rule for obtaining the resolvent is called the *resolution rule* (also known as *annihilation rule* or *cut rule*), with the literals u and $\bar{u}$ being *annihilated*. A *unit clause* is a clause with one literal. If either C or C' is a unit clause, we say their resolvent C'' is a *unit resolvent* of C and C'. Note that if both C and C' are unit clauses then their resolvent is the empty clause which we denote by $\square$ (not to be confused with the blank symbol, but the context should make it clear). By definition, the empty clause is unsatisfiable. Let F be a CNF formula in clause form. A *deduction* (resp. *unit deduction*) of a clause C_m from F is a sequence of clauses $C_1, \ldots, C_m$ where each C_i is either in F or is the resolvent (resp. unit resolvent) of two previous clauses in the sequence. The fundamental result on resolution is the following:

Resolution Theorem. *F is unsatisfiable iff there is a deduction of $\square$ from F.*

The proof is an exercise; we do not use the theorem in the following. The following P-complete problem was introduced by Jones and Laaser [18]:

Unit Resolution for 3CNF formulas (3UNIT)

Given: A 3CNF formula F.

Property: $\square$ can be deduced from F using unit deduction.

We first show that 3UNIT is in P. Consider the following algorithm that takes as input a CNF formula F. If $\square \in F$ then we accept at once. So assume otherwise. The algorithm maintains three sets of clauses G, G' and H. Initially, G' is empty, G consists of all the unit clauses in F and H is $F - G$. In general, the sets G, G'

will consist of unit clauses only; hence we may sometimes conveniently regard them as a set of literals. Also $G \cap G' = \emptyset$. The algorithm executes the following loop:

While $G \neq \emptyset$ do

1. Choose any $u \in G$. Let $G := G - \{u\}$; $G' := G' \cup \{u\}$.
2. If $\bar{u} \in G'$ then resolve $\{u\}$ and $\{\bar{u}\}$ to obtain $\square$ and accept at once.
3. For each clause C in H: if $\{u\}$ and C are resolvable, and their resolvent $C' = C - \{\bar{u}\}$ is not in $G \cup G' \cup H$ then add C' to G or to H according to whether C' is a unit clause or not.

If the algorithm does not accept within the while-loop, then it rejects. It is important to realize that in line 3, the clause C is not removed from H (the Exercises explain why this is necessary for the correctness of resolution).

Correctness: We must show that the algorithm accepts iff $F \in$ 3UNIT. Note that new clauses are generated only in line 3 using the unit resolution rule; also, all clauses in $G' \cup G \cup H$ come from the original set F or are unit deducible from F. Hence if the algorithm accepts in line 2 then F is in 3UNIT. To prove the converse, suppose

$$C_0, C_1, \ldots, C_m \quad (\text{where } C_m = \square) \tag{5.2}$$

is a unit deduction of $\square$. We will show that the algorithm accepts. It is crucial to note that step 3 is specified in such a way that each literal u is put into G at most once. Thus the number of iterations in the while-loop is at most the number of literals appearing in the formula F. Let G_i $(i = 0, 1, \ldots)$ denote the set of clauses in G at the end of the ith iteration of the loop. G'_i and H_i are similarly defined, and let $E_i := G_i \cup G'_i \cup H_i$. Clearly, $E_i \subseteq E_{i+1}$. CLAIM: each C_j in the unit deduction (5.2) belongs to some E_i. In particular C_{m-1} appears in the E_i of the last iteration, and the empty clause can be derived. So the algorithm accepts.

Complexity: As noted above, the number iterations of the while-loop is linear in the size of F. It therefore suffices to show that each iteration takes polynomial time. Note that if a clause of F has k literals, then the clause can potentially spawn 2^k subclauses. But since F is a 3CNF formula, the total number of new clauses is only linear in the number of original clauses. This in turn bounds the time to do each iteration. Thus the time of the algorithm for 3UNIT is polynomial.

We now show that 3UNIT is P-hard. The proof is similar to that for Cook's theorem. Let M accepts in deterministic time n^k for some k. Again we may assume that M is a simple Turing acceptor. We may further assume that M never moves left of its original head position (in cell 1) and that if it accepts at all then it returns to cell 1 and writes a special symbol a_0 there just before accepting. For any input x, we describe a 3CNF formula $f(x)$ such that x is accepted by M iff $f(x) \in$ 3UNIT. Let

$$C_0, C_1, \ldots, C_{n^k} \tag{5.3}$$

be the computation path of M on input x. As usual, if M halted before n^k steps we assume the last configuration is repeated in this path, and if M uses more than n^k steps, we will prematurely truncate the sequence. It is convenient to encode a configuration as a word of length $m := 1 + 2n^k$ in $\Sigma^* \cdot [Q \times \Sigma] \cdot \Sigma^*$ where Σ are the tape symbols (including $\Box$) of M and Q are the states and $[Q \times \Sigma]$ a new set of symbols of the form $[q, a]$ where $q \in Q, a \in \Sigma$. Thus a word of the form $w_1[q, a]w_2$ (where $w_i \in \Sigma^*$) represents the configuration whose tape contents (flanked by blanks as needed) are $w_1 a w_2$ with the machine scanning symbol a in state q. We next introduce the Boolean variables $P_{i,t}^a$ that stands for the following proposition:

"symbol a is in the ith cell of configuration C_t".

A similar meaning is accorded the Boolean variable $P_{i,t}^{[q,a]}$ where $[q, a] \in [Q \times \Sigma]$. Let $x = a_1 a_2 \cdots a_n$ where each $a_j \in \Sigma$. The formula $f(x)$ is a conjunction of the form $F_0 \wedge F_1 \wedge \cdots \wedge F_{m+1}$. The last formula F_{m+1} is special and is simply given by:

$$F_{m+1} \ : \ \neg P_{1,m}^{[q_a, a_0]}$$

The formula F_t $(t = 0, \ldots, m)$ is a 3CNF formula asserting that conditions that the above variables must satisfy if they encode the configuration C_t in (5.3). Thus the first configuration C_0 is

$$F_0 \ : \ P_{1,0}^{[q_0,a_1]} \wedge P_{2,0}^{a_2} \wedge \cdots \wedge P_{n,0}^{a_n} \wedge \left(\bigwedge_{i=n+1}^{m} P_{i,0}^{\Box} \right)$$

We indicate the form of the remaining formulas. Let

$$\partial : (\Sigma \cup [Q \times \Sigma])^3 \to \Sigma \cup [Q \times \Sigma]$$

be the function that encodes the transition table of M. Roughly, $\partial(a, b, c) = b'$ means that the symbol b in a cell whose left and right neighbors are a and c respectively will turn to b' in the next step (with understanding that the 'symbol' in a cell could mean a pair of the form $[q, a']$). For example, suppose the M has the instruction that says "on scanning symbol b in state q, write the symbol b', move the head to the right and enter the state q'". We would then have the following:

$$\partial([q, b], a, c) = [q', a], \quad \partial(a, [q, b], c) = b', \quad \partial(a, c, [q, b]) = c.$$

Also, clearly $\partial(a, b, c) = b$ for all a, b, c in Σ. Now it is easy to understand the following definition of F_t $(t = 1, \ldots, m)$:

$$F_t : \bigwedge_i \left(\bigwedge_{a,b,c} (\neg P_{i-1,t-1}^a \vee \neg P_{i,t-1}^b \vee \neg P_{i+1,t-1}^c \vee P_{i,t}^{\partial(a,b,c)}) \right).$$

We now claim:

(i) The unit clause $P^a_{i,t}$ is unit deducible from $f(x) = F_1 \wedge \cdots \wedge F_m$ iff a is the symbol or [state, symbol]-pair at cell i of C_t.

(ii) Furthermore, no other unit clause can be deduced.

The claim is clearly true for initial configuration, $t = 0$. The result can be established by a straightforward proof using induction on t; we leave this to the reader.

We now show that x is accepted by M iff $\square$ is unit deducible from $f(x)$. Claim (i) shows that x is accepted iff $P^{[q_a,a_0]}_{1,m}$ is deducible. But this clause can be unit resolved with F_{m+1} to obtain $\square$. Next claim (ii) easily implies that there are no other ways to deduce $\square$. This shows the correctness of the described formula. It is easy to construct $f(x)$ in logarithmic space.

Note that each of the clauses in the proof uses at most 4 literals. Using techniques similar to that in proving 3SAT *NP*-complete, we can replace these by clauses with exactly 3 literals per clause. **Q.E.D.**

5.3.2 Path Systems

We now consider a problem that is the first one to be proved complete for P. It was invented by Cook [7] who formulated it as an abstraction of the proof method of both Savitch's theorem (chapter 2) as well as a well-known theorem that context-free languages are in $DSPACE(\log^2 n)$ (see Exercises for this connection.)

Definition. A *path system* is a quadruple $\langle n, R, S, T\rangle$ where $n > 0$ is an integer, $R \subseteq [1..n] \times [1..n] \times [1..n]$ is a relation over $[1..n]$, and S, T are subsets of $[1..n]$. Each integer $u \in [1..n]$ represents a 'node' and node u is said to be *accessible* if $u \in T$ or if there exist accessible nodes v, w such that $\langle u, v, w\rangle$ is in R. The path system is *solvable* if some node in S is accessible.

Path System Accessibility (PSA)

Given: A path system $\langle n, R, S, T\rangle$.

Property: The system is solvable.

We first claim that PSA is in P: consider the Turing acceptor that begins by computing the sets $T_0, T_1, \ldots$ of nodes in stages. In the initial stage, it computes $T_0 = T$. In stage $i \geq 1$, it computes T_i consisting of those new nodes that are accessible from T_{i-1}. It stops when no more new nodes are accessible. Clearly there are at most n stages and each stage takes time $O(n^3)$. So the whole algorithm is $O(n^4)$. Note that the size of the input can be much smaller than n (in fact as small as $O(\log n)$) if not all the integers in $[1..n]$ occur in R, S, T. Hence the algorithm could be exponential in the input size. However, with a more careful implementation (cf. the demonstration that 2UNSAT is in *NLOG*), we can easily ensure that the

time is $O(m^4)$ where $m \leq n$ is the number of nodes that actually occur in the path system description. Hence our algorithm is polynomial time.

We now show that PSA is P-hard by reducing 3UNIT to it. Given a 3CNF formula F, let X be the set of all clauses that are subsets of clauses of F. We describe the corresponding path system $\langle n, R, S, T\rangle$. Let $n = |X|$ and let each integer $i \in [1..n]$ represent a clause of X. R consists of those triples $\langle i, j, k\rangle$ of clauses in X such that i is the unit resolvent of j and k. Let the set T be equal to F; let S consist of just the empty clause $\square$. It is easy to see that $\langle n, R, S, T\rangle$ is solvable if and only if F is in 3UNIT. This completes our proof that PSA is P-complete.

5.3.3 Non-associative Generator Problem

The next problem is closely related to the $NLOG$-complete problem AGEN. Basically, the AGEN turns into a P-complete problem when we remove the associative property. We now use $\otimes$ to denote the non-associative binary operation.

Generator Problem (GEN)

Given:	Given $\langle n, T, S, w\rangle$ as in AGEN with T representing the multiplication table of a binary operation $\otimes$ over $[1..n]$.
Property:	w is in the $\otimes$-closure of S.

The proof that GEN is in P is straightforward. To show that it is P-hard we can reduce 3UNIT to it, using a proof very similar to that for the PSA problem.

5.3.4 Other Problems

The following problem is due to Ladner [20]. Boolean circuits (which will be systematically treated in chapter 10) are directed acyclic graphs such that each node has in-degree of zero or two. The nodes of indegree zero are called *input* nodes and these are labeled by the integers $1, 2, \ldots, n$ if there are $n \geq 0$ input nodes. The other nodes, called *gates*, are each labeled by some two-variable Boolean function. If these Boolean functions are *logical-and*'s ($\wedge$) and *logical-or*'s ($\vee$) only, then the circuit is *monotone*. An *assignment* I to C is a function $I : [1..n] \to \{0, 1\}$. In a natural way, for each node v of G, we can inductively define a Boolean value $val_C(v, I)$.

Monotone Circuit Value Problem (MCVP)

Given:	A monotone Boolean circuit C with a distinguished node u, and an assignment I to the input variables of C.
Property:	$val_C(u, I) = 1$.

Ladner originally proved the P-completeness of the (general) circuit value problem (CVP) in which the circuit is not restricted to be monotone; the refinement to

monotone circuits is due to Goldschlager [12]. Goldschlager also shows that CVP (not MCVP) remains P-complete when we restrict the underlying graph of the circuits to be planar. Closely related to MCVP is this GAP-like problem:

AND-OR Graph Accessibility Problem (AND-OR-GAP)

Given: A directed acyclic graph G on vertices $[1..n]$, where each vertex is labeled by either 'AND' or 'OR'.

Property: Node n is accessible from node 1. Here a node i is accessible from another set S of nodes if either $i \in S$ (basis case) or else i is an AND-node (resp. OR-node) and all (resp. some) of the predecessors of i are accessible from S (recursive case).

Another P-complete problem is:

Rational Linear Programming problem (RLP)

Given: An $m \times n$ matrix A and an m-vector $\mathbf{b}$ where the entries of $A, \mathbf{b}$ are rational numbers.

Property: There is a rational n-vector $\mathbf{x}$ such that $A\mathbf{x} \geq \mathbf{b}$.

The fact that RLP is in P is a result of Khachian as noted in chapter 3; that the problem is P-hard is due to Dobkin, Reiss and Lipton[8]. The RLP problem has a famous history because of its connection to the simplex method and because it was originally belonged to the few problems known to be in $NP \cap co\text{-}NP$ that is neither known to be in P nor NP-complete.

Goldschlager, Shaw and Staples [11] have shown that a formulation of the well-known problem Maximum Flow in graphs is P-complete.

Adachi, Iwata and Kasai [1] have defined natural problems that are complete for $DTIME(n^k)$ for each $k \geq 1$.

5.4 Complete Problems for *PSPACE*

5.4.1 Word Problems

A rich source of natural problems with high complexity arises in the study of *extended regular expressions*. Meyer and Stockmeyer [21, 31, 32] and Hunt [17] were among the first to exploit such results. Given an alphabet Σ, the set of extended regular expressions over Σ is a recursively defined set of strings over the alphabet Σ together with the following 9 additional symbols

$$\lambda, +, \cdot, \cap, ^2, ^*, \neg,), ($$

which we assume are not in Σ. Each symbol in $\Sigma \cup \{\lambda\}$ will be called an *atom*. An *extended regular expression* (over Σ) is either an atom or recursively has one of the following forms:

$$(\alpha + \beta), (\alpha \cdot \beta), (\alpha \cap \beta), (\alpha)^2, (\alpha)^*, \neg(\alpha)$$

where α and β are extended regular expressions. Each expression α over Σ denotes a language $(\Sigma, L(\alpha))$ defined as follows: if α is the atom $a \in \Sigma \cup \{\lambda\}$ then $L(\alpha)$ is the language consisting of the single word[4] a; otherwise

Union:	$L(\alpha + \beta) = L(\alpha) \cup L(\beta)$
Concatenation:	$L(\alpha \cdot \beta) = L(\alpha) \cdot L(\beta)$
Intersection:	$L(\alpha \cap \beta) = L(\alpha) \cap L(\beta)$
Squaring:	$L(\alpha^2) = L(\alpha) \cdot L(\alpha)$
Kleene-star:	$L(\alpha^*) = L(\alpha)^* = \bigcup_{i \geq 0} L(\alpha)^i$
Complement:	$L(\neg\alpha) = \Sigma^* - L(\alpha)$

Note that if α is regarded as an expression over a different alphabet Γ (provided all the atoms appearing in the expression α are in Γ) it would denote a different language; in practice, the context will make clear which alphabet is meant or else the specific alphabet is irrelevant to the discussion.

Notations. We often omit the symbol for concatenation (writing '$\alpha\beta$' for '$\alpha \cdot \beta$'). Noting that all our binary operators are associative, and assuming some precedence of operators (unary operators precede binary ones and concatenation precedes union and intersection) we can omit some of the parentheses that might otherwise be needed. If $S = \{a_1, \ldots, a_m\}$ is a set of atoms, we often use the meta-symbol S as a shorthand for the expression $a_1 + a_2 + \cdots + a_m$. If α is any expression and $k \geq 1$ (k may be an integer expression), then we write α^k as a short hand for concatenating k copies of α (note the possible confusion of this notation with applications of the squaring operator if k written as a power of 2; the context will clarify which is meant).

The *size* $|\alpha|$ of an extended regular expression α is defined recursively as follows: if α is an atom then $|\alpha| = 1$; otherwise

$$|\alpha + \beta| = |\alpha \cdot \beta| = |\alpha \cap \beta| = 1 + |\alpha| + |\beta|;$$

and

$$|\alpha^2| = |\alpha^*| = |\neg\alpha| = 1 + |\alpha|.$$

Example 1 The following expression (of size 22) denotes the set of those words over $\Sigma = \{a, b, c\}$ of length between 4 and 8 beginning with a but not terminating with b:

$$a(a + b + c)^2((a + b + c + \lambda)^2)^2(a + c)$$

[4]We deliberately abuse notation here, by confusing the symbol 'λ' in extended regular expressions with the usual notation for the empty word; of course it comes to no harm since λ just stands for itself in this sense. Also, we really ought to use '$\cup$' for union for consistency with '$\cap$' for intersection; but it seems that the asymmetric choice has better visual aesthetics.

■

Let Ω be a subset of the extended regular operators $\{+, \cdot, \cap, ^2, ^*, \neg\}$. Then a Ω-*expression* is a extended regular expression that uses only the operators in Ω. Then the following *word problems* are defined:

Inequality:	$\text{INEQ}(\Sigma, \Omega) = \{(\alpha, \beta) :$ α and β are Ω-expressions over Σ and $L(\alpha) \neq L(\beta)\}$
Fullness:	$\text{FULL}(\Sigma, \Omega) = \{\alpha : \alpha$ is an Ω-expression and $L(\alpha) = \Sigma^*\}$
Emptiness:	$\text{EMPTY}(\Sigma, \Omega) = \{\alpha : \alpha$ is an Ω-expression and $L(\alpha) = \emptyset\}$
Membership:	$\text{MEMBER}(\Sigma, \Omega) = \{(x, \alpha) : \alpha$ is an Ω-expression and $x \in L(\alpha)\}$

In the presence of negation, $\neg \in \Omega$, it is evident that the fullness and emptiness problems are equivalent. Furthermore, the complement of the emptiness and fullness problems are special cases of the inequality problem whenever there is an Ω-expression denoting $\emptyset$ or Σ^* (resp.). We will mainly be showing the complexity of inequality and fullness problems in this and the next section. If Σ is $\{0, 1\}$ we just denote these problems by MEMBER(Ω), INEQ(Ω), etc. If $\Omega = \{\omega_1, \ldots, \omega_k\}$ then we also write INEQ($\omega_1, \ldots, \omega_k$), FULL($\omega_1, \ldots, \omega_k$), etc., instead of INEQ($\{\omega_1, \ldots, \omega_k\}$), FULL($\{\omega_1, \ldots, \omega_k\}$), etc. . We begin by showing that the complexity of these problems does not really depend on Σ provided $|\Sigma| \geq 2$:

Lemma 6 *Let Ω be any set of operators and $|\Sigma| \geq 2$.*

(i) *If Ω contains the concatenation operator then* $\text{INEQ}(\Sigma, \Omega) \equiv_m^{1FST} \text{INEQ}(\Omega)$.

(ii) *If Ω contains $\{+, \cdot, ^*\}$ then* $\text{FULL}(\Sigma, \Omega) \equiv_m^{1FST} \text{FULL}(\Omega)$.

Proof. (i) Recall the $\leq_m^{1FST}$-reducibility from chapter 4 (section 1). It is immediate that $\text{INEQ}(\Omega) \leq_m^{1FST} \text{INEQ}(\Sigma, \Omega)$. Conversely, to show $\text{INEQ}(\Sigma, \Omega) \leq_m^{1FST} \text{INEQ}(\Omega)$, we use the usual coding $h : \Sigma \to \{0, 1\}^*$ of a general alphabet by binary strings over $\{0, 1\}$ of fixed length k (for some $k \geq 2$). Then each Ω-expression α over Σ is systematically transformed to an Ω-expression $H(\alpha)$ over $\{0, 1\}$ simply by replacing each occurrence in α of the atom $a \in \Sigma$ by (the $\{\cdot\}$-expression denoting) $h(a)$. It is also easy to see that $(\alpha, \beta) \in \text{INEQ}(\Sigma, \Omega)$ iff $(H(\alpha), H(\beta)) \in \text{INEQ}(\Omega)$.

(ii) Again, one direction is trivial. To show $\text{FULL}(\Sigma, g) \leq_m^{1FST} \text{FULL}(\Omega)$, let h be the coding in (i) and let $C = \{h(a) : a \in \Sigma\} \cup \{\lambda\} \subseteq \{0, 1\}^*$. Let α be an Ω-expression over Σ, and let $H(\alpha)$ be its transformation as in (i). Clearly $L(H(\alpha)) \subseteq C^*$, with equality iff $L(\alpha) = \Sigma^*$. If we can write an Ω-expression β over $\{0, 1\}$ such that $L(\beta) = \Sigma^* - C^*$ then we see that

$$L(\alpha) = \Sigma^* \iff L(H(\alpha) + \beta) = \{0, 1\}^*.$$

It is easy to construct such an expression:

$$\beta : ((0+1)^k)^* \cdot ((0+1+\lambda)^k - C) \cdot ((0+1)^k)^*$$

where k is the length of the codes $h(a)$, $a \in \Sigma$, and the subexpression '$((0+1+\lambda)^k - C)$' is really a shorthand for an explicit enumeration of the words in the indicated set. These expressions involve only $\{+, \cdot, ^*\}$. **Q.E.D.**

Although it is convenient when proving upper bounds to assume a binary alphabet, it is easier to use the general alphabet Σ when proving lower bounds (i.e. hardness results). Below we shall switch between these two versions of the problem without warning.

5.4.2 Fullness Problem for Regular Expressions

Our first result is from Stockmeyer [32]:

Theorem 7 FULL$(+, \cdot, ^*)$ *is complete for the class LBA = NSPACE(n).*

Note that the $\{+, \cdot, ^*\}$-expressions are just the standard *regular expressions* of finite automata theory. In the following proof, we shall often exploit the fact that *NSPACE*(n) is closed under complementation. By lemma 2, we conclude:

The fullness problem for regular languages is complete for PSPACE.

Recall that a finite automaton is a 0-tape Turing acceptor with a 1-way input tape that runs in real-time ($n+1$ steps); the *size* of the automaton is the number of tuples in its transition table. Clearly the size is at most $|Q|^2(|\Sigma| + 1)$ where Q is the set of states and Σ the input alphabet (the '+1' comes from the need to consider rules for the blank symbol). To show that the fullness problem for regular expressions is in *NSPACE*(n), we first show:

Lemma 8 *For any regular expression α over Σ, we can construct in polynomial time and linear space a nondeterministic finite automaton that accepts the language $(\Sigma, L(\alpha))$; furthermore the automaton has $\leq \max\{4, |\alpha|\}$ states.*

Proof. We construct the transition table δ^α corresponding to α. Denote the start and accepting states for δ^α by q_0^α and q_a^α. The transition table of a nondeterministic finite automaton can be represented as a set of triples of the form $\langle$*current-state, symbol, next-state*$\rangle$. Furthermore, we assume that for any tuple $\langle q, b, q' \rangle$ in δ^α,

$$q' = q_a^\alpha \Longrightarrow b = \square.$$

In other words, all transitions into the accept state occur after the first blank symbol $\square$ is read. A state q is called *penultimate* if there is a transition from q to the accept state q_a^α. The table δ^α is defined by induction on the size of α:

(1) Suppose α is an atom b: if $b = \epsilon$ then the table δ^α consists of the single triple:

$$\langle q_0^\alpha, \square, q_a^\alpha \rangle.$$

If $b \in \Sigma$ then the table consists of two triples:

$$\langle q_0^\alpha, b, q \rangle, \langle q, \square, q_a^\alpha \rangle$$

for some state q.

(2) If α is $\beta + \gamma$ then first form the union $\delta^\beta \cup \delta^\gamma$. We then replace (in the triples) the start states of the original automata for β and γ with the start state for α, and do the same for the accept states.

(3) If α is $\beta \cdot \gamma$ then we again form the union $\delta^\beta \cup \delta^\gamma$, and do the following: if q is a penultimate state of δ^β and δ^γ contains the triple $\langle q_0^\gamma, b, q' \rangle$ then add the triple $\langle q, b, q' \rangle$ to δ^α. This ensures that after the automaton has seen a word in $L(\beta)$, it can continue to try to recognize a word in $L(\gamma)$. Also replace the start state of δ^β by the start state q_0^α and the accept state of δ^γ by the accept state q_a^α. Triples containing the accept state of δ^β and the start state of δ^γ are deleted.

(4) If α is β^* then we first take the table for β and replace its start and accept states with that for α. If q is a penultimate state of δ^β and $\langle q_0^\beta, b, q' \rangle \in \delta^\beta$ then add the triple $\langle q, b, q' \rangle$ to δ^α (this ensures that that the automaton accepts arbitrarily many copies of words in $L(\beta)$). Finally, add the new triple $\langle q_0^\alpha, \square, q_a^\alpha \rangle$ (this allows the empty string to be accepted). Thus δ^α has the same number of states as δ^β

The reader can easily show that the constructed transition table δ^α accepts $L(\alpha)$. The number of states is at most $|\alpha|$ when $|\alpha| \geq 4$. A case analysis shows that when $|\alpha| \leq 3$, δ^α has at most 4 states. The automaton δ^α can be constructed in linear space and polynomial time from α. **Q.E.D.**

We now show that FULL$(+, \cdot, ^*)$ is in *co-NSPACE*(n). It is enough to show how to accept the complement of the language FULL$(+, \cdot, ^*)$ in nondeterministic linear space. On input a regular expression α, we want to accept iff $L(\alpha) \neq \{0,1\}^*$. First we construct δ^α and then try to guess a word $x \notin L(\alpha)$: nondeterministically guess successive symbols of x and simulate *all computation paths* of the automaton on the guessed symbol. More precise, we guess successive symbols of x and keep track of the set S of all the states that can be reached by the automaton δ^α after reading the symbols guessed to this point. It is easy to see how to update S with each new guess, and since S has at most $2|\alpha|$ states, linear space suffices. When we are finished with guessing x, we make the next input symbol be the blank symbol $\square$ and update S for the last time. We accept if and only if S *does not contain* the accept state. To show the correctness of this construction, we see that (i) if there

exists such an x then there exists an accepting path, and (ii) if there does not exist such an x then no path is accepting. This proves $\mathrm{FULL}(+,\cdot,^*) \in \textit{co-NSPACE}(n)$.

To show that the fullness problem is *co-NSPACE*(n)-hard, let M be a nondeterministic simple Turing acceptor accepting in space n. We may assume that M never moves left of the cell 1 and that M enters the accepting state immediately after writing a special symbol a_0 in cell 1. For each input x of M we will construct a regular expression $E(x)$ over some alphabet Δ (see below) such that $x \notin L(\mathrm{M})$ iff $L(E(x)) = \Delta^*$. Let $t(n) = O_{\mathrm{M}}(1)^n$ be an upper bound on the running time of M on inputs of length $n = |x|$. Let

$$C_0 \vdash C_1 \vdash \cdots \vdash C_m \quad (m = t(n))$$

be a computation path on input x where as usual we assume that the last configuration is repeated as often as necessary if M halts in less that m steps. Let $I = \{1, \ldots, |\delta(\mathrm{M})|\}$ where $\delta(\mathrm{M})$ is the transition table of M. We assume that each C_i is encoded as a word w_i of length $n+2$ in

$$\Sigma^* \cdot [Q \times \Sigma \times I] \cdot \Sigma^*$$

where Σ and Q are the set of symbols and states (respectively) of M and $[Q \times \Sigma \times I]$ is the set of symbols of the form $[q, a, i]$, $q \in Q, a \in \Sigma$, $i \in I$. Note that C_i uses use $n+2$ rather than n symbols because we incorporate the adjacent blank symbols on either side of the input into the initial configuration (and thereafter assume the machine do not to exceed these squares). Intuitively, the symbol $[q, a, i]$ says that the current tape head is scanning symbol a in state q and the next instruction to be executed is the ith instruction from $\delta(\mathrm{M})$. Therefore computation path can be encoded as a word

$$\pi(x) = \#w_0\#w_1\#\cdots\#w_m\#$$

where $\#$ is some new symbol not in $\Sigma \cup [Q \times \Sigma \times I]$.

Notation. In the following, let $\Gamma = \Sigma \cup [Q \times \Sigma \times I]$ and let $\Delta = \Gamma \cup \{\#\}$. Thus $\pi(x)$ is a word over Δ.

The regular expression $E(x)$ will represent a language over Δ. In fact $E(x)$ will have the form $E_1 + E_2 + \cdots + E_6$ where the E_i's are next described.

(a) E_1 denotes the set of words over Δ that "does not begin with an #, does not end with a #, or has at most one #". Precisely,

$$E_1 = \Gamma^* + \Gamma^* \cdot \# \cdot \Gamma^* + \Gamma \cdot \Delta^* + \Delta^* \cdot \Gamma.$$

Since $+$ takes precedence over $\cdot$, this expression is unambiguous. As mentioned, we write Γ, Δ, etc., as in the shorthand where a set of symbols X stands for the regular expression $x_1 + \cdots + x_k$ if the distinct symbols in X are $x_1, \ldots, x_k$. Thus, each occurrence of Γ^* in E_1 represents a subexpression of size $2|\Gamma|$.

The remaining regular expressions in $E(x)$ will consider strings that are not in $L(E_1)$. Such strings necessarily have the form

$$y = \#x_1\#x_2\#\cdots\#x_k\# \tag{5.4}$$

for some $k \geq 1$ and $x_i \in \Gamma^*$. In our informal description of the expressions below, we will be refering to (5.4).

(b) "Some x_i in (5.4) does not have a unique symbol in $[Q \times \Sigma \times I]$"

$$E_2 = \Delta^* \cdot \# \cdot \Sigma^* \cdot \# \cdot \Delta^* + \Delta^* \cdot \# \cdot \Gamma^* \cdot [Q \times \Sigma \times I] \cdot \Gamma^* \cdot [Q \times \Sigma \times I] \cdot \Gamma^* \cdot \# \cdot \Delta^*.$$

(c) "Some x_i has length different from $n+2$"

$$E_3 = \Delta^* \cdot \# \cdot \Gamma^{n+3} \cdot \Gamma^* \cdot \# \cdot \Delta^* + \Delta^* \cdot \# \cdot (\Gamma \cup \{\lambda\})^{n+1} \cdot \# \cdot \Delta^*.$$

(d) "x_1 does not represent the initial configuration on input $x = a_1 \cdots a_n$"
Note that the initial configuration is represented by one of the forms

$$\boxdot[q_0, a_1, i]a_2 \cdots a_n\boxdot$$

where $i \in I$. Let $[q, a, I]$ denotes the set $\{[q, a, i] : i \in I\}$. For any subset S of Δ, we use the shorthand '$\bar{S}$' for the regular expression denoting the set $\Delta - S$. If $S = \{a\}$, we simply write $\bar{a}$ for $\Delta - \{a\}$. We then have:

$$E_4 = \# \cdot (\bar{\boxdot} + (\overline{[q_0, a_1, I]} + [q_0, a_1, I](\bar{a}_2 + a_2(\bar{a}_3 + \cdots + a_n(\bar{\boxdot} + \Gamma^*)\cdots)))) \cdot \Delta^*.$$

(e) "There are no accepting configurations in y"
That is, none of the symbols of $[q_a, a_0, I]$ appears in y. Here q_a is the accepting state and a_0 the special symbol that is written when the machine halts. Here we assume that the transition table for M contains trivial transitions from the accept state in which 'nothing changes' (the state, scanned symbol and head position are unchanged).

$$E_5 = (\Delta - [q_a, a_0, I])^*.$$

(f) "Some transition, from x_i to x_{i+1}, is not legal"
Recall the function $\partial : \Gamma \times \Gamma \times \Gamma \to \Gamma$ defined in the proof that 3UNIT is P-hard. There, the interpretation of $\partial(a, b, c) = b'$ is that a cell containing b with neighbors a and c will change its contents to b' in the next step. For the present proof, we can easily modify the function ∂ to account for the case where a or c might be the new symbol # and where a, b, c could be an element of $[Q \times \Sigma \times I]$. Furthermore, since M is nondeterministic, $\partial(a, b, c)$ is now a subset (possibly empty) of Δ.

For instance, if b has the form $[q, b', i]$, the ith instruction is indeed executable in state q scanning b', and b' is changed to b'' and the tape head moves right. Then we put b'' in $\partial(a, b, c)$. Moreover, if the ith instruction changes state q to q', then $\partial(b, c, d)$ (for any d) contains $[q', c, j]$ for each $j \in I$. Note that we are slightly wasteful here in allowing all possible j's to appear in the next head position, but it does not matter.

If the ith instruction is not executable, we simply set $\partial(a, b, c) = \emptyset$. Thus E_6 has the form

$$E_6 = F_1 + F_2 + \cdots$$

where each F_i corresponds to a triple of symbols in Δ^3. If F_i corresponds to the triple $(a, b, c) \in \Delta^3$ then

$$F_i = \Delta^* \cdot a \cdot b \cdot c \cdot \Delta^{n+1} \cdot \overline{\partial(a, b, c)} \cdot \Delta^*.$$

We have now completely described $E(x)$. A moment's reflection will convince the reader that $L(E(x)) = \Delta^*$ iff x is rejected by M; thus $E(x)$ is in FULL$(\Delta, \{+, \cdot, ^*\})$ iff $x \notin L(\mathrm{M})$. (Exercise: why do we need the explicit introduction of the variable i in $[q, a, i]$?) The size of $E(x)$ is given by

$$|E_1| + \cdots + |E_6| + 5 = O(1) + O(1) + O(n) + O(n) + O(1) + O(n)$$

which is $O(n)$. It is easy to verify that $E(x)$ can be computed from x in logarithmic space. Thus we have shown that every language in *co-NSPACE*(n) is reducible to FULL$(\Delta, \{+, \cdot, ^*\})$ and hence (by lemma 6(ii)) reducible to FULL$(+, \cdot, ^*)$.

5.4.3 Complexity of Games

Another rich source of problems with high complexity is the area of combinatorial games. One type of game can be defined as follows.

Definition. A *two-person game* is a quintuple

$$\langle \mathcal{I}_0, \mathcal{I}_1, p_0, \mathcal{R}_0, \mathcal{R}_1 \rangle$$

where $\mathcal{I}_0, \mathcal{I}_1$ are disjoint sets of *positions* (for *player* 0 and *player* 1, respectively), $p_0 \in \mathcal{I}_0$ is a distinguished *start* position, $\mathcal{R}_0 \subseteq \mathcal{I}_0 \times \mathcal{I}_1$ and $\mathcal{R}_1 \subseteq \mathcal{I}_1 \times \mathcal{I}_0$.

A pair $\langle p, p' \rangle$ in $\mathcal{R}_0$ is called a *move* for player 0; we say that there is a *move from* p *to* p'. The analogous definition holds for player 1. We call p an *end position* for player b ($b = 0, 1$) if $p \in \mathcal{I}_b$ and if there are no moves from p. A *match* is a sequence $\mu = (p_0, p_1, p_2, \ldots)$ of positions beginning with the start position such that $\langle p_i, p_{i+1} \rangle$ is a move for all $i \geq 0$ and either the sequence is finite with the last position an end position, or else the sequence is infinite. For $b = 0, 1$, we say that *player* b *loses a match* μ if the sequence μ is finite and the last position in μ is an end position for

player b; in that case player $1-b$ *wins the match*. In other words, the player whose turn it is to play loses if he has no move. The match is a *draw* if it is infinite. A position p is a *forced win* for Player b if (basis case) p is an end position for player $1-b$, or else (inductive case):

(a) either $p \in \mathcal{I}_b$ and there is a move from p to a forced win for player b

(b) or $p \in \mathcal{I}_{1-b}$ and for every $p' \in \mathcal{I}_b$, if there is a move from p to p' then p' is a forced win for player b.

An example of such a game is by given Even and Tarjan [10]: Let G be a given undirected graph on the vertex set $[1..n]$ for some $n \geq 2$. One player ('short') tries to construct a path from node 1 to node n and the other player ('cut') attempts to frustrate this goal by constructing an $(1,n)$-*antipath* (i.e. a set of nodes such that every path from node 1 to n must pass through the set). The game proceeds by the players alternately picking nodes from the set $[1..n]$: the first player to achieve his goal wins.

More formally: A position is a pair $\langle S_0, S_1\rangle$ where S_0 and S_1 are disjoint sets of vertices and $S_0 \cup S_1 \subseteq \{2,3,\ldots,n-1\}$. If $|S_0 \cup S_1|$ is even then it is a position of player 0, else of player 1. The start position is $\langle\emptyset,\emptyset\rangle$. An end position is $\langle S_0, S_1\rangle$ such that either

(a) it is a position for player 1 and there is a path in G from node 1 to node n passing only through the vertices in S_0, or

(b) it is a position for player 0 and S_1 is an $(1,n)$-antipath.

Note that (a) and (b) represents, respectively, a winning position for player 0 (short) and player 1 (cut). Also if $S_0 \cup S_1 = \{2,3,\ldots,n-1\}$ then p must be an end position. Hence there are no draws in this game. Suppose $p = \langle S_0, S_1\rangle$ is a position for player b but p is not an end position. Then there is a move from p to $\langle S_0', S_1'\rangle$ iff for some $v \in \{2,3,\ldots,n-1\} - (S_0 \cup S_1)$, such that $S_{1-b}' = S_{1-b}$ and $S_b' = S_b \cup \{v\}$. Thus no node is picked twice.

Generalized Hex (HEX)

Given: An undirected graph G over $[1..n]$, $n \geq 2$.

Property: Does player 0 have a forced win from the start position?

Remark: This game is also called *Shannon switching game on vertices*. For the analogous game where moves correspond to choosing edges the optimum strategy is considerably easier and can be determined in polynomial time.

Theorem 9 HEX *is PSPACE-complete.*

The problem is in PSPACE: The start position p_0 is a forced win for player 0 if and only if there exists some tree T with $\leq n-1$ levels with the following properties:

(a) The root is p_0 and counts as level 0. Nodes at even levels are positions of player 0 and nodes at odd levels are positions of player 1.

(b) If there is an edge from a node p to a child node q then there is a move from p to q. If p is a position of player 0 then p has exactly one child. If p is a position of player 1 then the set of children of p represents all possible moves from position p.

The condition that positions for player 0 has exactly one child implies that such positions cannot be leaves of the tree; thus all leaves are at odd levels. Some such tree T can be searched nondeterministically using a depth-first algorithm. The search is fairly standard – basically, we need to keep a stack for the path from the root to the node currently visited. This stack has depth n, so if each node of the path requires linear storage, we have an algorithm using quadratic space. (It is not hard to see that linear space is sufficient.)

We postpone the proof that HEX is *PSPACE*-hard until chapter 9 since it is easy to reduce HEX to the problem QBF (quantified Boolean formulas) that will be introduced there.

Schaefer [29] shows several other games that are complete for *PSPACE*. Orlin [24] and Papadimitriou [25] show other methods of deriving natural *PSPACE*-complete problems.

5.5 Complete problems with exponential complexity

5.5.1 The power of squaring

The next two subsections prove two results of Meyer and Stockmeyer:

Theorem 10 $\mathrm{FULL}(+, \cdot, ^*, ^2)$ *is EXPS-complete.*

Theorem 11 $\mathrm{INEQ}(+, \cdot, ^2)$ *is NEXPT-complete.*

Observe that the first result involves adding the squaring operator to the regular operators; in the second result we replace the Kleene-star operator by the squaring operator. The hardness proofs in both cases come from making simple modifications to the proof for $\mathrm{FULL}(+, \cdot, ^*)$ in the last section.[5]

In the proof for $\mathrm{FULL}(+, \cdot, ^*)$, the expression $E(x)$ contains subexpressions of the form S^k where S is one of $\Gamma, (\Gamma \cup \{\alpha\}), \Delta$, and where $k \in \{n-1, n, n+1\}$. Sets of the form S^k are called *rulers* (*of length k*) because they measure the distance between two corresponding (or neighboring) symbols in consecutive configurations. The crux of the present proofs lies in the ability to replace such expressions by

[5] An interesting remark is that the role of Kleene-star in transforming a problem complete for a time-class to a corresponding space-class is linked to the ability of forming 'transitive closure'. (Cf. [5])

squaring expressions of exponentially smaller size. The following lemma makes this precise:

Lemma 12 *Let Σ be an alphabet and k a positive integer.*

(i) *Then there is an $\{+, \cdot, {}^2\}$-expression α such that $L(\alpha) = \Sigma^k$ and $|\alpha| = O_\Sigma(\log k)$. Let $[\Sigma^k]_{sq}$ denote such an expression α.*

(ii) *There is a log-space transformation from the binary representation of integer k to $[\Sigma^k]_{sq}$.*

Proof. (i) The proof uses a well-known trick: using induction on k, we have $[\Sigma^{\,1}]_{sq} = \Sigma$, $[\Sigma^{2k}]_{sq} = ([\Sigma^k]_{sq})^2$ and $[\Sigma^{2k+1}]_{sq} = \Sigma \cdot [\Sigma^{2k}]_{sq}$. (ii) We leave this as an exercise. **Q.E.D.**

5.5.2 An exponential space complete problem

We prove theorem 10. To show that FULL$(+, \cdot, {}^*, {}^2)$ is in *EXPS*, we observe that any expression α with squaring can be expanded to an equivalent expression β without squaring (just replace each expression S^2 by $S \cdot S$). Clearly $|\beta| \leq 2^{|\alpha|}$. But β is a regular expression and by the results of the previous section it can be accepted in space $O(|\beta|)$.

To show that the problem is *EXPS*-hard, suppose that M is a 2^n space-bounded deterministic acceptor (other exponential bounds for *EXPS* machines can be treated in much the same way as shown here for 2^n). Indeed, to make the following notations less cluttered, we may assume that M accepts in space $2^{n-1} - 1$ (use space compression). For each input x of length n, let

$$\pi(x) = \#w_0\#w_1\#\cdots\#w_m\# \quad (\text{where } m = 2^{2^{O(n)}})$$

represent a computation path of M where each configuration w_i has length 2^n. We show how to construct in logarithmic space an expression $\hat{E}(x)$ such that $\hat{E}(x) \in$ FULL$(+, \cdot, {}^*, {}^2)$ iff $x \notin L(\text{M})$. (This is sufficient since *EXPS* is closed under complementation.)

We need rulers of length k where $k = 2^n - \delta$, $\delta \in \{-1, 0, +1\}$. We exploit the previous lemma to express these rulers using squaring expressions of size $O(n)$. Recall the regular expression $E(x)$ in the proof for regular expressions in the previous section; the ruler subexpressions in $E(x)$ are of the form $S^{n+\delta}$ where $\delta \in \{-1, 0, +1\}$. The expression $\hat{E}(x)$ is obtained from $E(x)$ by replacing each ruler subexpression $S^{n+\delta}$ in $E(x)$ by the squaring expression $[S^{2^n+\delta}]_{sq}$ defined in the previous lemma. The reader can easily confirm that $\hat{E}(x)$ has linear size.

5.5.3 An exponential time complete problem

We prove theorem 11. It is easy to see that the $\{+, \cdot, ^2\}$-expressions can only denote finite languages. We leave it as an exercise to show that the inequality problem for $\{+, \cdot, ^2\}$-expressions is in $NEXPT$.

To show that the problem is $NEXPT$-hard, suppose that M is a $2^n - 2$ time-bounded nondeterministic acceptor (again, other time bounds for $NEXPT$ machines can be treated in the same way). For each input x of length n, let

$$\pi(x) = \#w_0\#w_1\#\cdots\#w_m\# \quad (\text{where } m = 2^n)$$

represent a computation path of length 2^n where each configuration w_i has length 2^n. Note: we want the w_i's to represent the contents in tape cells whose absolute index (i.e., address) is $-2^n + 1$ to $2^n - 1$, so each symbol of w_i has as many tracks as the number of tapes of M, and each track encodes two tape symbols, etc. We will describe two $\{+, \cdot, ^2\}$-expressions $E^1(x)$ and $E^2(x)$ corresponding to the input word x such that $L(E^1(x)) \neq L(E^2(x))$ iff x is accepted by M. The expression $E^1(x)$ is very similar to $\hat{E}(x)$ in the above proof for $EXPS$. In fact, $E^1(x)$ is obtained in two steps:

(1) Since the Kleene-star operator is no longer allowed, we replace each subexpression of the form S^* (where $S = \Delta, \Gamma$, etc) with $(S \cup \{\lambda\})^{2^{2n+1}}$. The reason for the exponent '$2n + 1$' is because the string $\pi(x)$ has length $(2^n + 1)^2 < 2^{2n+1}$ for $n \geq 2$. We write S^{2^k} here just as a shorthand for k applications of the squaring operation. Thus the size of S^{2^k} is $O(k)$. Since the original expression $\hat{E}(x)$ has a fixed number of Kleene-star operators, the replacements maintain the linear size of the original expression.

(2) Let $\hat{E}^1(x)$ be the expression after the replacements of (a). The reader may verify that $\hat{E}^1(x)$ satisfies:

   (i) Each word w in $L(\hat{E}^1(x))$ has length $\leq 5 \cdot 2^{2n+1} + 4$. (This maximum length is achieved by the modified subexpression E_2 in $E(x)$.)

   (ii) A word $w \in \Delta^*$ of length $\leq 2^{2n+1}$ is in $L(\hat{E}^1(x))$ if and only if it does not represent an accepting computation of M on input x.

   We obtain $E^1(x)$ as $\hat{E}^1(x) + F$ where $L(F)$ denotes all words w of length $2^{2n+1} < |w| \leq 5 \cdot 2^{2n+1} + 4$. Clearly F can be written as

$$[\Delta^{2^{2n+1}+1}]_{sq} \cdot [(\Delta + \lambda)^{2^{2n+3}+3}]_{sq}.$$

Our description of $E^1(x)$ is complete. Let $E^2(x)$ express the set of words of length at most $5 \cdot 2^{2n+1} + 4$. Therefore $L(E^1(x)) \neq L(E^2(x))$ iff x is accepted by M. Clearly $E^1(x)$ and $E^2(x)$ have linear size by our remarks. This concludes the proof that INEQ$(+, \cdot, ^2)$ is $NEXPT$-complete.

5.5.4 Other Problems

The following chart summarizes the complexity of some word problems. Most of these results are from Meyer and Stockmeyer. In the chart, we say a class K is a 'lower bound' for a problem L to mean that L is K-hard. It is intuitively clear why such a result is regarded as a 'lower bound'; in the next chapter we show that such a result can be translated into explicit lower bounds on the complexity of any Turing machine accepting L.

Problem	Lower Bound	Upper Bound
INEQ$(+, \cdot, \neg)$	$DSPACE(\exp(\log n))$	$DSPACE(\exp(n))$
FULL$(+, \cdot, {}^2, {}^*)$	$EXPS$	$EXPS$
INEQ$(+, \cdot, {}^2)$	$NEXPT$	$NEXPT$
FULL$(+, \cdot, {}^*)$	$NSPACE(n)$	$NSPACE(n)$
INEQ$(+, \cdot)$	NP	NP
INEQ$(\{0\}, \{+, \cdot, {}^2, \neg\})$	$PSPACE$	$PSPACE$
INEQ$(\{0\}, \{+, \cdot, {}^*\})$	NP	NP
INEQ$(\{0\}, \{+, \cdot, \neg\})$	P	P

Notes:

(i) In row 1 we refer to the super-exponential function $\exp(n)$ defined in the appendix of chapter 4. The next section considers a closely related result.

(ii) We have so far assumed an alphabet Σ of size at least two; the last three rows of this table refers to unary alphabets.

(iii) With the exception of row 1, all the upper and lower bounds agree. In other words, each language L above is K-complete for its class K.

(iv) The lower bounds for the languages FULL$(+, \cdot, {}^2, {}^*)$, INEQ$(+, \cdot, {}^2)$ and FULL$(+, \cdot, {}^*)$ have just been given in sections 4 and 5. It is easily observed that the many-one reductions in these proofs all use transformations t that are linearly bounded i.e. $|t(x)| = O(|x|)$ for all x. This fact will be used in chapter 6.

Other problems based on games that are complete for exponential time or space can be found in [6].

5.6 Elementary Problems

5.6.1 The power of negation

Recall the super-exponential function $\exp(n, m)$ given in the appendix of chapter 4.

Definition 1 *A language is called* elementary *(or, elementary recursive) if it can be accepted in deterministic space* $\exp(k, n)$ *for some some integer* $k \geq 0$. *Let ELEMENTARY denote the class of elementary languages.* ■

The class of 'elementary functions' was defined by Kalmar [26]; the characterization of the corresponding class of languages in terms of its space complexity (as in the preceding definition) is due to Ritchie [27]. Our goal in this section is to show that if we admit the negation operator, then the complexity of the word problems we study is enormously increased. More precisely, we are concerned with Ω-expressions where $\Omega = \{+, \cdot, ^*, \neg\}$; call these the *regular expressions with negation*. We show that the fullness problem for such expressions are as hard as any elementary problem. The first natural problem shown to be *ELEMENTARY*-hard is the decision problem for the so-called *weak monadic second order theory of one successor* (WS1S), due to Meyer [22].[6] Meyer showed that the emptiness problem for the so-called *gamma expressions* (γ-expressions) can be efficiently reduced to WS1S. We shall define gamma expressions[7] as those that use the operators

$$\cdot, +, ^*, \neg, \gamma$$

where only the last operator, γ, needs explanation. γ is a unary operator such that for any γ-expression α, the γ-expression $\gamma(\alpha)$ denotes the set

$$L(\gamma(\alpha)) = \{w : (\exists x)[x \in L(\alpha) \text{ and } |w| = |x|]\}.$$

Recall from the last section that the set Σ^k is called a ruler of length k; if we can give very succinct γ-expressions to denote very long rulers then the appropriate word problem for the γ-expressions is proportionally hard to decide. We now describe the ideas of Meyer for describing very long rulers by exploiting negation and γ. We introduce a special Turing machine just for the purposes of our proofs.

Definition 2 *A counting machine is a deterministic simple Turing machine* M_c *with tape alphabet* $\{0, 1, \&\}$*, such that on input* $\&x\&$*,* x *a binary word of length* $n \geq 1$*, the tape head will remain within the left and right boundary markers '&' and will loop while performing the following repeatedly: treating the contents between the markers as a binary number between 0 and* $2^n - 1$*, the machine successively increments the binary number modulo* 2^n*.* ■

One realization of such a machine has three states q_0, q_1 and q_2, with the following deterministic transition table:

[6] This is the problem of deciding the validity of second order formulas where the second order variables vary over finite sets and the only non-logical symbol is the successor relation.

[7] Meyer's construction does not require negation or Kleene-star. We use these to simplify our illustration.

δ	0	1	&
q_0	–	–	$(q_1, \&, +1)$
q_1	$(q_1, 0, +1)$	$(q_1, 1, +1)$	$(q_2, \&, -1)$
q_2	$(q_1, 1, +1)$	$(q_2, 0, -1)$	$(q_0, \&, 0)$

The table rows and columns are labeled by states q and symbols b, respectively. An entry (q', b', d) in the q-row and b-column says if the machine is in state q scanning symbol b, it next enters state q', changes symbol b to b' and moves its head in the direction indicated by d. The first row of this table for state q_0 (the start state) is only defined for the input symbol '&'. We assume that the input string has the form $\&x\&$ where $x \in \{0,1\}^*$ and the machine is started in state q_0 scanning the leftmost symbol '&'. The behavior of this machine is easily described:

(i) The machine in state q_1 will move its head to the right until it encounters the right marker '&'; during this movement, it does not modify the contents of the tape. When it sees '&', it enters state q_2 and reverses direction.

(ii) The machine only enters state q_2 while at the right marker in the way indicated in (i); in state q_2, the machine will move its head leftward until it finds the first 0 or, if there are no 0's, until the left marker '&'. During the leftward motion, it changes the 0's into 1's. If it finally encounters the symbol 1, it changes it to a 0, reverses direction and enters state q_1. If it encounters the symbol & instead, it does not change the symbol but enters state q_0 while remaining stationary. Entering state q_0 indicates the start of a new cycle.

Notation and convention. From now on, when we say 'the counting machine' we refer to the particular machine just described. Let Σ_c be the alphabet for encoding computations of the counting machine, consisting of the thirteen symbols:

$$\#, \&, 0, 1, [q_j, 0], [q_j, 1], [q_j, \&]$$

where $j = 0, 1, 2$. (The symbol # will not be used until later.) If the non-blank portion of the tape of a simple Turing machine is a string of the form $w_1 b w_2$ (where $w_1, w_2 \in \{\&, 0, 1\}^*$ and b is the currently scanned symbol) and the machine is in state q, we encode this configuration as $C = w_1[q, b]w_2$. Also, whenever we write $C \vdash C'$ we assume the lengths of the encodings of the C and C' are equal: $|C| = |C'|$. We will assume that the machine is normally started with tape contents $\&0^n\&$ for some $n \geq 1$. As usual, we assume that the head of the machine is initially scanning the leftmost symbol of the input; the string encoding this initial configuration is denoted by

$$init(n) = [q_0, \&]0^n\&.$$

Using the ideas of the last two sections, we can easily construct a regular expression α_n that denotes all strings w that *fail* to satisfy at least one of the following:

(a) w has the form

$$w = C_0 C_1 \cdots C_m \quad \text{(for some } m \geq 0\text{)}$$

where the first and last symbol of each C_i is & or $[q_i, \&]$.

(b) C_0 is the configuration $init(n)$.

(c) C_i is the successor of configuration C_{i-1} $(i = 1, \ldots, m)$, and C_0 is the successor of C_m.

(d) All the C_i's are distinct.

Note that each C_i marks its own boundaries with the symbol &. The size of α_n, properly expressed, would be $O(n)$. Observe that a word w satisfying (a)-(d) has length at least $n2^n$ since the number of configurations m is at least 2^n ($O(2^n)$ is also an upper bound on m – see Exercises). Since M_c is deterministic, the complement of $L(\alpha_n)$ denotes a single word. The availability of the γ operator then gives us the expression $\gamma(\neg\alpha_n)$ which denotes a ruler of length $\geq n2^n$.

In order to recursively apply this idea to get longer rulers, we can further show:

(γ) There is a constant $c > 0$ such that, given an γ-expression ρ_n denoting a ruler of length n, we can efficiently construct a γ-expression ρ_m (for some $m \geq 2^n$) denoting a ruler of length m such that $|\rho_m| \leq c|\rho_n|$.

Hence, after k applications of this result, we can get a γ-expression of size $O(c^k n)$ denoting a ruler of length at least $\exp(k, n)$. Using such an expression, we can then describe Turing machine computations where each configuration in the path uses space $\exp(k, n)$. If the γ operator is not allowed, then it is less clear that the recursive construction can be carried out. The remainder of this section shows how this can be done, as shown by Stockmeyer [32].

For our result below, the concept of negation-depth is important. If we view an expression as a tree whose internal nodes are operands and leaves are atoms, then the negation-depth is the maximum number of negations encountered along any path of this tree. More precisely, the *negation-depth* (or $\neg$-depth) of an extended regular expression is recursively defined as follows: the $\neg$-depth of an atom is zero; the $\neg$-depth of

$$(\alpha + \beta), (\alpha \cdot \beta), (\alpha \cap \beta)$$

is the maximum of the $\neg$-depths of α and β; the $\neg$-depth of

$$\alpha^2, \alpha^*$$

is equal to the $\neg$-depth of α; finally the $\neg$-depth of $\neg\alpha$ is one plus the $\neg$-depth of α.

5.6.2 Homomorphism, rotation, smearing and padding

In the remainder of this section, 'expression' shall mean 'regular expression with negation' over a suitable alphabet Σ that varies with the context. The present subsection discusses technical tools to facilitate our rendition of Stockmeyer's construction. The reader should recall the notion of letter-homomorphisms (appendix in chapter 2) since it is the basis of many constructions.

Let Σ_1, Σ_2 be alphabets. In the following it is convenient to distinguish two ways to form composite alphabets: *horizontal composition*

$$[\Sigma_1 \times \Sigma_2] = \{[a, b] : a \in \Sigma_1, b \in \Sigma_2\}$$

and *vertical composition*

$$\left[\begin{array}{c} \Sigma_1 \\ \Sigma_2 \end{array} \right] = \{ \left[\begin{array}{c} a \\ b \end{array} \right] : a \in \Sigma_1, b \in \Sigma_2\}.$$

A string in $\left[\begin{array}{c} \Sigma_1 \\ \Sigma_2 \end{array} \right]^*$ can be regarded as the contents of a two-tracked tape. Define the functions h_1 and h_2 that respectively extract the first and second component of a horizontally composite symbol: $h_i([b_1, b_2]) = b_i$, $i = 1, 2$. These functions then extend in the usual manner to the letter-homomorphisms (still denoted by h_i):

$$h_i : [\Sigma_1 \times \Sigma_2]^* \to \Sigma_i^*.$$

We shall also use the inverse map

$$h_i^{-1} : \Sigma_i^* \to 2^{[\Sigma_1 \times \Sigma_2]^*}$$

where for any set X, we use 2^X to denote the set of subsets of X, and $h_i^{-1}(w)$ is defined to be the set of $x \in [\Sigma_1 \times \Sigma_2]^*$ such that $h_i(x) = w$.

For any homomorphism $h : \Sigma^* \to \Gamma^*$, and any language $L \subseteq \Sigma^*$, we call $h(L)$ the *h-image* of L. If $L' \subseteq \Gamma^*$, and $h(L') = L$ then we call L' an *h-preimage* (or *h-pre-image*) of L. Note that in general, an h-preimage of L is only a subset of $h^{-1}(L)$; we may call $h^{-1}(L)$ the *full* preimage of L.

Similarly, h_U and h_L extract the upper and lower components of a vertically composite symbol: $h_U(\left[\begin{array}{c} a \\ b \end{array} \right]) = a$, $h_L(\left[\begin{array}{c} a \\ b \end{array} \right]) = b$. Again we have the corresponding letter homomorphisms and their inverses. Since horizontal and vertical compositions have only pedagogical differences, we only need to state properties for just one of them. The following simple lemma will be one of our basic tools:

Lemma 13 *Suppose $h : \Gamma^* \to \Sigma^*$ is a letter-homomorphism. There is a log-space computable transformation t such that for any expression α over the alphabet Σ, we can construct another expression $t(\alpha)$ over Γ such that $L(t(\alpha)) = h^{-1}(L(\alpha))$. Furthermore, the size of $t(\alpha)$ is at most $c|\alpha|$ for some c that only depends on the size of Γ.*

Proof. Replace each atom $b \in \Sigma$ occurring in α with the subexpression $b_1+b_2+\cdots+b_t$ where $h^{-1}(b) = \{b_1, \ldots, b_t\}$. **Q.E.D.**

We describe three operations on strings:

Definition 3 *Let $x = b_1 b_2 b_3 \cdots b_m$ for some $m \geq 1$, and each $b_j \in \Sigma$ for some alphabet Σ.*

(1) *Then rotate(x) denotes the set of strings over Σ of the form $x_1 x_2$ ($x_i \in \Sigma^*$) such that $x_2 x_1 = x$.*

(2) *If $m \geq 3$, then smear(x) denotes the string*

$$[x_m, x_1, x_2][x_1, x_2, x_3][x_2, x_3, x_4] \cdots [x_{m-2}, x_{m-1}, x_m][x_{m-1}, x_m, x_1].$$

Note that smear(x) is over the alphabet $[\Sigma \times \Sigma \times \Sigma]$ and $|\text{smear}(x)| = |x|$. Also, $h_2(\text{smear}(x)) = x$ where h_2 is the homomorphism on $[\Sigma \times \Sigma \times \Sigma]$ that picks out the second component.

(3) *Assume $\# \notin \Sigma$, and $k \geq 0$ is an integer. Then pad$_k(x)$ denotes the set of strings*

$$b_1 x_1 b_2 x_2 \cdots x_{m-1} b_m x_m,$$

where each x_i is a string of #'s of length at least k. Note that the x_i's can all have the different lengths. The function unpad, which simply deletes all occurrences of '#' in its argument, acts as a partial inverse to pad$_k$: if w contains no '#' then unpad$(\text{pad}_k(w)) = w$.

■

The three operations extend in the natural way to sets of words: for instance, $\text{smear}(L) = \{\text{smear}(x) : x \in L \text{ and } |x| \geq 3\}$. The rotate and smear operations are seen to commute with each other, but they do not commute with the padding operation. Let

$$\begin{aligned} \Sigma_{3c} &= [\Sigma_c \times \Sigma_c \times \Sigma_c] \\ \Gamma_c &= \Sigma_{3c} \cup \{\#\}. \end{aligned}$$

Thus Σ_{3c} (resp., Γ_c) is the alphabet for representing smeared computations (resp., padded smeared computations).

5.6.3 Free computation and cyclic ruler

A key idea in Stockmeyer's construction is to modify the notion of rulers that we have used until now.

Definition 4 *Recall the alphabet Σ_c for encoding computations for the counting machine M_c. A free computation (of order n) ($n \geq 1$) is a string over Σ_c of the form*

$$\bar{C} = C_0 C_1 \cdots C_m \quad (m \geq 0)$$

such that

(1) *Each C_j is (an encoding of) a configuration of the counting machine* M_c, $j = 0, \ldots, m$.

(2) $C_0 = [q_0, \&]0^n\& = init(n)$.

(3) $C_{i-1} \vdash C_i$ *for* $i = 1, \ldots, m$, *and* $C_m \vdash C_0$.

■

Note that the boundary between consecutive configurations is naturally marked by '&&' (or variants containing state information). Observe that a free computation can be arbitrarily long since the configurations can repeat indefinitely. A *cycle* (of order n) is defined to be a free computation $\bar{C} = C_0 \cdots C_m$ such that all the C_i's are distinct and $C_0 = init(n)$. It follows that for each n, the cycle of order n is unique. Define the number-theoretic function g such that a cycle of order n is a string of length $g(n)$. Clearly $g(n) \geq n2^n$ and by earlier remarks, in fact $g(n) = \Theta(n2^n)$. Also the length of a free computation of order n is a multiple of $g(n)$. We use the following important notations:

$free(n)$	$=$	$\bigcup\{rotate(smear(x)) : x$ is a free computation of order $n\}$
$non\text{-}free_k(n)$	$=$	$\{x : x \notin pad_k(free(n))\}$
$ruler_k(n)$	$=$	$pad_k(rotate(smear(x)))$ where x is the cycle of order n.

We call a set of the form $ruler_k(n)$ a *cyclic ruler*. Clearly we intend to use cyclic rulers instead of the usual rulers. Note that $free(n) \subseteq (\Sigma_{3c})^*$ and $non\text{-}free_k(n) \cup ruler_k(n) \subseteq (\Gamma_c)^*$.

We now make a simple but important observation about our construction of the counting machine. Let $u_0 \in \Sigma_{3c}$ denote the special symbol

$$u_0 := [\&, [q_0, \&], 0].$$

Lemma 14 *Let $w = C_0 C_1 \cdots C_m$ denote a free computation of order $n \geq 1$, and $C'_0 C'_1 \cdots C'_m = smear(w)$ where C'_i corresponds to C_i in the natural way, $h_2(C'_i) = C_i$. For each i, we have: C'_i contains the symbol u_0 if and only if $C_i = init(n)$.*

Thus it is sufficient to use local information (the presence of the symbol u_0) to detect the initial configuration. The next lemma shows that we can obtain expressions for cyclic rulers from expressions for $non\text{-}free_k(n)$ at the cost of increasing the negation-depth by 1.

Lemma 15 *Let Σ be any alphabet and h a letter homomorphism from Σ to Γ_c.*

(i) *There is a log-space transformation that, given any expression α where $h(L(\alpha)) = non\text{-}free_k(n)$, constructs an expression β where $h(L(\beta)) = ruler_k(n)$.*

(ii) *The length of β satisfies $|\beta| = |\alpha| + O(1)$, and the negation-depth of β is one greater than that of α.*

Proof. A word w in $pad_k(free(n))$ fails to be in $ruler_k(n)$ precisely when w contains more than one occurrence u_0. Hence it is easy to satisfy (i) by letting β be

$$\beta = \neg(\alpha + \Sigma^* \cdot h^{-1}(u_0) \cdot \Sigma^* \cdot h^{-1}(u_0) \cdot \Sigma^*)$$

Clearly (ii) holds. **Q.E.D.**

Thus, given an expression denoting an h-preimage of $non\text{-}free_k(n)$, this lemma shows us how to obtain another expression denoting an h-preimage of $ruler_k(n)$. To get succinct expressions for h-preimage of $non\text{-}free_k(n)$, the next lemma shows how to do this inductively.

Lemma 16 (Key Construction) *Let Σ be any alphabet and h a letter homomorphism from Σ to Γ_c. Let $\Delta = \begin{bmatrix} \Gamma_c \\ \Sigma \end{bmatrix}$ with the usual homomorphisms $h_L, h_R : \Delta \to \Sigma$ selecting the upper and lower tracks. Let H be the letter homomorphism from Δ to Γ_c given by $H(x) = h(h_L(x))$ for all x.*

(i) *There is a log-space transformation t such that given any expression α where $h(L(\alpha)) = non\text{-}free_k(n)$, constructs the expression $t(\alpha)$ over the alphabet Δ where $H(L(t(\alpha))) = non\text{-}free_{k+1}(g(n))$. Recall the function $g(n) \geq n2_n$.*

(ii) *$|t(\alpha)| = O(|\alpha| \cdot |\Sigma|)$ and the negation-depth of $t(\alpha)$ is one greater than that of α.*

Proof. We construct the expression $t(\alpha)$ to denote all those words $w \in \Delta^*$ that are *not* of the following form:

b_1	$\#$	$\#^{i_1}$	b_2	$\#$	$\#^{i_2}$	b_3	$\cdots$
$h^{-1}(\#)$	$h^{-1}(a_1)$	$h^{-1}(\#^{i_1})$	$h^{-1}(\#)$	$h^{-1}(a_2)$	$h^{-1}(\#^{i_2})$	$h^{-1}(\#)$	$\cdots$

$\cdots$	$\#$	$\#^{i_{m-1}}$	b_m	$\#$	$\#^{i_m}$
$\cdots$	$h^{-1}(a_{m-1})$	$h^{-1}(\#^{i_{m-1}})$	$h^{-1}(\#)$	$h^{-1}(a_m)$	$h^{-1}(\#^{i_m})$

for some $m \geq 4$ where:

(a) $a_j, b_j \in \Sigma_{3c}$, and $i_j \geq k$ for all $j = 1, \ldots, m$.

(b) the bottom track contains a word u such that

$$\begin{aligned} h(u) &= \#x \\ &= \#a_1\#^{i_1+1}a_2\#^{i_2+1}\cdots\#^{i_{m-1}+1}a_m\#^{i_m} \end{aligned}$$

where $x \notin \textit{non-free}_k(n)$ (i.e. $x \in pad_k(free(n))$).

(c) the top track contains a word

$$v = b_1\#^{i_1+1}b_2\#^{i_2+1}\cdots\#^{i_{m-1}+1}b_m\#^{i_m+1}$$

where $v \notin \textit{non-free}_{k+1}(g(n))$ (i.e., $v \in pad_{k+1}(free(n))$).

Note that each b_i in the upper track is immediately followed by a_i in the lower track: we say b_i and a_i are 'paired'. By definition, only non-# symbols can be paired.[8] The expression $t(\alpha)$ is over the alphabet $\Delta = \begin{bmatrix} \Gamma_c \\ \Sigma \end{bmatrix}$. We shall write $t(\alpha)$ as the union $E_1+E_2+\cdots+E_8$. We now begin the lengthy details of this construction. A word w is in $t(\alpha)$ if it satisfies at least one of the following conditions.

(1) "The upper track of w is incoherent"
Say a string over Γ_c is *coherent* if it is of the form $pad_0(smear(x))$ for some $x \in \Sigma_c^*$. Say a pair (b, b') of symbols in Σ_{3c} is *incoherent* if they could not possibly be consecutive symbols in the smearing of some word over Σ_c. More precisely, with $h_i : \Sigma_{3c} \to \Sigma_c$ $(i = 1, 2, 3)$ the homomorphism that extracts the ith component of its argument, we say (b, b') is incoherent precisely when $h_2(b) \neq h_1(b')$ or $h_3(b) \neq h_2(b')$. Then the following expression denotes those words whose upper track is incoherent because of a consecutive pair of non-# symbols:

$$E_1' = \Delta^* \cdot h_U^{-1}\left(\sum_{(b,b')} b \cdot \#^* \cdot b'\right) \cdot \Delta^*$$

where the summation (denoting set union) is over all incoherent pairs (b, b'). Note that here and subsequently, for any expression σ over some Σ, and homomorphism $f : \Sigma \to \Sigma'$, we write $h^{-1}(\sigma)$ as a short-hand for the expression that denotes the set $h^{-1}(L(\sigma))$, as shown in lemma 13. The upper track can also be incoherent if the first and last non-# symbols do not agree:

$$E_1'' = \sum_{(b,b')} \left(h_U^{-1}(\#^* \cdot b') \cdot \Delta^* \cdot h_U^{-1}(b \cdot \#^*)\right)$$

Our first subexpression is given by $E_1 = E_1' + E_1''$.

[8] It may appear preferable to pair a_i with b_i by placing a_i directly above b_i instead of the 'staggered' fashion used here; this is possible though it causes some inconvenience elsewhere.

(2) "the h-image of the lower track of w is not of the form $\#x$ where $x \in pad_k(free(n))$"

$$E_2 = h_L^{-1}(\Sigma_c) \cdot \Delta^* + h_L^{-1}(\#) \cdot h_L^{-1}(\alpha)$$

The first summand of E_2 captures those words w whose h-image of the lower track does not begin with a #-symbol.

(3) "Some pairing of non-# symbols in the upper and lower tracks is violated" That is, there is a consecutive pair of symbols $\cdots bb' \cdots$ occurring in w such that either $h_U(b) = \#$ and $h(h_L(b')) \neq \#$ or $h_U(b) \neq \#$ and $h(h_L(b')) = \#$.

$$E_3 = \Delta^* \cdot \Big(h_U^{-1}(\#) \cdot h_L^{-1}(h^{-1}(\Sigma_{3c}))) + h_U^{-1}(\Sigma_{3c}) \cdot h_L^{-1}(h^{-1}(\#))\Big) \cdot \Delta^*.$$

Note that we had to apply h^{-1} to symbols that are in the lower track by virtue of the homomorphism h.

(4) "Some right marker of configurations in the upper track is not aligned" This condition corresponds to our intention that the upper track (before padding and rotation) represents a smeared computation path $C'_0 C'_1 \cdots C'_h$ where each smeared configuration $C'_i \in (\Sigma_{3c})^*$ is *aligned* in the sense that the rightmost symbol in each C'_i is paired with some symbol in $h^{-1}(u_0)$ in the lower track. Conversely, each $h^{-1}(u_0)$ in the lower track must pair with a rightmost symbol of some C'_i. Thus each C'_i, if they are all aligned, has length $g(n)$. To express this, we define the set of 'right marker symbols' of the upper track:

$$RM = \{[b, \&, \&], [b, [q, \&], \&], [b, \&, [q, \&]] : b \in \Sigma_c, q \in \{q_0, q_1, q_2\}\}.$$

The expression becomes

$$E_4 = \Delta^* \cdot \Big(h_U^{-1}(RM) \cdot h_L^{-1}(h^{-1}(\overline{u_0}))) + h_U^{-1}(\overline{RM}) \cdot h_L^{-1}(h^{-1}(u_0))\Big) \cdot \Delta^*.$$

Here, $\overline{RM} = \Sigma_{3c} - RM$ and $\overline{u_0} = \Sigma_{3c} - \{u_0\}$. Let us introduce the analogous set of 'left marker symbols':

$$LM = \{[\&, \&, b], [\&, [q, \&], b], [[q, \&], \&, b] : b \in \Sigma_c, q \in \{q_0, q_1, q_2\}\}.$$

Let

$$NM = \Gamma_c - (LM \cup RM)$$

be the set of 'non-marker symbols'.

(5) "There is some stray &-symbol in the upper track" This expresses our intent that between the &-symbols, the upper track contains

only 0's and 1's, possibly with state information. Define the set $Z \subseteq \Sigma_{3c}$ consisting of symbols of the form:

$$[b, \&, b'], [b, [q, \&], b']$$

where $b, b' \in \Sigma_c$ and $q \in \{q_0, q_1, q_2\}$. The desired condition is given by

$$E_5 = \Delta^* \cdot h_U^{-1}\left(LM \cdot NM^* \cdot Z \cdot NM^* \cdot RM\right) \cdot \Delta^*.$$

Note that $Z \subseteq NM$, so E_5 simply asserts that at least one &-symbol occurs between the left and right markers.

Before continuing, let us observe that for any word w not in $L(E_1 + \cdots + E_5)$, the upper track of w must be of the form

$$pad_{k+1}(rotate(smear(C_0 C_1 \cdots C_m))) \tag{5.5}$$

where each C_i (after removing the state information) encodes a binary string delimited by two &-symbols, with $|C_i| = g(n)$. We also write

$$C_0' C_1' \cdots C_m' \tag{5.6}$$

for $smear(C_0 \cdots C_m)$ where C_i' corresponds naturally to C_i. For the remaining subexpressions, we will refer to these 'configurations' C_i and C_i'.

(6) "Some C_i does not have a unique state symbol"
We leave this to the reader.

(7) "Some C_{i+1} is not directly derived from C_i"
A pair (b, b') in Σ_{3c} is *compatible* if there are smeared configurations C, C' such that $C \vdash C'$, b occurs in C, and b' occurs in the corresponding position in C. For example, if $b = [b_1, [q, b_2], b_3]$ and $b' = [[q', b_1'], b_2', b_3']$ then (b, b') is compatible iff $b_1 = b_1'$, $b_3 = b_3'$, and $\langle q, b_2, q', b_2', -1\rangle$ is in the transition table of the counting machine. We come to the place where the recursive nature of our construction is seen. We make the following observation:

> Suppose $w \in \Delta^* - L(E_1 + \cdots + E_6)$ is a word whose upper track $h_U(w)$ has the form (5.5). If $\cdots bxb' \cdots$ occurs in $h_U(w)$ where $h_U(b')$ is a non-# symbol. Then $h_L(x)$ is in $ruler_k(n)$ implies that $(h_U(b), h_U(b'))$ is a compatible pair of symbols.

Thus the cyclic ruler 'measures' the distance between corresponding symbols. By the previous lemma, we can construct from α the expression β where $h(L(\beta)) = ruler_k(n)$. It is important to realize that this observation depends on the way we pair non-# symbols in the tracks in a 'staggered' fashion. We define

$$E_7' = \Delta^* \cdot \left(\sum_{(b,b')} h_U^{-1}(b) \cdot h_L^{-1}(\beta) \cdot h_U^{-1}(b') \right) \cdot \Delta^*.$$

where the summation ranges over all pairs (b, b') that are not compatible. Because of the effect of rotation, a pair of corresponding symbols that must be compared for compatibility may appear at the opposite ends of the string w. To handle this wrap-around, we also observe:

> Suppose $w \in \Delta^* - L(E_1 + \cdots + E_6)$ is a word whose upper track has the form (5.5). If in addition,
>
> $$h_U(w) \in NM^* \cdot RM \cdot LM \cdot NM^* \cdot b' \cdot w' \cdot b \cdot NM^*$$
>
> where $h_U(b)$ and $h_U(b')$ are non-# symbols. Then $h_U(w') \in pad_k(free(n))$ (rather than $ruler_k(n)$!) implies that $(h_U(b), h_U(b'))$ forms a compatible pair.

To understand this observation, let b, b' be two corresponding symbols from a pair of successive configurations C, C' in the upper track of w. We need to check b and b' for compatibility. Assume that the free computation is rotated so that C splits into two parts forming a prefix and a suffix of w, respectively. If b is in the suffix part, then b' is left of b and the word w can be expressed as $p \cdot b' \cdots w' \cdot b \cdot s$ where p, w', s are words in Δ^*. Furthermore, w' must be a rotated and padded free computation, and there must be exactly one [rightmarker, leftmarker] pair in the upper track of p.

Depending on how the pair C, C' is split by the wrap-around, we may get two other possibilities:

$$h_U(w) \in NM^* \cdot b' \cdot w' \cdot b \cdot NM^* \cdot RM \cdot LM \cdot NM^*$$

and

$$h_U(w) \in LM \cdot NM^* \cdot b' \cdot w' \cdot b \cdot NM^* \cdot RM.$$

These lead to the expression

$$E_7'' = \sum_{(b,b')} (F_1(b, b') + F_2(b, b') + F_3(b, b'))$$

where

$$F_1(b, b') = h_U^{-1}(NM^* \cdot RM \cdot LM \cdot NM^* \cdot b') \cdot h_L^{-1}(\neg\alpha) \cdot h_U^{-1}(b \cdot NM^*)$$
$$F_2(b, b') = \cdot h_U^{-1}(NM^* \cdot b') \cdot h_L^{-1}(\neg\alpha) h_U^{-1}(b \cdot NM^* \cdot RM \cdot LM \cdot NM^*)$$
$$F_3(b, b') = h_U^{-1}(LM \cdot NM^* \cdot b') \cdot h_L^{-1}(\neg\alpha) \cdot h_U^{-1}(b \cdot NM^* \cdot RM)$$

We finally let $E_7 = E_7' + E_7''$.

(8) "None of the C_i's are equal to $init(g(n))$"
It is sufficient to assert that either the upper track of w does not contain the

symbol u_0 or else the symbol '1' appears in the configuration containing the u_0. This will ensure that no configuration C_i has the form $[\&, [q_0, \&], \&]0^m\&$; of course, this includes the case $m = g(n)$ which is all that we are interested in. Let $Y \subseteq \Sigma_{3c}$ be the set of symbols that contain '1' or '$[q, 1]$' as its second component. Let $E_8 = E'_8 + E''_8$ where

$$E'_8 = \Delta^* \cdot h_U^{-1}\left(LM \cdot (\overline{u_0}^*) + NM^* \cdot Y \cdot NM^*) \cdot RM\right) \cdot \Delta^*$$

and E''_8 is a similar expression to take care of wrap-around.

This concludes our construction of α. To verify (ii), note that each E_i, with the exception of E_7, has size $O(|\Delta|) = O(|\Sigma|)$. Also E_7 has size $O(|\alpha| \cdot |\Delta|)$. Similarly, the $\neg$-depth of each E_7 is one more than that of α while the other E_i has depth 0. Hence (ii) follows. This concludes our proof of lemma 16. **Q.E.D.**

Let us call our ruler construction a $g(n)$-construction, since we obtained a ruler of order $g(n)$ from one of order n. See Exercises for attempts to get a $g'(n)$-construction for faster growing functions $g'(n)$. An elegant alternative to cyclic rulers is given by Hunt [2]; however Hunt's construction requires the use of the intersection operator '$\cap$'.

5.6.4 The main result

Using the function $g(n)$ in the last section we now define the function $G(k, n)$ for $k, n \geq 0$ as follows: $G(0, n) = g(n)$ and $G(k, n) = g(G(k-1, n))$ for $k \geq 1$. Clearly $G(k, n)$ is the analog of the function $\exp(k, n)$ and $G(k, n) \geq \exp(k, n)$ for all k, n. We can apply the last lemma k times to describe the complement of free-computations (and hence, cyclic rulers) of order $G(k, n)$:

Lemma 17 *There is a constant $c > 0$ such that for all integers $k \geq 0, n \geq 1$, there is an expression $\alpha_{k,n}$ over a suitable alphabet Δ_k and a homomorphism $h : \Delta_k \to \Gamma_c$ such that*

(i) $h(L(\alpha_{k,n})) = \textit{non-free}_k(G(k, n))$.

(ii) *$\alpha_{k,n}$ can be computed in space $O(\log |\alpha_{k,n}|)$*

(iii) $|\alpha_{k,n}| = n \cdot O(1)^{k^2}$. *The $\neg$-depth of $\alpha_{k,n}$ is k and $|\Delta_k| = |\Gamma_c|^k$.*

Proof. If $k = 0$ then we can easily construct $\alpha_{0,n}$, by modifying the proof that the fullness problem for regular languages is complete for *LBA*. If $k > 0$ then the last lemma shows how to construct $\alpha_{k,n}$ from $\alpha_{k-1,n}$. The alphabet Δ_k is $\begin{bmatrix} \Gamma_c \\ \Delta_{k-1} \end{bmatrix}$, so $|\Delta_k| = |\Delta_{k-1}| \cdot |\Gamma_c| = |\Gamma_c|^k$. The properties (ii) and (iii) easily follow from the construction. **Q.E.D.**

We next apply the cyclic rulers of length $G(k, n)$ to describe accepting computations of Turing acceptors that accept in space $G(k, n)$:

Lemma 18 *Fix any nondeterministic acceptor* M *that accepts in space* $G(k,n)$. *There is a suitable alphabet* $\Gamma_{\rm M}$ *such that for all input words* x *of length* n, *we can construct an expression* $\alpha_{\rm M}(x)$ *over* $\Gamma_{\rm M}$ *such that*

(i) x *is accepted by* M *iff* $\alpha_{\rm M}(x) \neq \mathrm{FULL}(\Gamma_{\rm M}, \{+, \cdot, ^*, \neg\})$.

(ii) $|\alpha_{\rm M}(x)| = n \cdot O_{\rm M}(1)^{k^2}$ *and* $\alpha_{\rm M}(x)$ *can be constructed in space* $O_{\rm M}(\log |\alpha_{\rm M}(x)|)$.

(iii) *The* $\neg$*-depth of* $\alpha_{\rm M}(x)$ *is* k.

Proof. Let Δ_k be the alphabet of the expression $\alpha_{k,n}$ in the last lemma, and Σ be the alphabet to represent the smeared computations of M. Also, let $\Delta_{k,\#} = \Delta_k \cup \{\#\}$ and $\Sigma_\# = \Sigma \cup \{\#\}$. The expression $\alpha_{\rm M}(x)$ shall denote strings over the alphabet

$$\Gamma_{\rm M} := \left[\begin{array}{c} \Sigma_\# \\ \Delta_{k,\#} \end{array} \right].$$

A string w is in $\alpha_{\rm M}(x)$ iff it does *not* have the form:

b_1	$\#$	$\#^k$	b_2	$\#$	$\#^k$	b_3	$\#$	$\#^k$	$\cdots$	b_{m-1}	$\#$	$\#^k$	b_m	$\#$	$\#^k$
$\#$	a_1	$\#^k$	$\#$	a_2	$\#^k$	$\#$	a_3	$\#^k$	$\cdots$	$\#$	a_{m-1}	$\#^k$	$\#$	a_m	$\#^k$

where

(a) the lower track has the form $\#y$ where y is in $pad_k(\text{non-free}_k(G(k-1,n)))$,

(b) the upper track encodes a padded, smeared (but not rotated) computation of M on input x,

(c) the left marker for each configuration on the upper track is paired with the u_0 marker on the lower track,

(d) the last configuration on the upper track is accepting.

Note that (c) ensures that the configurations in the upper track have length exactly $G(k,n)$ since the lower track is a free computation of order $G(k-1,n)$. The technique for writing down $\alpha_{\rm M}(x)$ should by now be routine. We leave this as an exercise. Hence if x is accepted by M iff there is an accepting computation iff $L(\alpha_{\rm M}(x)) \neq (\Gamma_{\rm M})^*$. **Q.E.D.**

In section 4, we show that for any regular expression α, we can construct a nondeterministic finite automaton δ^α with $\leq |\alpha|$ states, $|\alpha| \geq 2$. A well-known construction of Rabin and Scott can be applied to give a deterministic finite automaton Δ^α with $\leq 2^{|\alpha|}$ states and which accepts the same language as δ^α: the states of the deterministic automaton are sets of states of δ^α, and transitions of Δ^α are defined

in a straightforward manner: if X is a set of Δ^α (so X is a set of states of δ^α) then on input symbol b, the next state of Δ^α is

$$X' = \{q' : (\exists q \in X)\ q' \text{ is the next state of } q \text{ on input } b\}.$$

We use this construction to show:

Lemma 19 *Let α be a regular expression with negation, $|\alpha| \geq 3$.*

(i) *If the negation-depth of α is k then there is a nondeterministic finite automaton δ^α with $\exp(k, |\alpha|)$ states that accepts $L(\alpha)$.*

(ii) *Furthermore, the transition table of the automaton can be constructed in $\exp(k, |\alpha|)$ space.*

Proof. (Basis) If the negation-depth k is 0 then the result follows from our construction in lemma 8.

(Inductively) Suppose $k \geq 1$. First assume $\alpha = \beta + \gamma$. Assuming $|\beta| \geq 3$ and $|\gamma| \geq 3$, then δ^β and δ^γ can be recursively constructed with at most $\exp(k, |\beta|)$ and $\exp(k, |\gamma|)$ states respectively. Then the construction in section 4 applied to δ^β and δ^γ gives δ^α with $\leq \exp(k, |\beta|) + \exp(k, |\gamma|) \leq \exp(k, |\alpha|)$ states. If $|\beta| \leq 2$ or $|\gamma| \leq 2$, the same bound holds as well.

If $\alpha = \beta \cdot \gamma$ or $\alpha = (\alpha)^*$, the arguments are similar.

Finally, suppose $\alpha = \neg\beta$. If $|\beta| \geq 3$, then by induction, we may assume that δ^β has been formed with $\exp(k-1, |\beta|)$ states. We first carry out the Rabin-Scott construction to get a deterministic machine with $\exp(k, |\beta|)$ states. To get δ^α, we can easily modify this deterministic acceptor by interchanging the roles of acceptance and rejection. This modification does not change the number of states, and we are done. If $|\beta| \leq 2$, then a direct argument clinches the proof.

Part (ii) is routine once the constructions of part (i) is understood. **Q.E.D.**

We now state with our main result: For each $k \geq 0$ and alphabet Σ, let $\text{FULL}_k(\Sigma, \{+, \cdot, ^*, \neg\})$ denote the set of regular expressions with negation α where the negation-depth of α is $\leq k$ and $L(\alpha) = \Sigma^*$. As usual, we let $\text{FULL}_k(+, \cdot, ^*, \neg)$ denote the case where Σ is $\{0, 1\}$. As usual, the problem $\text{FULL}_k(\Sigma, \{+, \cdot, ^*, \neg\})$ easily be reduces to the case $\text{FULL}_k(+, \cdot, ^*, \neg)$.

Theorem 20 *For each $k \geq 0$, the language $\text{FULL}_k(+, \cdot, ^*, \neg)$ is complete for the class NSPACE$(\exp(k, n))$.*

Proof. In the following proof, we exploit the closure of space classes under complementation. Lemma 18 shows that $\text{FULL}_k(+, \cdot, ^*, \neg)$ is *co-NSPACE*$(G(k, n))$-hard, and hence *co-NSPACE*$(\exp(k, n))$-hard. To show that $\text{FULL}_k(+, \cdot, ^*, \neg)$ is in *co-NSPACE*$(\exp(k, n))$, we use the previous lemma: given a regular expression with negation α, it is easy to check whether it has negation depth at most k. We then construct the nondeterministic finite automata δ^α that accepts $L(\alpha)$ in space

$\exp(k, |\alpha|)$. Using δ^α it is easy to nondeterministically decide (see the proof for regular expressions) if there is a word not in $L(\alpha)$, i.e., if $\alpha \notin \text{FULL}_k(+, \cdot, ^*, \neg)$. **Q.E.D.**

Corollary 21 *The problem* $\text{FULL}(+, \cdot, ^*, \neg)$ *is hard for the class ELEMENTARY.*

Remarks: Of course, in the presence of negation, we could have posed these results in terms of the problem $\text{EMPTY}(+, \cdot, ^*, \neg)$. Stockmeyer also shows that the use of Kleene-star in these results is not strictly necessary (see Exercises).

5.7 Final Remarks

1. An interesting feature of many of the complete problems for the various canonical classes is that they can be grouped together into natural families. For example, the following problems based on node accessibility in graphs,

1GAP, GAP, AND-OR-GAP, HEX

are complete for $DLOG, NLOG, P, PSPACE$, respectively. Similarly, the problems

2UNSAT, UNIT, SAT, QBF

form a family of complete problems for *NLOG*, *P*, *NP*, *PSPACE* (resp.) based on logical formulas. Alternatively, the first two problems in the above sequence can be replaced by two problems in Boolean logic:

FVP, CVP, SAT, QBF

where FVP is the *formula value problem.* In section 3, CVP is explained; FVP is a similar problem except that we replace Boolean circuits in the input by Boolean formulas. See the exercises which shows FVP to be *NLOG*-complete. Schaeffer and Lieberherr have generalized the satisfiability problem in several other directions. Galil has shown that various decision problems related to restrictions on Turing machines also form a family of problems complete for various classes in the canonical list. Such examples can be multiplied. Such families are useful as a guide to inherent complexity of problems in the 'real' world. This is because natural problems have families resemblances and, given a problem L that resembles members of a family of complete languages, we can often conjecture and confirm the complexity of L relative to the canonical ruler (see chapter 1, section 8).

2. In the family of problems involving graphs, we use directed graphs. A natural question to ask is what is the complexity of undirected graph reachability (denoted UGAP)? It is intermediate in complexity between 1GAP and GAP but is not known to be complete for *NLOG*. We shall return to this problem in a later chapter on probabilistic computation.

3. It is important to emphasize that although we have shown complete languages for many important classes, there is strong evidence that many other natural classes do not have complete languages (e.g., see [14, 30, 15]).

4. There have been interesting developments stemming from investigations about the isomorphism of complete languages. If F is a family of transformations, we say that two languages L and L' are isomorphic (modulo F) if there are transformations $t, t' \in F$ such that $t(L) \subseteq L', t'(L') \subseteq L$ and $t' \circ t$ and $t \circ t'$ are identities. An important case is the Berman-Hartmanis conjecture concerning the isomorphism of all NP-complete languages. This conjecture is confirmed for all known NP-complete languages; on the other hand, there is evidence against the truth of the conjecture. For example, a positive result about isomorphism of complete languages is [3] which shows that all complete languages for $DEXPT$ are equivalent via 1-1 length increasing polynomial time transformations (see also [34, 33]). We shall return to this topic in volume 2.

Exercises

[5.1] (i) Show complete languages for $DTIME(2^{2^{O(n)}})$ and $DSPACE(2^{n2^n})$ under suitable reducibilities.
(ii) Let f be a complexity function. Under what conditions can we say that $XSPACE(f), XTIME(f), XREVERSAL(f)$ have complete languages under $\leq_m^L$?
(This exercise suggests that the special place of $\leq_m^L$ is principally derived from the special emphasis we have on the canonical list.)

[5.2] Why does the proof used for the other classes in the canonical list fail to produce a complete class for the class $PLOG$?

[5.3] Consider the structure of the proof for P and some of the other cases in theorem 1. The outline of these proofs could be described as follows: let $(K, (\mu, \rho, F))$ be any characteristic class in the canonical list where μ is a mode, ρ a resource and F a family of complexity functions. Suppose $MACHINES(\mu, \rho, F)$ has an efficient universal simulator U, where each U_i accepts in resource ρ bound $O_i(f_i)$ where $f_i \in F$. Consider the language

$$L^K = \{i\#x\#0^m : m = f_i(|x|) \wedge x \in U_i\}.$$

State the other properties required in order for L^K to be K-complete under $\leq_m^L$ (e.g., F must be 'efficiently presentable' in some sense).

[5.4] Recall the definition of the language 1GAP consisting of all directed graphs (encoded by an edge list) with outdegree 1 and where there is a path from the first node to the last node. Let 1GAP$'$ be the variant where the graphs are encoded by adjacency matrix. Show that it is impossible to reduce the 1GAP$'$ to 1GAP using $\leq_m^{1L}$-reducibility.

[5.5] Show that UGAP (undirected graph accessibility problem, defined in the concluding remarks of the chapter) is $DLOG$-hard under $\leq_m^{1L}$-reducibility.

[5.6] (Laaser) Show that the following connectivity problems on graphs are complete for $NLOG$: define CON and SCON to be the set of inputs of the form $\langle n, G\rangle$ (as in GAP) satisfying (respectively):

$$\forall m(m \in [1..n] \Rightarrow \exists \text{path from node 1to} m)$$

$$\forall m, p(m, p \in [1..n] \Rightarrow \exists \text{path from node} m \text{to} p)$$

[5.7] * Give a counter example to a 'standard' fallacy about resolution: Let C_1 and C_2 be clauses in a CNF formula F and C is their resolvent. Then F is unsatisfiable iff $(F - \{C_1, C_2\}) \cup \{C\}$ is unsatisfiable. For which direction is this true? Provide a counter example to the false direction.

[5.8] Prove the resolution theorem.

[5.9] Complete the proof that 3UNIT is in P by proving the correctness of the algorithm in the text.

[5.10] Complete the proof that 3UNIT is P-complete by replacing those clauses that do not have 3 literals by clauses that have exactly 3.

[5.11] Let M be any nondeterministic acceptor that is $s(n)$ space bounded. For each input x of length n, construct a path system $S(x)$ such that $S(x)$ is solvable iff x is accepted by M. *Hint:* Imitate the proof of Savitch's theorem.

[5.12] (Ladner) Show that the Circuit Value Problem (CVP) is P-complete under log-space many-one reducibility. To show that the problem is P-hard, give a *direct* proof (i.e., instead of reducing a known P-hard problem to CVP).

[5.13] Show the the Formula Value Problem (FVP) is $NLOG$-complete under log-space many-one reducibility.

[5.14] Show that AND-OR-GAP problem is P-complete.

[5.15] (Dobkin, Lipton, Reiss) Show that Rational Linear Programming (RLP) is P-hard under log-space many-one reducibility.

[5.16] Show direct log-space reductions between AND-OR-GAP and MCVP (there are reductions in two directions to do).

[5.17] (Reif) Show that the following decision problem related to the depth-first search algorithm is P-complete under log-space many-one reducibility. *Given*: $\langle G, u, v\rangle$ where G is a digraph over the vertices $[1..n]$, u and v are vertices. *Property*: Node u visited before v in a depth-first search of G that starts from node 1. It is important for this problem to assume that the search always choose the smallest numbered vertex to search next.

[5.18] Show that MEMBER$(+, \cdot, {}^2, {}^*, \neg)$ is in P.

[5.19] (Meyer, Stockmeyer) There is a family of problems that resembles the word problems of section 4: an *integer expression* involve the operators $+, \cdot, \cap, \cup$, built up recursively in the obvious way from the constants 0 and 1. For example, $((1+1+1)\cdot(1\cup(1+1))+(0\cup 1)$. Each expression now denotes a set of non-negative positive integers, with 0 and 1 denoting the integers zero and one, $+$ denoting addition (not union, as in extended regular expressions!), $\cdot$ denoting product, $\cap$ and $\cup$ denoting intersection and union of sets. Let N-MEMBER$(+, \cup)$ be the membership problem for integer expression. More precisely, this is the problem of recognizing all pairs of the form (x, α) where x is a binary number and α is an integer $\{+, \cup\}$-expression such that x is in $L(\alpha)$. Show that this problem is NP-complete.

[5.20] (Meyer, Stockmeyer) Prove that INEQ($\{0\}, \{+, \cdot, ^*\}$) is *NP*-complete.

[5.21] (Meyer, Stockmeyer) Prove that INEQ($+, \cdot$) is *NP*-complete.

[5.22] Give a direct procedure for deciding if $L(\alpha) = \Sigma^*$ in time $O(1)^{|\alpha|}$, avoiding the reduction to a nondeterministic finite automaton as in the text.

[5.23] (Hunt, Hopcroft) Let NONEMPTY(Ω) denote those Ω-expressions α where $L(\alpha) \neq \emptyset$. Note that if $\Omega \subseteq \{+, \cdot, ^*, ^2\}$ then $L(\alpha) \neq \emptyset$ always holds. Show that NONEMPTY($+, \cdot, ^*, \cap$) is *PSPACE*-complete.

[5.24] Consider the *equivalence problem* EQUIV(Σ, Ω) that consists of all pairs (α, β) of Ω-expressions such that $L(\alpha) = L(\beta)$. What is the relation between this and the inequivalence problem INEQ(Σ, Ω)?

[5.25] Construct the γ-expressions ρ_m as claimed in (γ) of section 6.

[5.26] (Stockmeyer) Show that the problem of FULL($+, \cdot, \neg$) is *ELEMENTARY*-hard. In other words, you have to eliminate the use of Kleene-star from the constructions of section 6.

[5.27] Show that the counting machine makes $O(2^n)$ moves. *Hint:* let c_n be the number of counting machine moves to count from 1 to $2^n - 1$. For example, we have

$$1 \to 10 \to 11 \to 100 \to 101 \to 110 \to 111$$

where the 6 transformations require 4+2+6+2+4+2=20 moves, so $c_3 = 20$. (Here we assume that the head scans the first symbol to the right of the low-order bit of the number, and returns to it after each transformation.) Also $c_1 = 0$, $c_2 = 12$. Give a recurrence for c_n.

[5.28] * The upper and lower bounds for the complexity of FULL($+, \cdot, ^*, \neg$) is not tight. Improve the reduction of the elementary problems to this problem. One way to do this is to obtain a '$\hat{g}(n)$-construction' corresponding to the Key lemma where $\hat{g}(n)$ grows faster than the $g(n) = n2^n$. For instance, it is easy to modify our counting machine to obtain a $(n^2 2^n)$-construction (how?). Such an improvement is inconsequential since we could have used k-ary counting and obtained a k^n-construction. What is the optimal choice of k?

[5.29] * Develop a theory of complete transformations. Let us say that a transformation $t_0 : \Sigma_0^* \to \Gamma_0^*$ is *hard for a family T of transformations under log-space reducibilities* if for all $t \in T$, $t : \Sigma_1^* \to \Gamma_1^*$, there are log-space transformations $s_0 : \Sigma_1^* \to \Sigma_0^*$ and $s_1 : \Gamma_0^* \to \Gamma_1^*$ such that for all $x \in \Sigma_1^*$, $s_1(t_0(s_0(x))) = t(x)$. Let T be the class of transformations computable by deterministic polynomial time transducer. Show complete transformations for T.

[5.30] * Little is known about natural complete problems for reversal or simultaneous complexity classes.

Bibliography

[1] A. Adachi, S. Iwata, and T. Kasai. Some combinatorial game problems require $\omega(n^k)$ time. *Journal of the ACM*, 31(2):361–377, 1984.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The design and analysis of computer algorithms.* Addison-Wesley, 1974.

[3] Leonard Berman. *Polynomial reducibilities and complete sets.* PhD thesis, Cornell University, 1977. PhD Thesis, Computer Science Department.

[4] Ronald V. Book. Translational lemmas, polynomial time, and $(\log\ n)^j$-space. *Theoretical Computer Science*, 1:215–226, 1976.

[5] Ronald V. Book. On languages accepted by space-bounded oracle machines. *Acta Informatica*, 12:177–185, 1979.

[6] A. K. Chandra and L. J. Stockmeyer. Alternation. *17th Proc. IEEE Symp. Found. Comput. Sci.*, pages 98–108, 1976.

[7] Steven A. Cook. An observation of time-storage trade off. *5rd Proc. ACM Symp. Theory of Comp. Sci.*, pages 29–33, 1973.

[8] D. P. Dobkin, R. J. Lipton, and S. Reiss. Linear programming is LOG-SPACE hard for P. *Information Processing Letters*, 8:96–97, 1979.

[9] Jian er Chen and Chee-Keng Yap. Reversal complexity. *SIAM J. Computing*, to appear, 1991.

[10] S. Even and R. E. Tarjan. A combinatorial problem which is complete in polynomial space. *Journal of the ACM*, 23:710–719, 1976.

[11] L. M. Goldschlager, R. A. Shaw, and J. Staples. The maximum flow problem is log space complete for P. *Theoretical Computer Science*, 21:105–111, 1982.

[12] Leslie M. Goldschlager. The monotone and planar circuit value problems are log space complete for P. *SIGACT news*, 9(2):25–29, 1977.

[13] J. Hartmanis. *Feasible Computations and Provable Complexity Properties.* S.I.A.M., Philadelphia, Pennsylvania, 1978.

[14] Juris Hartmanis and Lane A. Hemachandra. Complexity classes without machines: on complete languages for *UP*. *Theoretical Computer Science*, 58:129–142, 1988.

[15] Juris N. Hartmanis and Neil Immerman. On complete problems for *NP*∩*co-NP*. *12th ICALP (LNCS No. 194)*, pages 250–259, 1985.

[16] Juris N. Hartmanis, Neil Immerman, and Steven Mahaney. One-way log-tape reductions. *19th Symposium FOCS*, pages 65–71, 1978.

[17] Harry B. Hunt, III. On the time and tape complexity of languages, I. *5th Proc. ACM Symp. Theory of Comp. Sci.*, pages 10–19, 1973.

[18] N. D. Jones and W. T. Laaser. Complete problems for deterministic polynomial time. *Theoretical Comp. Sci.*, 3:105–117, 1977.

[19] Neil D. Jones. Space-bounded reducibility among combinatorial problems. *Journal of Computers and Systems Science*, 11:68–85, 1975.

[20] Richard E. Ladner. The circuit value problem is log space complete for *P*. *SIGACT News*, 7(1):18–20, 1975.

[21] A. R. Meyer and L. J. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential space. *13th Proc. IEEE Symp. Found. Comput. Sci.*, pages 125–129, 1972.

[22] Albert R. Meyer. Weak monadic second order theory of successor is not elementary-recursive. In Dold and Eckmann (eds.), editors, *Logic Colloquium: Symposium on Logic Held at Boston University, 1972-73*, pages 132–154. Springer-Verlag, 1975.

[23] Burkhard Monien and Ivan Hal Sudborough. On eliminating nondeterminism from Turing machines which use less than logarithm worktape space. In *Lecture Notes in Computer Science*, volume 71, pages 431–445, Berlin, 1979. Springer-Verlag. Proc. Symposium on Automata, Languages and Programming.

[24] James B. Orlin. The complexity of dynamic languages and dynamic optimization problems. *13th Proc. ACM Symp. Theory of Comp. Sci.*, pages 218–227, 1981.

[25] Christos H. Papadimitriou. Games against nature. *Journal of Computers and Systems Science*, 31:288–301, 1985.

[26] R. Péter. *Recursive Functions.* Academic Press, New York, 1967.

[27] Robert W. Ritchie. Classes of predictably computable functions. *Trans. AMS*, 106:139–173, 1963.

[28] W. J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computers and Systems Science*, 4:177–192, 1970.

[29] Thomas J. Schaefer. On the complexity of two-person perfect-information games. *Journal of Computers and Systems Science*, 16:185–225, 1978.

[30] Michael Sipser. On relativization and existence of complete sets. *9th ICALP (LNCS No. 140)*, pages 523–531, 1982.

[31] L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time. *5th Proc. ACM Symp. Theory of Comp. Sci.*, pages 1–9, 1973.

[32] Larry J. Stockmeyer. The complexity of decision problems in automata theory and logic. Technical Report Project MAC Tech. Rep. TR-133, M.I.T., 1974. PhD Thesis.

[33] Osamu Watanabe. On one-one polynomial time equivalence relations. *Theoretical Computer Science*, 38:157–165, 1985.

[34] Paul Young. Juris Hartmanis: fundamental contributions to isomorphism problems. *Structure in Complexity Theory*, 3:138–154, 1988.

Contents

Chapter 6

Separation Results

March 3, 1999

6.1 Separation Results

Let K and K' be language classes. By a *separation result* for (K, K') we mean one showing the existence of a language $L \in K - K'$; L is said to *separate K from K'*. If the answers to the inclusion questions "Is K included in K'?" of chapter 4 are mostly negative as generally conjectured, it follows that techniques for separating classes are needed to resolve these open problems. The K-complete languages are the obvious candidates for languages L that separate K from K'. Unfortunately, given a specific L (such as a K-complete language) and a specific K', a proof that $L \notin K'$ typically appears to require powerful combinatorial methods quite beyond our present ability. Separation results based on combinatorial methods are rare (but we will see them in a later chapter). Instead, it is easier to construct a non-specific L in stages: for instance, in each stage we try to include in or exclude from L some words so as to ensure that L is not equal to each language in K'. While diagonalizing over K', we must ensure that L remains in K. We are just diagonalizing over K', of course, and chapter 4 contains such constructions. In any case, assuming that we manage to separate K from K', there are two typical consequences:

(a) We can infer that other pairs of classes must also be separated as follows. By a *translation result* we mean one with the following structure:

$$K_1 \subseteq K_2 \Rightarrow K_1' \subseteq K_2'$$

where K_i, K_i' $(i = 1, 2)$ are classes. Using this translation result, a separation result for (K_1', K_2') implies a separation result for (K_1, K_2). This method is especially useful in separating nondeterministic classes which tend to resist direct diagonalization.

(b) We can infer lower bounds on languages in K. In particular, any K-complete language (with respect to some reducibility $\leq$) is not in K', provided K' is closed under the reducibility $\leq$. For instance, we can separate $PSPACE$ from $NLOG$ using the diagonalization technique above. Since the fullness problem for regular expressions (chapter 5) is $PSPACE$-complete, we conclude that the problem cannot be solved in logarithmic space.

Most of the translational techniques are based on the idea of 'padding'. The idea originated with Ruby and Fischer [25] and was popularized by Ibarra [14]. We have already encountered padding in section 1 of the previous chapter: for instance, padding can 'translate' a complete problem for $NSPACE(n^{k+1})$ into a complete problem for $NSPACE(n^k)$. Padding techniques typically translate an inclusion between two classes with lower complexity into an inclusion between classes at higher complexity. For this reason, such results are sometimes called *upward translation results*. For instance, the following will be shown:

$$\textit{If } NLOG \subseteq DLOG \textit{ then } LBA \subseteq DLBA.$$
$$\textit{If } NP \subseteq P \textit{ then } NEXPT \subseteq DEXPT.$$

Limited forms of 'downward translation result' are known. Translational techniques will treated in sections 3 and 4.

Section 5 draws conclusions of type (b) above. The technique is based on efficient reducibilities; Meyer [19, 29] first exploited such techniques. The efficient reductions shown in the previous chapter allow us to deduce lower bounds on the various complete problems studied there.

Section 6 considers what may be described as *weak separation results*: these show that two classes K and K' must be distinct, $K \neq K'$. Hence either $K \setminus K'$ or $K' \setminus K$ (or perhaps both) must be non-empty although the proof does not indicate which is the case. For instance, we can show $DLOG \neq NTIME(n^k)$ for any k. Again the padding technique is useful. It should be noted that these weak results cannot be strengthened too easily: this will be clarified in the chapter on relativization where we show that these results can be relativized to force inclusion in either direction.

Section 7 presents *strong separation results*. The idea is that the explicit lower bounds derived in section 5 are of the 'infinitely often' type (e.g., every acceptor for such a language must, for some $c > 0$, take at least c^n steps for infinitely many n). We want to strengthen this to the 'eventually'.

Section 8 shows the limitations to the separation results. For instance, we demonstrate that there are complexity functions f, g such that f is 'arbitrarily more complex' than g and yet $DSPACE(f) = DSPACE(g)$. In other words, these two classes are inseparable.

On strong separation for characteristic classes. Let us make precise our above remarks about infinitely-often versus eventually bounds. The terminology will be useful in other contexts as well.

Definition 1 *Let D be a set, θ a binary relation over D, and let $f, g : D \to D$ be partial functions with domain and range D.*
(i) The relation θ is said to hold infinitely often between f and g if there are infinitely many $x \in D$ such that both $f(x)$ and $g(x)$ are defined, and the relation $f(x)\theta g(x)$ holds. We write '$f\theta g$ (i.o.)' or '$(\exists_\infty x)f(x)\theta g(x)$' in this case.
(ii) The relation θ is said to hold eventually *between f and g if for all but finitely many $x \in D$, whenever $f(x)$ and $g(x)$ are defined then the relation $f(x)\theta g(x)$ holds. We write '$f\theta g$ (ev.)' or '$(\forall_\infty x)f(x)\theta g(x)$' in this case.* ■

As an example, if f and g are total complexity functions, then f dominates g if and only if $f \geq g$ (ev.). Note that if $f\theta g$ (*i.o.*) then there are infinitely many values of $x \in D$ at which f and g are simultaneously defined. However, it is possible that $f\theta g$ (ev.) for the trivial reason that there are only finitely many of values of $x \in D$ for which both f and g are defined.

Suppose we have a language L that separates the pair of characteristic classes $(DTIME(t), DTIME(t'))$, i.e., $L \in DTIME(t) - DTIME(t')$. This implies that for any acceptor M for L, there exists a $c > 0$ such that there are infinitely many values of n such that M requires more than $t'(n)$ steps on inputs of length n:

$$AcceptTime_{\mathrm{M}}(n) > ct'(n) \text{ (i.o.)}. \tag{6.1}$$

Recall that $AcceptTime_{\mathrm{M}}(n)$ is undefined unless M accepts some word of length n. Note that (6.1) is equivalent to $AcceptTime_{\mathrm{M}}(n) \neq O(t'(n))$. We want to strengthen (6.1) to:

$$AcceptTime_{\mathrm{M}}(n) > ct'(n) \text{ (ev.)}. \tag{6.2}$$

for some $c > 0$. We could write this in the equivalent form: $AcceptTime_{\mathrm{M}}(n) = \Omega(t'(n))$.

We say that we have a *strong separation result* for the pair of characteristic classes $(DTIME(t), DTIME(t'))$ if we show the existence of an infinite[1] language L in $DTIME(t) - DTIME(t')$ such that any acceptor M for L satisfies (6.2). This notion of strong separation extends naturally to other characteristic classes.

6.2 The Basic Separation Results

Hartmanis and Stearns [12] began the study of time-based complexity theory and obtained the so-called *deterministic time hierarchy theorem*. Together with Lewis [11], they extended the result to space complexity. The main goal of this section is to present these two separation theorems, together with a separation theorem for reversal complexity. These theorems are considered basic because their proofs involve the simplest form of diagonalization and also because most other separation results ultimately rely on them. We also see that the diagonalization arguments here

[1] To avoid the trivial interpretation of 'eventually' explained earlier.

depend on the existence of tape reduction theorems (for space, time and reversal, respectively).

Our first hierarchy theorem is the following.[2]

Theorem 1 (Deterministic space hierarchy) *Let s_1 and s_2 be complexity functions where $s_2(n) = \omega(s_1(n))$. If s_2 is space-constructible then*

$$DSPACE(s_2) - DSPACE(s_1) \neq \emptyset.$$

Proof. Let $\Sigma = \{0, 1\}$ and fix any encoding of all 1-tape Turing machines with input alphabet Σ. Let

$$\phi_0, \phi_1, \ldots$$

be a listing of these machines in the order of their code. We may assume that the code for machine ϕ_i is the binary string (still denoted) i. Note that each machine uses states and symbols from the universal sets Q_∞ and Σ_∞, so our encoding assumes a fixed letter homomorphism h_0 from $Q_\infty \cup \Sigma_\infty$ to binary strings.

We now describe an acceptor M that diagonalizes over each machine ϕ_i. On input x of length n, M does the following: M marks off exactly $s_2(n)$ tape cells. Since $s_2(n) < \infty$ (by definition of space-constructible), the marking takes a finite number of steps. Treating x as the code of the machine ϕ_x, we would like M to "begin simulating ϕ_x on x, accepting iff either ϕ_x does not accept or if ϕ_x tries to leave the area containing the marked cells". The problem is that ϕ_x may loop without ever leaving the marked cells, and a simplistic simulation would not detect this. To overcome this difficulty, we apply Sipser's technique as shown in section 9 of chapter 2: for each accepting configuration C of ϕ_x that 'can fit' within the marked cells, M does a search of the tree $T(C)$ rooted at C. If it discovers the initial configuration $C_0(x)$ in $T(C)$, M rejects. If for all such C's, M fails to discover the initial configuration then M accepts. M can do this search in space $s_2(n)$. We should clarify the qualification that C 'can fit' within the marked cells: by this we mean that the work tapes of C can be represented using the marked cells, but the input tape of C is directly represented as the input tape of the simulating machine.

We now show that the language $L(\mathrm{M})$ separates $DSPACE(s_2)$ from $DSPACE(s_1)$. By construction, $L(\mathrm{M}) \in DSPACE(s_2)$. It remains to show that $L(\mathrm{M})$ is not in $DSPACE(s_1)$. Suppose $L(\mathrm{M})$ is accepted by some deterministic N in space s_1. By the tape reduction theorem for space, we may assume that N is 1-tape and hence, by properties of the usual coding of Turing machines, we get this property: there are infinitely many indices x such that $L(\mathrm{M}) = L(\phi_x)$ and ϕ_x accepts in space $s_1(n)$.

Using the said letter homomorphism h_0 that encodes states and symbols from the universal sets as binary strings, we may assume that each configuration of N that uses space s is encoded by a binary string of length $O_{\mathrm{N}}(s)$. Choose the length of x sufficiently large so that $O_{\mathrm{N}}(s_1(|x|)) \leq s_2(|x|)$. We now prove that M accepts x iff

[2]The original result requires the qualification $s_2(n) > \log n$.

ϕ_x rejects x. If ϕ_x accepts x then it accepts in a configuration C with space at most $s_1(|x|)$ and C can be encoded in space $O_{\rm N}(s_1(|x|)) \leq s_2(|x|)$. Thus we will search the tree $T(C)$ and find the initial configuration, leading to rejecting. On the other hand, if ϕ_x does not accept x, then the search of the tree $T(C)$ for each accepting configuration C fails and by definition M accepts. This proves $L(\mathrm{M}) \neq L(\phi_x)$. **Q.E.D.**

If we are interested in separating the running space class $DSPACE_r(s_2)$ from $DSPACE_r(s_1)$, then we can avoid the assumption s_2 be space-constructible (Exercises).

We now show the time analogue of the previous theorem. However the hierarchy is not as tight because the tape reduction theorem for deterministic time (viz., the Hennie-Stearns theorem) incurs a logarithmic slow-down.

Theorem 2 (Deterministic time hierarchy) *Let t_1 and t_2 be complexity functions with $t_1(n)\log t_1(n) = o(t_2(n))$. If t_2 is time-constructible and $t_1(n) > n$ then*

$$DTIME(t_2) - DTIME(t_1) \neq \emptyset.$$

Proof. The proof is similar to the previous one, except that we now appeal to the existence of a universal machine.[3] Let $U = \{U_0, U_1, \ldots\}$ be a universal machine simulating all 2-tape deterministic machines over some fixed input alphabet. Again we construct an acceptor M that diagonalizes over each U_i that happens to accept in time $o(t_2)$. The machine M operates as follows: on input x of length n, M first copies the input x onto two of its work-tapes. Next, M uses these two copies of x as input to simulate the universal machine U_x on input x for at most $t_2(n)$ of steps (each step of M corresponds to a step of U_x). To ensure that at most $t_2(n)$ steps are used, M will concurrently time-construct $t_2(n)$, using the original x as input. If U_x accepts within $t_2(n)$ steps then M rejects; otherwise M accepts. Clearly $L(\mathrm{M})$ is in $DTIME(2n + t_2(n)) = DTIME(t_2)$.

It remains to show that $L(\mathrm{M}) \notin DTIME(t_1)$. For the sake of a contradiction, assume that $L(\mathrm{M})$ is accepted by some acceptor in time t_1. The Hennie-Stearns theorem then implies that $L(\mathrm{M})$ is accepted by some 2-tape machine N in time $t_1 \log t_1$. From the recurrence property of universal machines, there are infinitely many indices x such that $L(\mathrm{N}) = L(U_x)$ and U_x uses time $O_{\rm N}(t_1 \log t_1)$. Choosing x such that $n = |x|$ is sufficiently large,

$$2n + O_{\rm N}(t_1(n)\log t_1(n)) \leq t_2(n).$$

If U_x accepts x, then M needs $2n + O_{\rm N}(t_1(n)\log t_1(n))$ steps to simulate U_x on x until completion. Since this number of steps is at most $t_2(n)$, M will discover that

[3] We could, as in the previous proof, argue directly about the standard encoding of Turing machines, but the use of universal machines is a slightly more general (abstract) approach. Our deterministic space hierarchy theorem, however, does not seem to yield as easily to an argument by universal machines. Why?

U_x accepts x and M so rejects x by our construction. If U_x does not accept x, then we similarly see that M will accept x. This proves $L(\mathrm{M}) \neq L(U_x) = L(\mathrm{N})$, a contradiction. **Q.E.D.**

The above result can be sharpened in two ways: if t_1 is also time-constructible then Paul [22] shows that the above separation of time classes can be achieved with $t_2(n) = o(t_1(n)\log^{\epsilon} t_1(n))$ for any $\epsilon > 0$. If we consider classes defined by Turing machines with some fixed number of tapes, Fürer [10] has shown the above separation can be achieved with $t_2 = o(t_1)$ provided we restrict attention to Turing machines with a fixed number k of tapes, $k \geq 2$.

Using the space and time hierarchy theorems, we can infer separation for some of the inclusions on our canonical list:

Corollary 3

(a) $P \subset DEXPT \subset DEXPTIME$

(b) $NLOG \subset PSPACE \subset EXPS \subset EXPSPACE$

Finally we present a hierarchy theorem for reversal complexity[5]. Although we can give a direct proof using the tape reduction theorem for reversals, the following shorter proof follows from relationships between reversal and space complexity shown in chapter 2.

Theorem 4 (Deterministic reversal hierarchy) *Let $r_1(n)$ and $r_2(n)$ be complexity functions such that $(r_1(n))^2 = o(r_2(n))$, $r_1(n) = \Omega(\log n)$ and $r_2(n)$ is reversal-constructible. Then*

$$DREVERSAL(r_2(n)) - DREVERSAL(r_1(n)) \neq \emptyset.$$

Proof. Since $(r_1(n))^2 = o(r_2(n))$, by the deterministic space hierarchy theorem, there exists a language L such that

$$L \in DSPACE(r_2(n)) - DSPACE((r_1(n))^2).$$

Using the relation $DSPACE(r_2) \subseteq DREVERSAL(r_2)$ and $DREVERSAL(r_1) \subseteq DSPACE(r_1^2)$, we conclude

$$L \in DREVERSAL(r_2(n)) - DREVERSAL(r_1(n)).$$

Q.E.D.

6.3 Padding Arguments and Translational Lemmas

The theme of the hierarchy theorems in the previous section is: given a complexity function $f_1(n)$ what is the smallest complexity function $f_2(n)$ such that there is a language L accepted within complexity $f_2(n)$ but not $f_1(n)$. In each case, $f_2(n)$ need not be more than quadratic in $f_1(n)$. The basic technique in these proofs requires constructing a diagonalizing machine M that can "efficiently decide" whether a simulated machine N accepts in a given amount of resource.

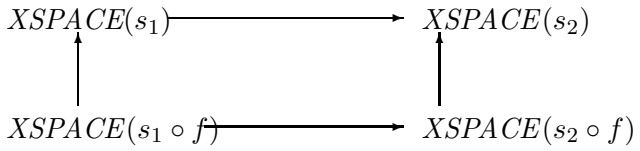
This approach becomes quite ineffectual when N is nondeterministic and we want to decide if N accepts within time t. The best nondeterministic method known for deciding this question amounts to a naive *deterministic* simulation of N, a procedure that takes time exponential in t. This implies that $f_2(n)$ is exponentially larger than $f_1(n)$.

To separate nondeterministic space, we could use Savitch's technique to deterministically simulate nondeterministic space-bounded computations, using quadratically more space. The next two sections show more efficient techniques to separate nondeterministic time and space.

We begin with a translational lemma under composition of complexity functions.

Lemma 5 (Function-composition translation for space) *Let $s_1(n) \geq n$, $s_2 \geq n$ and $f \geq n$ be complexity functions, and assume f is space-constructible. For $X = D$ or N, $XSPACE(s_1) \subseteq XSPACE(s_2)$ implies $XSPACE(s_1 \circ f) \subseteq XSPACE(s_2 \circ f)$.*

Proof. The structure of the proof is suggested by following the arrows in this "commutative diagram":



Suppose that $XSPACE(s_1) \subseteq XSPACE(s_2)$ and (Σ, L) is accepted by some M in space $s_1 \circ f$. We want to show that $L \in XSPACE(s_2 \circ f)$. First 'translate' L to $XSPACE(s_1)$ by padding. More precisely, the padded version of L is

$$L' = \{x\$^i : x \in L,\ |x\$^i| = f(|x|)\}$$

where \$ is a new symbol not in Σ. First we demonstrate that L' is in $XSPACE(s_1)$: on input $x\$^i$ of length n we check if $|x\$^i| = f(|x|)$ using space at most n, since f is space-constructible. If not, reject; else we simulate M on input x, and accept iff M accepts. Clearly the space used is at most $\max\{n, s_1(f(|x|))\} = s_1(n)$, as desired. Therefore, by assumption, L' is accepted by some M$'$ in space s_2. Next we demonstrate that L is in space $XSPACE(s_2 \circ f)$: on input x of length n, construct $x\$^i$

such that $|x\$^i| = f(|x|)$ using space $f(n)$. Now simulate M′ on $x\$^i$ using $s_2(|x\$^i|) = s_2(f(n))$ space. **Q.E.D.**

We illustrate the use of this lemma in the next result. In the proof, we use the fact that n^r for any rational number $r \geq 1$ is space-constructible (see Exercises in chapter 2).

Lemma 6 (Ibarra [14]) *For all reals $r > s \geq 1$, $NSPACE(n^r) - NSPACE(n^s) \neq \emptyset$.*

Proof. Choose positive integers a, b such that

$$r > \frac{a+1}{b} > \frac{a}{b} > s.$$

Note that $a > b \geq 1$. For the sake of contradiction, assume that $NSPACE(n^r) = NSPACE(n^s)$. Then

$$NSPACE(n^{(a+1)/b}) \subseteq NSPACE(n^{a/b}).$$

From this inclusion, an application of the previous translation lemma with $f(n) = n^{(a+1)b}$ and also with $f(n) = n^{ab}$, yields (respectively)

$$\begin{aligned} NSPACE(n^{(a+1)^2}) &\subseteq NSPACE(n^{a(a+1)}), \\ NSPACE(n^{a(a+1)}) &\subseteq NSPACE(n^{a^2}). \end{aligned}$$

Hence

$$NSPACE(n^{(a+1)^2}) \subseteq NSPACE(n^{a^2})$$

We now claim that for any $k \geq 2$ that is a power of two, the inclusion

$$NSPACE(n^{(a+1)^k}) \subseteq NSPACE(n^{a^k})$$

holds. The basis case has just been shown. For the induction, assume the inclusion holds for k. Then we can deduce

$$NSPACE(n^{(a+1)^{2k}}) \subseteq NSPACE(n^{a^{2k}})$$

by two applications of the translation lemma (the reader should verify this). If we choose $k \geq a$, then $(a+1)^k \geq 2a^k + 1$ for $a > 1$. Thus

$$\begin{aligned} DSPACE(n^{1+2a^k}) &\subseteq NSPACE(n^{1+2a^k}) \\ &\subseteq NSPACE(n^{(a+1)^k}) \\ &\subseteq NSPACE(n^{a^k}), \quad \text{(by what we had just shown)} \\ &\subseteq DSPACE(n^{2a^k}) \end{aligned}$$

where the last inclusion follows from Savitch's theorem. But $DSPACE(n^{1+2a^k}) \subseteq DSPACE(n^{2a^k})$ contradicts the deterministic space hierarchy theorem. **Q.E.D.**

This result will be improved in the next section. We now prove another translational lemma due to Savitch.

Definition 2 *Let $s(n) < \infty$ be a complexity function that is defined for all sufficiently large n. We say s is a moderately growing if it is unbounded, $s(n) \neq O(1)$, and there is a constant c such that, eventually, $s(n) \leq c \cdot s(n-1)$.* ■

Observe that the functions in $\log n$, $\log^{O(1)} n$, $n^{O(1)}$ and $O(1)^n$ are moderately growing. However functions such as 2^{2^n} and 2^{n^k} $(k > 1)$ are not.

Theorem 7 (Upward translation of space) *Let s, s' be complexity functions. If s is moderately growing and space-constructible, and if $s(n) \leq s'(n) < \infty$ for all n, then*

$$NSPACE(s) \subseteq DSPACE(s) \Longrightarrow NSPACE(s') \subseteq DSPACE(s').$$

Proof. The proof is similar in structure to that for the translational lemma. Let (Σ, L) be accepted by some nondeterministic M in space s'. We shall show that L is in $DSPACE(s')$. Again we translate L to a related problem in $NSPACE(s)$ as follows: let \$ be a new symbol not in Σ and define the following padded version of L:

$$L' = \{x\$^i : \text{ M accepts } x \text{ in space } \ s(|x\$^i|), i \geq 0\}.$$

Clearly, $L' \in NSPACE(s)$.

Since $NSPACE(s) \subseteq DSPACE(s)$, we infer that L' is accepted by some halting deterministic M$'$ in space s.

We now construct a deterministic M$''$ to accept L as follows: on input $x \in \Sigma^*$, simulate M$'$ on $x\$^i$ for $i = 0, 1, 2, \ldots$, until the first i such that M$'$ accepts $x\$^i$. At that point M$''$ accepts. Otherwise M$''$ runs forever.

Correctness of M$''$: It is easy to show that M$''$ accepts L. We next claim that M$''$ accepts in space $s'(n)$. To see this, let $x \in L$. If $s'(|x|) = s(|x|)$ then it is not hard to see that M$'$ accepts x in space $s'(|x|)$ space. Otherwise, $s'(|x|) > s(|x|)$ and there is a smallest $i > 0$ such that

$$s(|x\$^i|) \geq s'(|x|) > s(|x\$^{i-1}|).$$

Note that i is well-defined since s is unbounded and s' is finite. Now for any $j = 0, \ldots, i-1$, $s(|x\$^j|) < s'(|x|)$. Hence if M$'$ accepts $x\$^j$, then M$''$ accepts x in less than $s'(|x|)$ space; otherwise, surely M$'$ accepts $x\$^i$. This is because we see that M accepts x in space $s(|x\$^i|)$, and by definition of L', $x\$^i \in L' = L(\text{M}')$. Hence M$''$ accepts in space at most $s(|x\$^i|)$. But since s is moderately growing, there is some $c \geq 1$ such that $s(|x\$^i|) \leq cs(|x\$^{i-1}|)$. This proves that M$''$ accepts in space $cs(|x\$^{i-1}|)$. The claimed bound follows by space compression. This completes the proof. **Q.E.D.**

Corollary 8 *If NLOG $\subseteq$ DLOG then LBA=DLBA.*

Similarly, we can prove translational lemmas for time complexity (Exercises) and deduce:

$$\text{If } NP \subseteq P \text{ then } NEXPT \subseteq DEXPT.$$

These upward translation results raises the possibility of some form of 'downward translation'. Our next result may be regarded as partial downward translation. It involves the so-called *tally* or *contentless languages*: these are languages over a single letter alphabet, say $\{1\}$. In the remainder of this section, let Σ be any alphabet with $k \geq 2$ letters. We might as well assume $\Sigma = \{1, 2, \ldots, k\}$. For any $w \in \Sigma$, let $tally(w) \in \{1\}^*$ denote the unary representation of the integer w (regarded as a k-adic number). Then for any language (Σ, L), define the language $(\{1\}, tally(L))$ where

$$tally(L) = \{tally(w) : w \in L\}.$$

Conversely, define the function *untally* that takes any unary word $w \in \{1\}^*$ to its k-adic representation $untally(w) \in \Sigma^*$. For any tally language $(\{1\}, L)$, define the language $(\Sigma, untally(L)\}$ where

$$untally(L) = \{untally(w) : w \in L\}.$$

Thus *tally* and *untally* are inverses of each other.

Lemma 9 (Space translation for tally languages) *Let $X = N$ or D, let L be a language and f a complexity function with $f(n) \geq n$. Then $L \in XSPACE(f(O(n)))$ iff $tally(L) \in XSPACE(f(O(\log n)))$.*

Proof. If (Σ, L) is accepted in space $f(O(n))$ by some machine M then we can accept $tally(L)$ as follows: on input 1^n, we first compute $w \in \Sigma^*$ such that $\nu(w) = n$. This takes space $|w| = O(\log n)$. Now we simulate M on w, taking space $f(O(\log n))$. Conversely, if $tally(L)$ is accepted by some N in space $f(O(\log n))$ then we could try to accept L as follows:

> On input $w \in \Sigma^*$, compute $tally(w)$ and then simulate N on $tally(w)$ using space $f(O(\log(|tally(w)|))) = f(O(|w|))$.

The only problem is that $untally(w)$ needs space $O(1)^{|w|}$. To circumvent this, because $untally(w)$ is 'contentless', it suffices to keep track of the position of the input head of N on the virtual input $tally(w)$. The space necessary to keep track of the head position is $O(|w|)$ which is order of $f(O(|w|))$ since $f(n) \geq n$. **Q.E.D.**

We then obtain the following weak downward translation:

Corollary 10 (Savitch) *If $DLBA = LBA$ then $DLOG|\{1\} = NLOG|\{1\}$.*

Proof. Let L be a tally language in $NLOG = NSPACE(O(\log n))$. Then the above lemma implies that $untally(L) \in LBA = NSPACE(O(n))$. So by assumption $untally(L) \in DLBA$. Another application of the lemma shows that $L = tally(untally(L)) \in DLOG$. **Q.E.D.**

Combining the upward and weak downward translation results, we conclude that

$$DLBA = LBA \iff DLOG|\{1\} = NLOG|\{1\}.$$

Similar results relating to time complexity of tally languages can be shown.

6.4 Separation for Nondeterministic Classes

The reader may verify that the proof of Ibarra in the last section fails for nondeterministic time. This situation was first remedied by Cook [6] who showed the analogue of Ibarra's result [14]: for all reals $r > s \geq 1$,

$$NTIME(n^r) - NTIME(n^s) \neq \emptyset.$$

Cook's technique was generalized to a very strong form by Seiferas, Fischer and Meyer [27, 28]. Unfortunately their original proof is quite involved (using a form of recursion theorem for nondeterministic time – see Exercises). Simpler (but still delicate) proofs have been found by Žák [32] and by Li [16]; both these proofs have the added bonus of providing a tally language to separate the classes, answering an open question in [27]. Here we follow the proof of Žák.

We require some preparatory results. First, we note a simple but important consequence of the nondeterministic tape reduction theorem of Book, Greibach and Wegbreit (chapter 2):

Lemma 11 *For any alphabet Σ, there is a 2-tape universal acceptor $U = \{U_0, U_1, \ldots\}$ for the class $RE|\Sigma$ such that for each nondeterministic acceptor* M *over the input alphabet Σ, and each complexity function $t(n) > n$, if* M *accepts in time $t(n)$ there exist infinitely many indices i such that U_i accepts $L(\mathrm{M})$ in time $O_{U,\mathrm{M}}(t(n))$.*

Proof. For any machine N accepting in time $t(n) > n$, it follows from the nondeterministic tape reduction theorem that there is a 2-tape machine M accepting $L(\mathrm{N})$. It is seen from our standard construction of a universal machine U for $RE|\Sigma$ that there are infinitely many indices i such that $L(U_i) = L(\mathrm{M})$ and U_i accepts in time $c \cdot t(n)$ for some c that depends on M (rather than on i) and U. **Q.E.D.**

So the point of this lemma is that we can efficiently simulate any multitape acceptor M by infinitely many 2-tape versions as given by a universal machine U.

Let us fix $\Sigma = \{0, 1\}$ and U as in the above lemma. For any complexity function t and any Turing acceptor M, define the *t-cutoff* language defined by M to be

$$L^t(\mathrm{M}) = \{x \in \{0,1\}^* : \ \mathrm{M} \textit{ accepts } x \textit{ in time } t(|x|)\}.$$

Note that $L^t(\mathrm{M}) \subseteq L(\mathrm{M})$ and $L^t(\mathrm{M})$ is not necessarily in $NTIME(t)$ unless t is time-constructible or if M accepts in time t. Relative to the universal machine U we define:

$$NTIME_U(t) = \{L(U_i) : U_i \textit{ accepts in time } t\}$$
$$NTIME_U^{cut}(t) = \{L^t(U_i) : i = 0, 1, \ldots\}$$

Discussion. We may call the classes $NTIME_U(t)$ and $NTIME_U^{cut}(t)$ *universal-time classes* (relative to U) since they refer to a 'universal' standard of time-keeping

as defined by the steps of U. In contrast, the usual classes may be called *local-time classes* since a time step as defined by a Turing machine ϕ_i is not quite comparable to that defined by another ϕ_j when the tape alphabets and state sets of ϕ_i and ϕ_j are different. The connection between universal and local time is as follows: for each i, there is a constant $O_i(1)$ such that each step of ϕ_i is simulated by $O_i(1)$ steps of U_i. Note that the universal-time classes $NSPACE_U(t)$ do not in general enjoy the linear speedup property: it is not clear that $NTIME_U(t)$ is equal to $NTIME_U(2t)$, for instance. However, by the linear speedup theorem for local-time classes we can conclude that

$$NTIME(t) = NTIME_U(O(t)).$$

The crux of the diagonal process in our proof is captured in the following somewhat technical result. The statement of the result is somewhat long but its proof is not much longer.

Lemma 12 *Let $K = K|\Sigma$ be a class and U be any universal acceptor for K. Suppose $1 \in \Sigma$, and there exist languages (Σ, L) and (Σ, D), and functions $\alpha : L \to \mathbf{N} = \{0, 1, 2, \ldots\}$ and $\beta : L \to \mathbf{N}$ with the following property: For all $x \in L$ and for all j $(0 \leq j \leq \beta(x))$,*

$$x1^j \in D \iff \begin{cases} x1^{j+1} \in U_{\alpha(x)} & \text{if } j < \beta(x) \\ x \notin U_{\alpha(x)} & \text{if } j = \beta(x). \end{cases} \tag{6.3}$$

If α is an onto function then $D \notin K$.

Let us call L the 'unpadded' set and D the 'diagonal' set. Intuitively, deciding if an unpadded word x is in D is equivalent to the question whether $U_{\alpha(x)}$ accepts the padded strings $x1^j$ (for $j = 0, \ldots, \beta(x)$). On the other hand, $U_{\alpha(x)}$ accepts $x1^{\beta(x)}$ iff x is not in the diagonal set D. These two incompatible conditions imply $D \notin K$. Observe the highly stylized nature of this translation.

Proof. Aiming towards a contradiction, assume $D = L(U_i)$ for some $i \geq 0$. Since $\alpha : L \to \mathbf{N}$ is onto, let $x \in L$ such that $\alpha(x) = i$. If $\beta(x) = 0$ then we have that $x \in D$ iff $x \notin U_{\alpha(x)}$, contradicting our assumption that $D = L(U_{\alpha(x)})$. Observe that if $\beta(x) \geq 1$ then $x \in D$ iff $x1 \in L(U_{\alpha(x)})$ iff $x1 \in D$. If $\beta(x) \geq 2$ then $x1 \in D$ iff $x1^2 \in L(U_{\alpha(x)})$ iff $x1^2 \in D$. Repeating this, we see that $x, x1, x1^2, \ldots, x1^{\beta(x)}$ are all in D or none are in D. However, $x \in D$ iff $x1^{\beta(x)} \notin L(U_{\alpha(x)}) = D$, contradiction. **Q.E.D.**

Theorem 13 *If $t(n) > n + 1$ is time-constructible then there is a tally language D in $NTIME(t(n)) - NTIME_U^{cut}(t(n-1))$.*

Proof. We use the universal machine U for $RE|\Sigma$ of lemma 11. Let Σ be the unary alphabet $\{1\}$. Let U' be an 'efficient' universal acceptor for the class $NTIME_U^{cut}(t(n-1))|\Sigma$ obtained by simulating exactly $t(n-1)$ steps of U, accepting if and only if U

accepts within $t(n-1)$ steps. Note that U' can simulate U in realtime (i.e. step for step). However, U' needs an extra n steps to initially write onto another tape the word 1^{n-1} which then serves as input for the 'parallel' process to time-construct $t(n-1)$. Hence, U' on inputs of length n runs in time $n + t(n-1)$.

We inductively define an increasing sequence of integers

$$n_0 < n_1 < \cdots < n_i < \cdots$$

as follows: $n_0 = 1$ and

$$n_{i+1} = 1 + n_i + c_i$$

where c_i is the number of steps sufficient (for some fixed Turing machine) to *deterministically* simulate the behaviour of U_i' on input 1^{n_i}. Observe that c_i is well-defined because U_i' accepts in at most $t(n-1)$ steps on inputs of length n. (In general we expect c_i to be exponential in $t(n_i - 1)$, but no matter.) To apply the previous lemma, we define the 'unpadded set' L to be $\{1^{n_i} : i = 0, 1, \ldots\}$ and the functions $\alpha, \beta : L \to \mathbf{N}$ are given by:

$$\begin{aligned} \alpha(1^{n_i}) &= i, \\ \beta(1^{n_i}) &= 1 + c_i \end{aligned}$$

for $i \geq 0$. Finally, we define the diagonal language $D \subseteq \{1\}^*$ by constructing a machine M_D to accept D:

(A) On input 1^n, M_D computes in *phases* where in phase i ($i = 0, 1, \ldots$) M_D simulates U_i' on input 1^{n_i}. M_D stops in the middle of phase $k+1$ where k is defined by the inequality

$$n_{k+1} < n \leq n_{k+2}.$$

It is easy to organize the computation of M_D so that at the end of each phase, when M_D has just finished simulating U_i' on input 1^{n_i}, M_D has a copy of $1^{n_{i+1}}$ on a separate tape, ready to be used as input for the next phase. Furthermore, the time to carry out each phase is $O(n_{i+1})$ steps (actually, $O(n_i)$ steps suffice) for $i = 0, \ldots, k$, and the partial phase $k+1$ uses only $O(n)$ time. The total time to do phases 0 to k, including the partial $(k+1)$st phase, is

$$\sum_{i=0}^{k} O(n_{i+1}) + O(n) = \sum_{i=0}^{k} O(\frac{n}{2^i}) + O(n) = O(n).$$

Here we use the fact that $c_i \geq n_i$ and so $n_{i+1} > 2n_i$.

(B) If $n = n_{k+1} - 1$ then M_D can in $O(n)$ steps discover whether $1^{n_k} \in U_k'$. This is because $n_{k+1} - 1 > c_k$ and M_D can deterministically simulate U_k' on 1^{n_k} in at most c_k steps. M_D rejects iff U_k' accepts 1^{n_k}. Note that in this case, with $x = 1^{n_k}$,

$$1^n = x1^{\beta(x)} = x1^{1+c_k} \in L(\mathrm{M}_D) \iff x \notin L(U'_{\alpha(x)}).$$

(C) If $n < n_{k+1} - 1$ then M_D simulates U'_k on 1^{n+1}, accepting if and only if U'_k accepts. Thus with $x = 1^{n_k}$ and $j = n - n_k < n_{k+1} - n_k - 1 \leq c_k$,

$$x1^j \in L(\mathrm{M}_D) \iff x1^{j+1} \in L(U'_{\alpha(x)})$$

We observe that steps (A) and (B) take $O(n)$ time; step (C) takes $1+n+t(n)$ since U' takes $1 + n + t(n)$ steps on inputs of length $n + 1$. This implies $D = L(\mathrm{M}_D) \in NTIME(t+O(n)) = NTIME(t)$, by the speedup theorem for nondeterministic time. An application of the previous lemma shows $D \notin NTIME_U^{cut}(t(n-1))$. **Q.E.D.**

Note that step (C) is where the padding takes place: it reduces the query about $x1^j$ to one above x^{j+1}. Our main result of this section follows immediately.

Theorem 14 (Nondeterministic time hierarchy) *If $t(n) > n+1$ is time-constructible then there exists a tally language in*

$$NTIME(t(n)) - NTIME(o(t(n-1))$$

Proof. For any function $t'(n)$, if $t'(n) = o(t(n-1))$ then

$$NTIME(t') \subseteq NTIME_U^{cut}(t(n-1)).$$

An application of the previous theorem yields the desired separation. **Q.E.D.**

In the statement of this theorem, we need the '-1' in defining one of the complexity classes because we need to pad at least one symbol for the induction to go through. It is not known if this '-1' can be removed or if it is essential.

Now we can infer Cook's result that $NTIME(n^r) - NTIME(n^s) \neq \emptyset$ $(r > s \geq 1)$: first choose a rational number b such that $r > b > s$. The function n^b is "approximately" time-constructible in this sense: there exists a time-constructible function t such that $t(n) = \Theta(n^b)$ (Exercises). Clearly $(n+1)^s = o(t(n))$, so the preceding theorem implies $NTIME(t) - NTIME(n^s)$ (and hence $NTIME(n^r) - NTIME(n^s)$) is non-empty.

As another application of this theorem, we infer that $NTIME(n2^n) - NTIME(2^n)$ is non-empty. On the other hand, the theorem fails to decide whether $NTIME(n2^{2^n}) - NTIME(2^{2^n})$ is empty. This remains an open problem.

We now show a corresponding separation result for nondeterministic space [27]. The proof below employs the technique used in showing that nondeterministic space is closed under complementation.[4]

Theorem 15 *Let s_2 be space-constructible, $s_2(n) \geq \log n$. If $s_1(n) = o(s_2(n))$ then $NSPACE(s_2) - NSPACE(s_1) \neq \emptyset$.*

[4]The original proof of Seiferas-Fischer-Meyer is more involved. Immerman [15] attributes the idea of the present proof to M. Fischer.

Proof. Let U be a universal acceptor for all nondeterministic 1-tape Turing acceptors. We describe an acceptor M to diagonalize over each U_i: on input x of length n, we mark out $s_2(n)$ cells. For each accepting configuration C that fits inside these marked cells, we call a subroutine that uses Immerman's technique to unequivocally check if C can be reached from the initial configuration. (Recall that this means that there is at least one terminating computation path and further all terminating computation paths accept or all reject.) If any subroutine call loops, then we loop; if any accepts, then we reject; if all reject, then we accept.

By now, it is a simple exercise to show that $L(M)$ separates $NSPACE(s_2)$ from $NSPACE(s_1)$. **Q.E.D.**

6.5 Applications to Lower Bounds

Informally, if $L \leq L'$ where $\leq$ denotes some efficient reducibility then the complexity of L is at most the complexity of L' plus the complexity of the reducibility $\leq$. For a simple illustration of such results, we consider many-one reducibilities. First, a definition.

Definition 3 *For any transformation $t : \Sigma^* \to \Gamma^*$, and f a complexity function, we say that t is f-bounded if for all x, $|t(x)| \leq f(|x|)$.* ■

Lemma 16 *Let L be many-one reducible to L' via a transformation g and $L' \in X\text{-}TIME\text{-}SPACE(t, s)$ where $X = N$ or D, and let t, s be non-decreasing complexity functions. If g can be computed in time $u(n)$ and is $f(n)$-bounded then*

(i) $L \in XTIME(t(f(n)) + u(n))$ *and*

(ii) $L \in X\text{-}TIME\text{-}SPACE(u(n)t(f(n)), s(f(n)) + \log f(n))$.

Proof. Let M accept L' in time t and space s, and let T be the log-space transducer that transforms L to L'. It is straightforward to show (i) by constructing an acceptor N for L: on input x, N simulates T on x to obtain $T(x)$; then it simulates M on $T(x)$, accepting if and only if M accepts. Since $|T(x)| \leq f(|x|)$ and t is non-decreasing, the desired time bound of $t(f(n)) + u(n)$ on N follows immediately. Note that if M is deterministic then so is N.

We show (ii). To achieve a space bound of $s(f(n))) + \log f(n)$, we modify the above construction of M by using the technique from chapter 4 (section 2): simulate the acceptor M on input $T(x)$ without keeping the entire input string in storage, but use T as a subroutine to (re)compute each symbol of $T(x)$ as needed by M. To do this, we need $O(\log f(n))$ space to represent the position of the input head of M on $T(x)$. The space bound of (ii) follows from this '$O(\log f(n))$' plus the $s(f(n))$ space used by M on $T(x)$. **Q.E.D.**

The reader can find analogous results for other types of reducibilities.

It is desirable in applying the lemma to have a small bounding function f. As seen in the transformations in chapter 5, f is typically linear, $f(n) = O(n)$; this simply means that t belongs to the class **Llin** of log-linear transformations defined in chapter 4 (section 2).

The principal application of such a lemma is to obtain lower bounds on *specific* languages. Meyer and Stockmeyer [20, 29, 21] and Hunt [13] were the first to infer such lower bounds on natural computational problems. The basic structure of such proofs is outlined next. Assume that we want to prove a lower bound of $s(n)$ on the deterministic space-complexity of a language L_0.

(a) *Show that L_0 is hard for some complexity class.* Choose a suitable class K of languages such that each $L \in K$ is efficiently reducible to L_0. For instance, suppose there is a $k \geq 1$ such that each $L \in K$ is many-one reducible to L_0 via a n^k-bounded log-space transformation.

(b) *Infer the lower bound by appeal to a separation result.* Assume that we want to show a space lower bound. Suppose that $s(n) \geq \log n$ is non-decreasing and there is a separation result for $(K, DSPACE(s))$. Then we claim that $s'(n) = s(n^{1/k})$ is an i.o. lower bound on the space-complexity of L_0. For the sake of contradiction, assume otherwise that $L_0 \in DSPACE(s')$. By our choice of K, there exists a language $L_1 \in K - DSPACE(s)$. But the above lemma implies that L_1 can be accepted in space $O(s'(n^k) + \log n) = O(s(n))$, contradiction.

Note that in this outline, we normally only have to show step (a) since step (b) is usually routine[5]. Since step (a) involves showing L_0 to be K-hard, this explains why showing a problem to be hard for a class is often called a 'lower bound proof'.

The remainder of this section illustrates such applications.

Lower bound on the fullness problem for regular languages. Recall the problem FULL = FULL$(+, \cdot, *)$ of checking if a given regular expression α denotes the set $\{0,1\}^*$. In chapter 5 we showed that the problem FULL is hard for $NSPACE(n)$ under log-linear transformations. It easily follows from the outline (a) and (b), by appealing to the nondeterministic space hierarchy theorem, that

$$\text{FULL} \notin NSPACE(o(n)).$$

Hence every nondeterministic acceptor for FULL must use more than linear space infinitely often. Stated in isolation, this statement should be appreciated for its depth since, as remarked in chapter 1, it is a statement about all imaginable (but mostly unimaginable) Turing machines that accept FULL. Yet, because of the long development leading up to this point, this statement may seem rather easy.

[5]Or rather, radically new separation results do not seem to be easy to derive – and so step (b) is typically an appeal to one of the separation theorems we have shown here.

Similarly, using the log-linear reductions in section 5 of chapter 5, we conclude that any deterministic acceptor M for the problem FULL$(+, \cdot, *, ^2)$ uses space more than c^n infinitely often, for some $c \geq 0$.[6]

Nondeterministic time lower bounds for complements of languages. It follows from the log-linear transformation shown in chapter 5 that any nondeterministic acceptor M for the problem INEQ = INEQ$(+, \cdot, ^2)$ uses time more than c^n infinitely often, $c > 0$. Now consider what is essentially[7] the complementary problem: let EQUIV = EQUIV$(+, \cdot, ^2)$ denote the set of pairs (α, β) of $\{+, \cdot, ^2\}$-expressions encoded over the binary alphabet such that $L(\alpha) = L(\beta)$. We would like a lower bound on EQUIV based on a lower bound on INEQ. Towards this end, we use a nice separation result attributed to Young [30].

Lemma 17 *If t is time-constructible then*

$$NTIME(n \cdot t(n)) - co\text{-}NTIME(t(n)) \neq \emptyset.$$

Proof. Let U be an efficient universal acceptor for the characteristic class $NTIME(t)|\Sigma$ where $\Sigma = \{0, 1\}$. So for any $(\Sigma, L) \in NTIME(t)$ there are infinitely many indices i such that U_i accepts L in time $O_L(t(n))$. Using the usual encodings, we may assume that U_i accepts in time $c \cdot |i| \cdot t(n)$ for some constant $c = c(U) > 0$ that does not depend on i or n. Let L_0 consist of those words x such that U_x accepts x. Then it follows that L_0 can be accepted in $n \cdot t(n)$ using a direct simulation of U. To prove the lemma, it remains to show that $L_0 \notin co\text{-}NTIME(t(n))$. Suppose otherwise, $L_0 \in co\text{-}NTIME(t(n))$. Then let $co\text{-}L_0$ be accepted by U_x for some x; this means that $x \in co\text{-}L_0$ iff U_x accepts x. On the other hand, by definition of L_0, we have $x \in co\text{-}L_0$ iff U_x does not accept x. Contradiction. **Q.E.D.**

Lemma 18 *The problem* EQUIV *requires nondeterministic time greater than c^n i.o., for some $c > 0$.*

Proof. The above lemma shows the existence of an L in $NTIME(n2^n) - co\text{-}NTIME(2^n)$. Since L is in $NEXPT$, chapter 5 shows that L is many-one reducible to INEQ via some log-linear transformation t. Furthermore, t has the property that for all x, $t(x)$ represents a well-formed pair of $\{+, \cdot, ^2\}$-expressions. This implies $co\text{-}L$ is many-one reducible to EQUIV via t. Choose $c = 2^{1/b}$ where t is bn-bounded, $b > 0$. Assume for the sake of contradiction that EQUIV $\in NTIME(c^n)$. Then lemma 16 (i), implies that $co\text{-}L \in NTIME(c^{bn}) = NTIME(2^n)$. This contradicts our assumption $L \notin co\text{-}NTIME(2^n)$. **Q.E.D.**

[6] One is tempted to say, this problem is not in $NSPACE(o(O(1)^n))$. But notice that we have not defined the meaning of $o(E)$ where E is a big-Oh expression.

[7] EQUIV only differs from the complement of INEQ by being restricted to words that represent pairs of well-formed expressions.

6.6 Weak Separation

The following result of Book [2] is useful in distinguishing between two classes:

Theorem 19 *Let J, K be classes and $\leq$ a reducibility. Suppose K has a complete language under $\leq$-reducibility. If J is the limit of the some strictly increasing sequence*

$$J_1 \subset J_2 \subset J_3 \subset \cdots$$

where each J_i is closed under $\leq$-reducibility, then $J \neq K$.

Proof. Let L be K-complete under $\leq$-reducibility. If $J = K$ then $L \in J_i$ for some i. By the basic inclusion lemma, chapter 4, $K \subseteq J_i$. This contradicts the fact that J properly contains J_i. **Q.E.D.**

Of course, this theorem achieves only a weak separation between J and K, since it does not tell us if $J - K \neq \emptyset$ or $K - J \neq \emptyset$ (although one of these must hold true). We illustrate an application of the lemma:

Theorem 20 *PLOG is distinct from NP and from P.*

Proof. We know that *NP* has complete languages under $\leq_m^L$. The technique (in chapter 4) for showing that *DLOG* is closed under log-space reducibility easily shows that $DSPACE(\log^k n)$ is closed under $\leq_m^L$ reducibility. *PLOG* is the limit of the increasing sequence

$$DSPACE(\log n) \subseteq DSPACE(\log^2 n) \subseteq DSPACE(\log^3 n) \subseteq \cdots.$$

By the deterministic space hierarchy theorem, we know that this sequence is strictly increasing. Hence the previous theorem applies showing that $NP \neq PLOG$. Similarly, since P also has complete languages under $\leq_m^L$, we also conclude $P \neq PLOG$. **Q.E.D.**

Consider the following attempt to show $DLOG \neq P$: we can define the strictly increasing sequence

$$DTIME(n) \subseteq DTIME(n^2) \subseteq DTIME(n^3) \subseteq \cdots$$

and using $\leq_m^L$ as our reducibility, etc., we find that one of the conditions of the lemma fails (where?).

The following chart from Book[2]: shows some of the known distinctions between the various classes.

	$PLOG$	(a)	$DLOG$	NP	(b)	(c)	P	(d)
$PLOG$								
$(a) = DSPACE(\log^k n)$	$\neq$							
$DLOG$	$\neq$	$\neq$						
NP	$\neq$	?	?					
$(b) = NTIME(n^k)$	$\neq$	$\neq$	$\neq$	$\neq$				
$(c) = NTIME(n+1)$	$\neq$	$\neq$	$\neq$	$\neq$	$\neq$			
P	$\neq$	?	?	?	$\neq$	$\neq$		
$(d) = DTIME(n^k)$	$\neq$	$\neq$	$\neq$	$\neq$	?	?	$\neq$	
$DTIME(n+1)$	$\neq$	$\neq$	$\neq$	$\neq$	$\neq$	$\neq$	$\neq$	$\neq$

Notes: k is any integer greater than 1. An entry "?" indicates that it is not known if the two classes are equal or not.

6.7 Strong Separation

In section 5 we show the 'infinitely often' (i.o.) type of lower bound on the complexity of languages. In this section we consider the 'almost every' (a.e.) version. It is important to realize that strong separation results only make sense for characteristic classes. Geske and Huynh have proved strong hierarchy theorems in this sense.

Observe that most natural problems do not seem to possess non-trivial a.e. lower bounds. For instance, the reader can easily be convinced after checking some cases that all *known* NP-complete problems have infinite subsets that are easily recognizable in polynomial time. It is unknown whether there are NP-complete problems without this property. One of the few examples of a natural problem that *may* possess a non-trivial a.e. lower bound is primality testing: it is unknown if there is an infinite subset of the prime numbers that can be recognized in deterministic polynomial time. More precisely[8], is there a language $L \in P$ such that $L \cap Primes$ is infinite?

Meyer and McCreight [19] shows that for any space-constructible complexity function $s(n) \geq n$, there exists a language whose *running* space complexity is lower bounded by $s(n)$ (a.e.). The following theorem adapts the proof for *accepting* space complexity and avoids the assumption $s(n) \geq n$.

Theorem 21 *Let s be an non-decreasing, unbounded space-constructible function. Then there exists an infinite language L_0 in $DSPACE(s)$ such that if* N *is any acceptor for L_0,*

$$AcceptSpace_{\mathrm{N}}(n) = \Omega(s(n)).$$

[8]Recently Goldwasser and Killian show that there is an infinite subset that can be recognized in *expected* polynomial time. Expected complexity classes will be considered in a later chapter.

This result can be regarded as a strong version of the deterministic space hierarchy theorem for characteristic classes. Similar results have been obtained in [3, 31]. These proofs use ideas from the earlier work of Rabin [23] and Blum [1].

Proof. The language L_0 will be defined by describing an acceptor M for it. Let $U = \{U_0, U_1, \ldots\}$ be a universal machine for the class $RE|\{0,1\}$. The basic idea is for M to diagonalize over each U_i that accepts in space $cs(n)$ for some $c > 0$. More precisely, suppose the input to M is the binary string i (regarded as an integer when convenient). First we mark out exactly $s(|i|)$ tape squares on each of its work-tape. In the subsequent simulation, the computation will never exceed these marked squares. Hence $L(\mathrm{M})$ clearly is in $DSPACE(s)$.

In the following description, we see that M on input i will compute an index $\delta(i) \geq 0$ such that

$$\begin{aligned} \delta(i) = \mathit{odd} &\Rightarrow \mathrm{M} \text{ accepts } i \text{ and } U_{\lfloor \delta(i)/2 \rfloor} \text{ rejects } i. \\ \delta(i) = \mathit{even} &\Rightarrow \mathrm{M} \text{ rejects } i \text{ and } U_{\lfloor \delta(i)/2 \rfloor} \text{ accepts } i. \end{aligned}$$

We say that the index $j \geq 0$ is 'cancelled' by input i if $j = \lfloor \delta(i)/2 \rfloor$; thus if j is cancelled, then $L(U_j) \neq L(\mathrm{M})$. Note that we try to cancel each j twice but of course, this is impossible if $L(U_j)$ is a trivial language. The binary string $\delta(i)$ will be written on a tape reserved for this purpose (say tape 1) at the end of the computation on i. So the machine M is doing double duty: as an acceptor as well as some kind of transducer (but only on the side).

Define $\hat{s}(n) = \max\{n, s(n)\}$ and let C_i denote the set

$$C_i = \{\delta(j) : 0 \leq j \leq \hat{s}(|i|)\}$$

Note that $C_i \subseteq C_{i+1}$ and we define C_∞ to be the union over all C_i.

Let the input to M be i. Our goal is to cancel some $k \notin C_i$. To do this, we successively submit each $k = 0, \ldots, \hat{s}(|i|)$ to the following 'test': we say k passes the test if it satisfies the following three conditions.

(i) First $k \notin C_i$.
(ii) If $k = \mathit{odd}$ then $U_{\lfloor k/2 \rfloor}$ does not accept i in space $s(|i|)$. This covers three possibilities: the machine either tries to use more than $s(|i|)$ space, or rejects within $s(|i|)$ space, or loops within $s(|i|)$ space.
(iii) If $k = \mathit{even}$ then $U_{\lfloor k/2 \rfloor}$ accepts i in space $s(|i|)$.

We claim that this test can be done in $s(|i|)$ space for each $k = 0, \ldots, \hat{s}(|i|)$. To do part (i) of the test, we check if $k = \delta(j)$ for each $j = 0, \ldots, \hat{s}(|i|)$. For each j, we determine $\delta(j)$ by recursively calling M on input j. (Note that the restriction that $j \leq \hat{s}(|i|)$ means that $j \leq |i| < i$ and hence there is no problem of self-reference; this is the reason we use $\hat{s}(n)$ instead of $s(n)$ in defining C_i.) Since $s(n)$ is non-decreasing, we do not use more than $s(|i|)$ space. For part (ii) and (iii) of the test,

we see that M can decide whether $U_{\lfloor k/2 \rfloor}$ accepts i within space $s(|i|)$: in particular, we must be able to detect when $U_{\lfloor k/2 \rfloor}$ loops within space $s(|i|)$ (which we know how to do from chapter 2, section 9). Thus the test can indeed be carried out within the marked space.

If any of these k passes the test, we write this k on tape 1 (so $\delta(i) = k$) and we accept iff k is odd. Otherwise, every such value of k fails the test and we write 0 on tape 1 and reject the input i. (We may assume that the index 0 corresponds to a machine that accepts all its inputs.)

This completes the description of M. Now we must prove that our construction is correct. First we claim that M accepts infinitely many inputs because for each index k corresponding to a machine U_k that rejects all its inputs in constant space, there is some input $x = x(k)$ such that M on input x accepts and outputs $\delta(x) = 2k + 1$. This claim amounts to showing that $2k + 1 \in C_\infty$. Choose the smallest x such that C_x contains each $j < 2k + 1$ that will eventually be cancelled, i.e., C_x contains $C_\infty \cap \{0, \ldots, 2k\}$, and $\hat{s}(|x|) \geq 2k + 1$. (Such a choice can be found.) Then M on input x will eventually test $2k+1$, and by choice of k, it will detect that U_k does not accept x within the space $s(|x|)$. Hence M accepts with output $2k + 1$ as desired.

Let N be any acceptor satisfying

$$\mathit{AcceptSpace}_{\mathrm{N}}(n) \leq c \cdot s(n) \text{ (i.o.)} \tag{6.4}$$

for each choice of $c > 0$. It remains show that N cannot accept the same language as M.

Since we have shown that M accepts infinitely many inputs, we may assume that $\mathit{AcceptSpace}_{\mathrm{N}}(n)$ is defined for infinitely many values of n. Now there is some $h \geq 0$ such that N accepts the language $L(U_h)$. By usual properties of universal machines,

$$\mathit{AcceptSpace}_{U_h}(n) \leq c_0 \cdot \mathit{AcceptSpace}_{\mathrm{N}}(n) \tag{6.5}$$

for some constant $c_0 > 0$. Choosing $c = 1/c_0$, inequalities (6.4) and (6.5) imply that $\mathit{AcceptSpace}_{U_h}(|x|) \leq c \cdot c_0 s(|x|) = s(|x|)$. So it suffices to show that h is cancelled (since $\lfloor \delta(x)/2 \rfloor = h$ implies that M on input x will accept iff U_h does not accept x in space $s(|x|)$ iff $x \notin L(U_h) = L(\mathrm{N})$.)

Since s is unbounded and non-decreasing, we may choose the smallest input x such that

(a) $\hat{s}(|x|) \geq 2h$,

(b) C_x contains all indices $k < 2h$ that are eventually cancelled, $C_\infty \cap \{0, \ldots, 2h - 1\} \subseteq C_x$.

(c) $x \in L(U_h)$.

Consider the action of M on such an input x: M will test each $k = 0, 1, \ldots, 2h-1$ and, by choice of x, each such k will fail the test. Thus M on input x will eventually

test $2h$. If $2h \in C_x$ then h is cancelled already, and we are done. If $2h \notin C_x$ then our test calls for M to simulate U_h running on input x. But we just showed that U_h on $|x|$ uses at most $s(|x|)$ space. Since $x \in L(U_h)$, the test succeeds with M rejecting x and outputting $\delta(x) = 2h$. Thus h is cancelled after all. **Q.E.D.**

The preceding theorem shows a language L_0 that is hard (a.e.) for the characteristic class $DSPACE(s)$. In order to infer that problems reducible to L_0 are also hard (a.e.), we need some converse of lemma 16. First, we define a language (Σ, L) to be *invariant under padding* if there is a symbol $\# \in \Sigma$ such that for all $x \in \Sigma^*$, $x \in L$ iff $x\# \in L$. The following is due to Stockmeyer [30]. The present proof applies to running space only:

Theorem 22 *Suppose L is reducible to L' via some log-linear transformation and L' is invariant under padding. If $s(n) \geq \log n$ is non-decreasing and the running space for L is at least $s(n)$ (a.e.) then for some $c > 0$, the running space for L' at least $s(cn)$ (a.e.).*

Proof. Suppose to the contrary that there is an acceptor N for L' such that for all $c > 0$, N uses running space less than $s(cn)$ (i.o.). Let $L \leq_m^{Llin} L'$ via some log-linear transformation t where $|t(x)| \leq b|x|$ for integer $b \geq 1$. We obtain an acceptor M for L from any acceptor N for L' as follows. On input x:

For $i = 0, 1, 2, \ldots$ **do**:
  For $j = 0, 1, 2, \ldots, b|x|$, **do**:
    If $|t(x)\#^j| > b|x|$
      then exit current for-loop (i.e. go to $i + 1$);
    Simulate N on input $t(x) \cdot \#^j$ using only i space;
    If N attempts to use more than i space,
      then continue current for-loop (i.e. go to $j + 1$);
    If N halts, **then** accept if N accepts, else reject;
  End
End

The outer for-loop is potentially infinite since i can grow arbitrarily large, but the inner for-loop is bounded by $b|x|$. It should be clear that M accepts L. Let $reduce(x)$ denote the set

$$\{t(x) \cdot \#^j : |t(x)\#^j| \leq b|x|,\ j \geq 0\}.$$

The basic idea of M is to reduce the decision on x to deciding a member of $reduce(x)$ for which N requires the least space. Because of padding, we see that $x \in L$ implies $reduce(x) \subseteq L'$ and $x \notin L$ implies $reduce(x) \cap L' = \emptyset$. To see the space usage of M, suppose $y \in reduce(x)$ requires the least space to compute. In the above simulation of N on $y = t(x) \cdot \#^j$, we assume that M does not explicitly store the word y, but uses counters to store the value j and the input head position on y. Then

$$RunSpace_{\mathrm{M}}(x) \leq RunSpace_{\mathrm{N}}(y) + O_1(\log |y|) \tag{6.6}$$

where the term $O_1(\log|y|)$ comes from storing the counters for j and the head position on y.

Choose $c = 1/b$. To derive our contradiction, we show that if N runs in less than $s(cn)$ (i.o.) then M will also use less than $s(n)$ space (i.o.), contradicting our assumption on L. Let E be an infinite set of 'easy lengths' for N. More precisely, for each input y with length $|y|$ in E, $RunSpace_{\rm N}(y) < s(cn)$. Let E' be the set of lengths of inputs for M that can be reduced to those with length is in E, i.e. for each input x with $|x| \in E'$ there is some $y \in reduce(x)$ with $|y| \in E$. Then (6.6) shows that for such x,

$$\begin{aligned} RunSpace_{\rm M}(x) &< s(c|y|) + O_1(\log|y|) \\ &= O_2(s(cb|x|)) = O_2(s(|x|)). \end{aligned}$$

Using linear compression of space, M can be modified to ensure that for all n in E',

$$RunSpace_{\rm M}(n) < s(n).$$

We also see that E' is infinite: for each $n \in E$, we have $\lceil cn \rceil \in E'$ since if $|x| = \lceil cn \rceil$ then some member of $reduce(x)$ has length n. This contradicts the assumption that the running space for L is $\geq s(n)$ (a.e.). **Q.E.D.**

The requirement that L' is invariant under padding cannot be omitted in the above theorem. To see this, suppose the language (Σ, L) requires running space $\geq s(n)$ (a.e.). Define the language

$$L' = \{xx : x \in L\} \cup \{x \in \Sigma^* : |x| = odd\}.$$

Clearly $L \leq_m^{Llin} L'$ but it is easy to design an acceptor for L' that uses no space for all inputs of odd length.

Lynch [17] shows that if a problem L is not in P then L contains a subset whose running time complexity is lower bounded by n^k (a.e.) for every $k \geq 1$; see Even, Selman and Yacobi [9] for a generalization of such results. Schnorr and Klupp [26] obtains a result similar to Lynch within a natural context: there is a subset of the language SAT that is a.e. hard. These results should remind the reader of the result of Ladner in chapter 4.

6.8 Inseparability Results

To complement the separation results, we now show some results that reveal limits to our attempts to separate complexity classes. These results are somewhat surprising because they only rely on certain rather simple properties that are seen to hold for typical complexity measures such as time and space. Blum [1] first formalized such properties of complexity measures.

Let $\Sigma = \{0, 1\}$ and let $U = \{U_i\}$ be a universal machine for some class $K = K|\Sigma$, and R be any language over the alphabet $\Sigma \cup \{\#\}$. We say that the pair (U, R) is a *Blum measure* for K if:

(B1) $x \in L(U_i)$ iff there is an m such that $i\#x\#m \in R$.

(B2) R is recursive and defines a partial function $r(i,x)$ in the sense that if $r(i,x)\downarrow$ (i.e. $r(i,x)$ is defined) then there is a unique $m \in \Sigma^*$ such that $i\#x\#m \in R$ and if $r(i,x)\uparrow$ (i.e. is undefined) then for all $m \in \Sigma^*$, $i\#x\#m \notin R$.

We also write $R_i(x)$ for $r(i,x)$. Intuitively, $R_i(x) = m$ implies that U_i on input x accepts using m units of some abstract resource. Note that if $x \notin L(U_i)$ then, consistent with our use of acceptance complexity, $R_i(x)\uparrow$. The reader may verify that we may interpret this abstract resource to be time, space, reversal or some other combinations (e.g. products) of these. One caveat is that for space resource, we must now say that the space usage is infinite or undefined whenever the machine loops (even if it loops in finite space).

The hierarchy theorems suggests a general theorem of this form: there is total recursive function $g(n)$ there exists for all total recursive functions $t(n)$, the pair

$$(DTIME(g \circ t), DTIME(t))$$

can be separated: $DTIME(g(t(n)) - DTIME(t(n)) \neq \emptyset$. Unfortunately, we now show that such a theorem is impossible without restrictions on $t(n)$. This is implied by the following theorem of Borodin [4, 18, page 148].

Theorem 23 (Gap lemma) *Let (U,R) be a fixed Blum measure for all recursively enumerable languages. For any total recursive function $g(n) > n$ there exists a total recursive function $t(n)$ such for all i, there are only finitely many n such that $t(n) < R_i(n) < g(t(n))$.*

Proof. We define $t(n)$ to be the smallest value m satisfying predicate

$$P(n,m) \equiv (\forall i)[m < R_i(n) < g(m) \Rightarrow n \leq i]$$

As usual, when an integer n appears in a position (e.g. $R_i(n)$) that expects a string, we regard it as its binary representation. Note that this definition of $t(n)$, if total and recursive, would have the desired properties for our theorem. First we show that the predicate $P(n,m)$ is partial recursive. $P(n,m)$ can be rewritten as $(\forall i)[n \leq i$ or $\neg(m < R_i(n) < g(m))]$, or

$$(\forall i < n)[m \geq R_i(n) \text{ or } R_i(n) \geq g(m)].$$

It is now clear that $P(n,m)$ is decidable, and so, by searching for successive values of m, we get a partial recursive procedure for $t(n)$. To show that this procedure is total recursive, we show that for any n, there is at least one value of m satisfying $P(n,m)$. Define $m_0 = 0$ and for each $i = 1, 2, .$, define $m_i = g(m_{i-1})$. Hence $m_0 < m_1 < \cdots$. Consider the gaps between $m_0, m_1, \ldots, m_{n+1}$: there are $n+1$ gaps so that at least

one of them $[m_j, m_{j+1}]$ $(j = 0, \ldots, n)$ does not contain a value of the form $R_i(n)$, $i = 0, \ldots, n-1$. Then $P(n, m_j)$ holds. **Q.E.D.**

For instance it easily follows that there exist complexity functions t_i $(i = 1, 2, 3)$ such that the following holds:

$$DTIME(t_1) = DTIME(2^{t_1})$$
$$DTIME(t_2) = NTIME(t_2)$$
$$DSPACE(t_3) = NSPACE(t_3)$$

Thus such pairs of classes are inseparable. Such results, like the union theorem mentioned in section 6, employ complexity functions that are highly non-constructible. Such *inseparability results* are instructive: it tells us that the constructibility conditions in our separation results cannot be removed with impunity. There are many other results of this nature (e.g., [24, 8]).

6.9 Conclusion

This chapter shows some basic separation results and uses translational methods to obtain further separations. We also illustrate techniques for stronger or weaker notions of separation. These results mostly apply to space and time classes: it would be satisfying to round up our knowledge in this 'classical' area by extending the results here to encompass reversal or simultaneous complexity classes. We also show that there are limits to such separations if we do not restrict ourselves to nice complexity functions.

Exercises

[6.1] Let t be any total time-constructible function. Show that there is time-constructible function t' such that $NTIME(t') - NTIME(t) \neq \emptyset$.

[6.2] (A distributed decrementing counter) Reconstruct Fürer's result stated in section 2. We want to simulate each $t_1(n)$-bounded machines in $t_2(n)$ steps. On input w, we want to simulate the machine M_w for $t_2(|w|)$ steps. To do this, we a construct a "decrementing-counter" initialized to the value of $t_2(|w|)$, and try to decrement this counter for each step of the simulation. The problem is that since we are restricted to $k \geq 2$ (which is fixed in Fürer's version of the theorem) tapes. So the counter must reside on one of the tracks of a work tape (which has to be used for the actual simulation of M_w. The idea is to use a "distributed representation of numbers" such that there are low order bits scattered throughout the representation (note that to decrement, it is easy to work on low order bits, and propagate the borrow to a higher order bit if necessary). So we form a balanced binary with nodes labeled by a digit between 0 and $B-1$ (B is a fixed base of the representation, and $B = 4$ suffices for us). A node at height $h \geq 0$ (the leaves are height 0) with a digit d represents a value of dB^h. The number represented is the sum of values at all nodes. Clearly there are lots of redundancy and all the leaves are low-order bits. We store the labels of the nodes in an in-order fashion so the root label is in the middle of the representation. We want to decrement at ANY leaf position. If necessary, borrows are taken from the parent of a node. When the path from any leaf to the root has only zero labels, then this representation is exhausted and we must redistribute values. Show that if we do a total of m decrements (as long as the tree is not exhausted) then the time is $O(m)$ on a Turing machine. Apply this to our stated theorem (we have to show that the tree is not too quickly exhausted and we must show how to reconstruct exhausted trees)

[6.3] Reprove the deterministic space hierarchy theorem for running complexity. The statement of the theorem is as before except that we do not assume s_2 to be space constructible.

[6.4] (a) (Hong) Show that the transformation $w \mapsto tally(w)$ (where $w \in \{1, \ldots, k\}^*$ and $tally(w) \in \{1\}^*$) can be computed in space-reversals $O(n, \log n)$ and also in space-reversals $O(\log n, n)$.
(b) Generalize this result by giving tradeoffs between space and reversals.

[6.5] Show that for t any total time-constructible function then there exists a deterministic 1-tape acceptor M that time-constructs some t' such that

$$L(\mathrm{M}) \in DTIME(t') - \bigcup_{j=0}^{\infty} NTIME(t(n+j)).$$

[6.6] Separate the following pairs of classes:
(a) $DTIME(2^n)$ and $DTIME(3^n)$
(b) $NSPACE(2^n)$ and $NSPACE(3^n)$
(c) $NSPACE(2^{2^n})$ and $DSPACE(2^{3^n})$.

[6.7] Show the translational lemma for time resource analogous to lemma 5. Conclude that if $NP \subseteq P$ then $NEXPT \subseteq DEXPT$.

[6.8] (I. Simon) Assume the one-way oracle machines (section 3, chapter 4) for this question. Show that Savitch's upward space translation result can be relativized to any oracle. More precisely, for any oracle A, and space-constructible and moderately growing function s, $NSPACE^A(s) \subseteq DSPACE^A(s)$ implies that that for all $s'(n) \geq s(n)$, $NSPACE^A(s') \subseteq DSPACE^A(s)$.

[6.9] Where does the proof of Ibarra's theorem break down for nondeterministic time complexity?

[6.10] Complete the proof of Cook's result that $NTIME(n^r) - NTIME(n^s) \neq \emptyset$ from section 4: It is enough to show that for any $b = k2^{-m} \geq 1$ where k, m are positive integers, there exists a time-constructible function $t(n) = \Theta(n^b)$. *Hint:* Show that in linear time, you can mark out $\Theta(n^{1/2})$ cells. Extend this to $\Theta(n^{2^{-m}})$, and use the time-constructibility of the function n^k for any integer $k \geq 1$ and an exercise in chapter 2.

[6.11] (Recursion theorem for nondeterministic time) Let $\Sigma = \{0, 1, \$\}$ and let $U = \{U_i\}$ be a universal machine for the class $RE|\Sigma$. For all indices i, if U_i accepts in time t_0 then there exists an index j such that U_j accepts the language

$$\{x : j\$x \in L(U_i)\}$$

in time $O_i(1) + t_0(|j\$| + n)$. **Remark.** This lemma is used the Seiferas-Fisher-Meyer proof of the hierarchy theorem for nondeterministic time. This lemma is analogous to the second recursion theorem (also called the fixed point theorem) in recursive function theory.[9]

[6.12] (Žák) If t is time-constructible and $t(n+1) = O(t(n))$ then for each $c > 0$, $NTIME(t) - NTIME_U^{cut}(ct) \neq \emptyset$. Hint: use Žák's theorem 13.

[6.13] (Book) Show the time analogue of the space translation lemma for tally languages: Let $f(n) \geq 2^n$, $X = D$ or N. Then

$$L \in XTIME(f(O(n))) \iff tally(L) \in XTIME(f(O(\log n))).$$

[9]The second recursion theorem is as follows: let U be a universal machine for $RE|\{0,1\}$. Then for every total recursive function f there exists an index i such that $U_i(x) = U_{f(i)}(x)$ for all x. As pointed out in the excellent exposition in [7, chapter 11], this theorem is the quintessence of diagonal arguments; it can be used (perhaps as an overkill) to give short proofs for many diagonalization results.

[6.14] (Wilson) Show that if every tally language in $NP \cap co\text{-}NP$ belongs to P then $NEXPT \cap co\text{-}NEXPT \subseteq DEXPT$. *Hint:* You may use the result of the previous exercise.

[6.15] (Book) Let $\Phi \subseteq \mathbf{DLOG}$ (the logarithmic space computable transformations), $X = D$ or N. Show that if L is $XTIME(O(n))$-complete under $\leq_m^{\Phi}$ then L is $XTIME(n^{O(1)})$-complete under $\leq_m^{\mathbf{DLOG}}$.

[6.16] Provide the proofs for the rest of the entries in the chart at the end of section 5.

[6.17] (Book) Show that if $NP \subseteq P$ then $NEXPT \subseteq DEXPT$.

[6.18] (Book) Show that $NP|\{1\} \neq DSPACE(\log^{O(1)} n))|\{1\}$.

[6.19] (Loui) Try to imitate the proof of theorem 20 to attempt to show that $P \neq PSPACE$. Where does the proof break down?

[6.20] (open) Separate $NTIME(2^{2^n})$ from $NTIME(n2^{2^n})$.

[6.21] (open) Improve the nondeterministic time hierarchy theorem of Seiferas-Fischer-Meyer by changing the theorem statement from '$o(t(n-1))$' to '$o(t(n))$'.

[6.22] Suppose that the outline (a) and (b) in section 5 were stated for the class $DSPACE(F)$ in place of K, where F is a set of complexity functions. State the conditions on F for the same conclusion to hold.

[6.23] (Stockmeyer) Say a language (Σ, L) is *naturally padded* if there is a symbol $\# \notin \Sigma$, some integer j_0, and a logspace transformation $t : \Sigma^*\#^* \to \Sigma^*$ such that (a) $L \cdot \#^* \leq_m^L L$ via t, and (b) $|t(x\#^j)| = |x| + j$ for all $x \in \Sigma^*$ and $j \geq j_0$. Prove theorem 22 using this " natural padding" instead of the "invariance under padding" assumption.

[6.24] (Stockmeyer) Suppose L is Karp-reducible to L' via a polynomial time transformation t that is linearly bounded. If L has an $t(n)$ (a.e.) lower bound on its acceptance time, show a corresponding a.e. lower bound on L'.

[6.25] Define an even stronger notion of separation: the characteristic classes $DTIME(t')$ and $DTIME(t)$ are said to be *very strongly separated* if there exists an infinite language L in $DTIME(t') - DTIME(t)$ such that for any acceptor M for L, $AcceptTime_{\mathrm{M}}(x) > t(|x|)$ for almost every input $x \in L(\mathrm{M})$. (In other words, we use 'a.e. x' rather than 'a.e. n'.) Extend the proof of theorem 21 to this setting of very strong separation.

[6.26] (Even, Long, Yacobi) Say L is *easier than* L' if for all N accepting L' there is an M accepting L such that

1. $AcceptTime_{\mathrm{M}}(x) = |x|^{O(1)} + AcceptTime_{\mathrm{N}}(x)$ for all input x.
2. For some $k > 0$, $(AcceptTime_{\mathrm{M}}(x))^k \leq AcceptTime_{\mathrm{N}}(x)$ for infinitely many x.

If only (1) is satisfied then we say that L is *not harder than* L'. Show that if there exists a language $L_0 \in NP - co\text{-}NP$ then:
(a) There exists a recursive language $L_1 \notin NP \cup co\text{-}NP$ such that L_0 is not harder than L_1 and L_1 is not harder than L_0.
(b) There exists a recursive language $L_2 \notin NP \cup co\text{-}NP$ such that L_1 is easier than L_0.

[6.27] (Ming Li) Show that for all $j > 1$, $NTIME(n^j) - D\text{-}TIME\text{-}SPACE(n^j, o(n))$ is non-empty.

[6.28] (Ming Li) Show that there is a language in $NTIME(n)$ that cannot be accepted by any deterministic simple Turing machine in time $O(n^{1.366})$.

Bibliography

[1] Manuel Blum. A machine-independent theory of the complexity of recursive functions. *Journal of the ACM*, 14(2):322–336, 1967.

[2] Ronald V. Book. Translational lemmas, polynomial time, and $(\log\ n)^j$-space. *Theoretical Computer Science*, 1:215–226, 1976.

[3] A. Borodin, R. Constable, and J. Hopcroft. Dense and non-dense families of complexity classes. *10th Proc. IEEE Symp. Found. Comput. Sci.*, pages 7–19, 1969.

[4] Alan Borodin. Computational complexity and the existence of complexity gaps. *Journal of the ACM*, 19(1):158–174, 1972.

[5] Jianer Chen and Chee-Keng Yap. Reversal complexity. *SIAM J. Computing*, to appear, 1991.

[6] Steven A. Cook. A hierarchy for nondeterministic time complexity. *Journal of Computers and Systems Science*, 7:343–353, 1973.

[7] Nigel J. Cutland. *Computability: an introduction to recursive function theory.* Cambridge University Press, 1980.

[8] S. Even, T. J. Long, and Y. Yacobi. A note on deterministic and nonderministic time complexity. *Information and Control*, 55:117–124, 1982.

[9] S. Even, A. L. Selman, and Y. Yacobi. Hard-core theorems for complexity classes. *Journal of the ACM*, 32(1):205–217, 1985.

[10] Martin Fürer. The tight deterministic time hierarchy. *14th Proc. ACM Symp. Theory of Comp. Sci.*, pages 8–16, 1982.

[11] J. Hartmanis, P. M. Lewis II, and R. E. Stearns. Hierarchies of memory limited computations. *IEEE Conf. Record on Switching Circuit Theory and Logical Design*, pages 179–190, 1965.

[12] J. Hartmanis and R. E. Stearns. On the computational complexity of algorithms. *Trans. Amer. Math. Soc.*, 117:285–306, 1965.

[13] Harry B. Hunt, III. On the time and tape complexity of languages, I. *5th Proc. ACM Symp. Theory of Comp. Sci.*, pages 10–19, 1973.

[14] O. Ibarra. A note concerning nondeterministic tape complexities. *J. ACM*, 19:608–612, 1972.

[15] Neil Immerman. Nondeterministic space is closed under complement. *Structure in Complexity Theory*, 3:112–115, 1988.

[16] Ming Li. Some separation results. Manuscript, 1985.

[17] Nancy A. Lynch. On reducibility to complex or sparse sets. *Journal of the ACM*, 22:341–345, 1975.

[18] M. Machtey and P. Young. *An Introduction to the General Theory of Algorithms*. Elsevier North Holland, New York, 1978.

[19] A. R. Meyer and E. M. McCreight. Computationally complex and pseudo-random zero-one valued functions. In Z. Kohavi and A. Paz, editors, *Theory of machines and computations*, pages 19–42. Academic Press, 1971.

[20] A. R. Meyer and L. J. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential space. *13th Proc. IEEE Symp. Found. Comput. Sci.*, pages 125–129, 1972.

[21] Albert R. Meyer. Weak monadic second order theory of successor is not elementary-recursive. In Dold and Eckmann (eds.), editors, *Logic Colloquium: Symposium on Logic Held at Boston University, 1972-73*, pages 132–154. Springer-Verlag, 1975.

[22] Wolfgang Paul. *Komplexitaetstheorie*. Teubner, Stuttgart, 1978.

[23] Michael Rabin. Degree of difficulty of computing a function. Technical Report Tech. Report 2, Hebrew Univ., 1960.

[24] C. W. Rackoff and J. I. Seiferas. Limitations on separating nondeterministic complexity classes. *SIAM J. Computing*, 10(4):742–745, 1981.

[25] S. Ruby and P. C. Fischer. Translational methods in computational complexity. *6th IEEE Conf. Record on Switching Circuit Thoery, and Logical Design*, pages 173–178, 1965.

[26] C. P. Schnorr and H. Klupp. A universally hard set of formulae with respect to non-deterministic Turing acceptors. *IPL*, 6(2):35–37, 1977.

[27] J. I. Seiferas, M. J. Fischer, and A. R. Meyer. Refinements of the nondeterministic time and space hierarchies. *14th Annual Symposium on Switching and Automata Theory*, pages 130–136, 1973.

[28] J. I. Seiferas, M. J. Fischer, and A. R. Meyer. Separating nondeterministic time complexity classes. *Journal of the ACM*, 25(1):146–167, 1978.

[29] L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time. *5th Proc. ACM Symp. Theory of Comp. Sci.*, pages 1–9, 1973.

[30] Larry J. Stockmeyer. The complexity of decision problems in automata theory and logic. Technical Report Project MAC Tech. Rep. TR-133, M.I.T., 1974. PhD Thesis.

[31] B. A. Trachtenbrot. On autoreducibility. *Soviet Math. Dokl.*, 11(3):814–817, 1970.

[32] Stanislav Žák. A Turing machine time hierarchy. *Theoretical Computer Science*, 26:327–333, 1983.

Contents

Chapter 7

Alternating Choices

March 8, 1999

7.1 Introduction to computing with choice

The choice-mode of computation comes in two main flavors. The first, already illustrated in Chapter 1 (section 6.2), is based on probability. The second is a generalization of nondeterministism called **alternation**. Let us briefly see what an alternating computation looks like. Let δ be the usual Turing transition table that has choice and let $C_0(w)$ denote the initial configuration of δ on input w. For this illustration, assume that every computation path is finite; in particular, this implies that no configuration is repeated in a computation path. The computation tree $T(w) = T_\delta(w)$ is defined in the obvious way: the nodes of $T(w)$ are configurations, with $C_0(w)$ as the root; if configuration C is a node of $T(w)$ and $C \vdash C'$ then C' is a child of C in $T(w)$. Thus the leaves of $T(w)$ are terminal configurations. The description of an alternating machine M amounts to specifying a transition table δ together with an assignment γ of a Boolean function $\gamma(q) \in \{\wedge, \vee\neg\}$ to each state q in δ. This induces a Boolean value on each node of $T(w)$ as follows: the leaves are assigned 1 or 0 depending on whether the configuration is accepting or not. If C is not a leaf, and q is the state in C, then we require that the number of children of C is equal to the arity of $\gamma(q)$. For instance, if C has two children whose assigned values are x and y then C is assigend the value $\gamma(q)(x, y)$. Finally we say M accepts w if the root $C_0(w)$ is assigned value 1.

The reader will see that nondeterministic computation corresponds to the case where $\gamma(q) = \vee$ for all q. Since the introduction of alternating machines by Chandra, Kozen and Stockmeyer[3] in 1978, the concept has proven to be an extremely useful tool in complexity theory.

The model of probabilistic machines we study was introduced by Gill[7]. Let us rephrase the description of probabilistic computation in chapter 1 in terms of assigning values to nodes of a computation tree. A probabilistic machine is formally

a transition table δ where each configuration spawns either zero or two children. For any input w, we again have the usual computation tree $T(w)$. The leaves of $T(w)$ are given a value of 0 or 1 as in the alternating case. However, an internal node u of $T(w)$ is assigned the average $(x+y)/2$ of the values x, y of the two children of u. The input w is accepted if the root is assigned a value greater than $1/2$. (The reader should be convinced that this description is equivalent to the one given in chapter 1.) The function $f(x,y) = (x+y)/2$ is called the *toss* function because in probabilistic computations, making choices is interpreted as branching according to the outcomes of tossing a fair coin.

Hence, we see that a common feature of probabilistic and alternating modes is that each mode amounts to a systematic bottom-up method of assigning values to nodes of computation trees. One difference is that, whereas probabilistic nodes are given (rational) values between 0 and 1, the alternating nodes are assigned Boolean values. We modify this view of alternating machines by regarding the Boolean values as the real numbers 0 and 1, and interpreting the Boolean functions $\wedge$, $\vee$ and $\neg$ as the real functions min, max and $f(x) = 1 - x$ (respectively).

With this shift of perspective, we have almost accomplished the transition to a new syncretistic model that we call *probabilistic-alternating machines*. This model was first studied in [23]. A probabilistic-alternating machine M is specified by giving a transition table δ and each state is associated with one of the four real functions

$$\min(x,y), \quad \max(x,y), \quad 1-x, \quad \frac{x+y}{2}. \tag{7.1}$$

A configuration in state q spawns either zero or m children where m is the arity of the function associated with q. Given an input w, we construct the tree $T(w)$ and assign values to its nodes in the usual bottom-up fashion (again, assuming $T(w)$ is a finite tree). We say M accepts the input w if the value at the root of $T(w)$ is $> 1/2$.

Probabilistic and alternating machines in the literature are often studied independently. In combining these two modes, we extend results known for only one of the modes, or unify distinct results for the separate modes. More importantly, it paves the way towards a general class of machines that we call *choice machines*. Computations by choice machines are characterized by the systematic assignment of 'values' to nodes of computation trees, relative to the functions $\gamma(q)$ which the machine associates to each state q. These functions are similar to those in (7.1), although an immediate question is what properties should these functions satisfy? This will be answered when the theory is developed. We call any assignment of 'values' to the nodes of a computation tree a *valuation*.[1] Intuitively, these values represent probabilities and lies in the unit interval $[0,1]$. But because of infinite computation trees, we are forced to take as 'values' any subinterval $[a,b]$ of the unit interval $[0,1]$. Such intervals represent uncertainty ranges in the probabilities.

[1]The term 'valuation' in algebra refers to a real function on a ring that satisfies certain axioms. Despite some concern, we will expropriate this terminology, seeing little danger of a context in which both senses of the term might be gainfully employed.

This leads to the use of a simple interval algebra. The present chapter develops the valuation mechanism needed for our theory of choice machines. We will extend probabilistic machines to a more general class called stochastic machines. This chapter focuses on alternation machines, leaving stochastic machines to the next chapter.

Other choice modes. Other authors independently proposed a variety of computational modes that turn out to be special cases of our probabilistic-alternating mode: *interactive proof systems* (Goldwasser, Micali and Rackoff [8]), *Arthur-Merlin games* (Babai [2]), *stochastic Turing machines* (Papadimitriou [16]), *probabilistic-nondeterministic machines* (Goldwasser and Sipser [9]). In general, communication protocols and game playing models can be translated as choice machines. In particular, this holds for the *probabilistic game automata* (Condon and Ladner [4]) which generalize interactive proof systems and stochastic machines[2]. Alternating machines are generalized to *logical type machines* (Hong [11]) where machine states can now be associated with any of the 16 Boolean functions on two variables. Some modes bearing little resemblance to choice machines can nevertheless be viewed as choice machines: for example, in chapter 9 we describe a choice mode that generalizes nondeterminism in a different direction than alternation. (This gives rise to the so-called *Boolean Hierarchy*.) These examples suggests that the theory of valuation gives a proper foundation for choice modes of computation. The literature can avoid our systematic development only by restrictions such as requiring constructible time-bounds.

7.2 Interval algebra

The above introduction to probabilistic-alternating machines explicitly avoided infinite computation trees $T(x)$. Infinite trees cannot be avoided in general; such is the case with space-bounded computations or with probabilistic choices. To see why infinite trees are problematic, recall that we want to systematically assign a value in $[0, 1]$ to each node of $T(x)$, in a bottom-up fashion. But if a node u of $T(x)$ lies on an infinite path, it is not obvious what value to assign to u.

Our solution [23] lies in assigning to u the smallest 'confidence' interval $I(u) \subseteq [0, 1]$ guaranteed to contain the 'true' value of u. This leads us to the following development of an **interval algebra**[3].

In the following, u, v, x, y, etc., denote real numbers in the unit interval [0, 1]. Let

$$INT := \{[u, v] : 0 \leq u \leq v \leq 1\}$$

[2]This game model incorporates 'partially-hidden information'. It will be clear that we could add partially-hidden information to choice machines too.

[3]*Interval arithmetic*, a subject in numerical analysis, is related to our algebra but serves a rather different purpose. We refer the reader to, for example, Moore [15].

denote the set of closed subintervals of $[0,1]$. An interval $[u,v]$ is *exact* if $u = v$, and we identify the exact interval $[u,u]$ with the real number u. We call u and v (respectively) the *upper* and *lower bounds* of the interval $[u,v]$. The unit interval $[0,1]$ is also called *bottom* and denoted $\bot$.

By an *interval function* we mean a function $f : INT^n \to INT$, where $n \geq 0$ denotes the arity of the function. We are interested in six interval functions. The first is the unary function of *negation* ($\neg$), defined as follows:

$$\neg[x,y] = [1-y, 1-x].$$

The remaining five are binary functions:

minimum ($\wedge$), *maximum* ($\vee$),
toss ($\fatsemi$),
probabilistic-and ($\otimes$), *probabilistic-or* ($\oplus$).

It is convenient to first define them as real functions. The real functions of minimum and maximum are obvious. The toss function is defined by

$$x \fatsemi y := \frac{x+y}{2}.$$

We saw in our introduction how this function arises from probabilistic (coin-tossing) algorithms. The last two functions are defined as follows:

$$\begin{aligned} x \otimes y &:= xy \\ x \oplus y &:= x + y - xy \end{aligned}$$

Thus $\otimes$ is ordinary multiplication of numbers but we give it a new name to signify the interpretation of the numbers as probabilities. If E is the event that *both* E_1 *and* E_2 occur, then the probability $\Pr(E)$ of E occurring is given by

$$\Pr(E) = \Pr(E_1) \otimes \Pr(E_2).$$

We assume that E_1, E_2 are independent events. Similarly $\oplus$ has a probabilistic interpretation: if E is the event that *either* E_1 *or* E_2 occurs, then

$$\Pr(E) = \Pr(E_1) \oplus \Pr(E_2).$$

To see this, simply note that $x \oplus y$ can also be expressed as $1 - (1-x)(1-y)$. For brevity, we suggest reading $\otimes$ and $\oplus$ as 'prand' and 'pror', respectively.

We note that these 5 real functions can also be regarded as functions on $[0,1]$ (*i.e.*, if their arguments are in $[0,1]$ then their values remain in $[0,1]$). We may then extend them to the subintervals INT of the unit interval as follows. If $\circ$ is any of these 5 functions, then we define

$$[x,y] \circ [u,v] := [(x \circ u), (y \circ v)].$$

For instance, $[x, y] \otimes [u, v] = [xu, yv]$ and $[x, y] \wedge [u, v] = [\min(x, u), \min(y, v)]$.

Alternatively, for any continuous function $f : [0, 1] \to [0, 1]$, we extend the range and domain of f from $[0, 1]$ to INT by the definition $f(I) = \{f(x) : x \in I\}$. If f is also monotonic, this is equivalent to the above.

One easily verifies:

Lemma 1 *All five binary functions are commutative. With the exception of $\oplus$, they are also associative.*

The set INT forms a lattice with $\wedge$ and $\vee$ as the join and meet functions, respectively[4]. It is well-known that we can define a partial order $\leq$ in any lattice by:

$$[x, y] \leq [u, v] \iff ([x, y] \wedge [u, v]) = [x, y]. \tag{7.2}$$

Note that (7.2) is equivalent to:

$$[x, y] \leq [u, v] \iff x \leq u \text{ and } y \leq v.$$

When we restrict this partial ordering to exact intervals, we get the usual ordering of real numbers. For reference, we will call $\leq$ the *lattice-theoretic ordering* on INT.

The negation function is not a complementation function (in the sense of Boolean algebra [5]) since neither $I \wedge \neg I = 0$ nor[5] $I \vee \neg I = 1$ holds for all $I \in INT$. However it is idempotent, $\neg\neg I = I$. Probabilistic-and and probabilistic-or can be recovered from each other in the presence of negation. For example,

$$I \otimes J = \neg(\neg I \oplus \neg J).$$

It easy to verify the following forms of de Morgan's law:

Lemma 2

$$\begin{aligned}
\neg(I \wedge J) &= \neg I \vee \neg J \\
\neg(I \vee J) &= \neg I \wedge \neg J \\
\neg(I \oplus J) &= \neg I \oplus \neg J \\
\neg(I \otimes J) &= \neg I \otimes \neg J \\
\neg(I \otimes J) &= \neg I \oplus \neg J
\end{aligned}$$

where $I, J \in INT$.

[4] A lattice X has two binary functions, join and meet, satisfying certain axioms (essentially all the properties we expect from max and min). Lattice-theoretic notations can be found, for instance, in [5]. The lattice-theoretic properties are not essential for the development of our results.

[5] We assume that $\neg$ has higher precedence than the binary operators so we may omit parenthesis when convenient.

In view of these laws, we say that the functions $\wedge$ and $\vee$ are *duals of each other* (with respect to negation); similarly for the pair $\otimes$ and $\oplus$. However, $\owedge$ is self-dual.

We verify the distributivity of $\wedge$ and $\vee$ with respect to each other:

$$I \vee (J_1 \wedge J_2) = (I \vee J_1) \wedge (I \vee J_2)$$
$$I \wedge (J_1 \vee J_2) = (I \wedge J_1) \vee (I \wedge J_2).$$

Furthermore, $\owedge$, $\otimes$ and $\oplus$ each distributes over both $\wedge$ and $\vee$:

$$I \owedge (J_1 \wedge J_2) = (I \owedge J_1) \wedge (I \owedge J_2), \quad I \owedge (J_1 \vee J_2) = (I \owedge J_1) \vee (I \owedge J_2)$$
$$I \otimes (J_1 \wedge J_2) = (I \otimes J_1) \wedge (I \otimes J_2), \quad I \otimes (J_1 \vee J_2) = (I \otimes J_1) \vee (I \otimes J_2)$$
$$I \oplus (J_1 \wedge J_2) = (I \oplus J_1) \wedge (I \oplus J_2), \quad I \oplus (J_1 \vee J_2) = (I \oplus J_1) \vee (I \oplus J_2)$$

However $\otimes$ and $\oplus$ do not distribute with respect to each other (we only have $x \otimes (y \oplus z) \leq (x \otimes y) \oplus (x \otimes z)$). And neither $\wedge$ nor $\vee$ distributes over $\owedge$, $\otimes$ or $\oplus$.

Another Partial Order. For our applications, it turns out that a more useful partial order on INT is $\sqsubseteq$, defined by:

$$[x, y] \sqsubseteq [u, v] \iff x \leq u \text{ and } v \leq y.$$

Clearly $\sqsubseteq$ is the reverse of the set inclusion relation between intervals: $I \sqsubseteq J \iff J \supseteq I$ as sets. With respect to the $\sqsubseteq$-ordering, all exact intervals are maximal and pairwise incomparable[6]. In view of our interpretation of intervals as 'intervals of confidence', if $I \sqsubseteq J$ then we say J has 'at least as much information' as I. For this reason, we call $\sqsubseteq$ the *information-ordering*. In contrast to the lattice-theoretic $\leq$-ordering, $\sqsubseteq$ only gives rise to a lower semi-lattice with the meet function $\sqcap$ defined by

$$[x, y] \sqcap [u, v] = [\min(x, u), \max(y, v)].$$

(The following suggestion for defining the join $\sqcup$ fails: $[x, y] \sqcup [u, v] = [\max(x, u), \min(y, v)]$.) Note that bottom $\bot$ is the least element ("no information") in the information-ordering.

Example 1 The strong 3-valued algebra described[7] by Chandra, Kozen and Stockmeyer [3] is a subalgebra of our interval algebra, obtained by restricting values to $\{0, 1, \bot\}$. See figure 7.1 for its operation tables. They only were interested in the functions $\wedge$, $\vee$, $\neg$. Thus our interval algebra gives a model (interpretation) for this 3-valued algebra.

The contrast between $\leq$ and $\sqsubseteq$ is best exemplified by the respective partial orders restricted to this 3-valued algebra, put graphically:

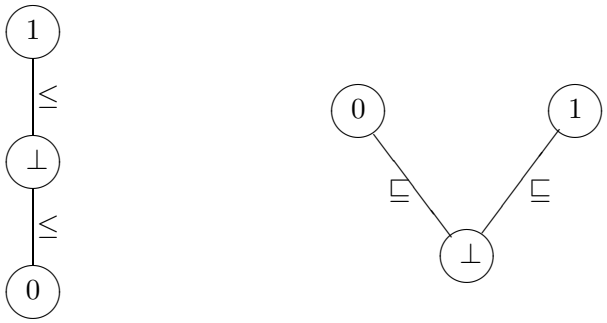
[6] I and J are $\sqsubseteq$-*comparable* if $I \sqsubseteq J$ or $J \sqsubseteq I$, otherwise they are $\sqsubseteq$-*incomparable*.

[7] Attributed to Kleene.

$\wedge$	0	1	$\perp$
0	0	0	0
1	0	1	$\perp$
$\perp$	0	$\perp$	$\perp$

$\vee$	0	1	$\perp$
0	0	1	$\perp$
1	1	1	1
$\perp$	$\perp$	1	$\perp$

Figure 7.1: The strong 3-valued algebra.



■

This information ordering gives rise to some important properties of interval functions:

Definition 1
(i) An n-ary function

$$f : INT^n \to INT$$

is monotonic *if for all intervals* $J_1, \ldots, J_n, J'_1, \ldots, J'_n$*:*

$$J_1 \sqsubseteq J'_1, \ldots, J_n \sqsubseteq J'_n \quad \Rightarrow \quad f(J_1, \ldots, J_n) \sqsubseteq f(J'_1, \ldots, J'_n).$$

(ii) f *is* continuous *if it is monotonic and for all non-decreasing sequences*

$$I_i^{(1)} \sqsubseteq I_i^{(2)} \sqsubseteq I_i^{(3)} \sqsubseteq \cdots$$

$(i = 1, \ldots, n)$*, we have that*

$$f(\lim_j\{I_1^{(j)}\}, \ldots, \lim_j\{I_n^{(j)}\}) = \lim_j f(I_1^{(j)}, \ldots, I_n^{(j)}). \tag{7.3}$$

■

Note that a continuous function is assumed monotonic. This ensures that the definition above is well-defined. More precisely, taking limit on the right-hand side of (7.3) is meaningful because monotonicity of f implies

$$f(I_1^{(1)}, \ldots, I_n^{(1)}) \sqsubseteq f(I_1^{(2)}, \ldots, I_n^{(2)}) \sqsubseteq f(I_1^{(3)}, \ldots, I_n^{(3)}) \sqsubseteq \cdots.$$

Lemma 3 *The six functions $\wedge$, $\vee$, $\overline{\wedge}$, $\otimes$, $\oplus$ and $\neg$ are continuous.*

We leave the proof as an exercise. Continuity of these functions comes from continuity of their real counterpart.

Example 2 The *cut-off function* $\delta_{\frac{1}{2}}(x)$ is defined to be 1 if $x > \frac{1}{2}$ and 0 otherwise. We extend this function to intervals in the natural way: $\delta_{\frac{1}{2}}([u, v]) = [\delta_{\frac{1}{2}}(u), \delta_{\frac{1}{2}}(v)]$. This function is monotonic but not continuous. ■

The following simple observation is useful for obtaining monotonic and continuous functions.

Lemma 4
(i) Any composition of monotonic functions is monotonic.
(ii) Any composition of continuous functions is continuous.

7.3 Theory of Valuations

To give a precise definition for the choice mode of computation, and its associated complexity measures, we introduced the theory of valuations. We first signal a change in emphasis as we go into choice modes:

- Until now, 'rejection' is usually equated with 'non-acceptance'. Henceforth we subdivide non-acceptance into two possibilities: rejection and indecision.

- We refine our terminology to distinguish between the *global decision* versus *local answers* of a Turing acceptor. On a given input, there is one global decision whether to accept, reject or to remain undecided. This decision is based the the entire computation tree. However, each terminal configuration gives a local answer, based on the path leading to that configuration. We retain the terminology of accept/reject/undecided for the global decision but to avoid confusion, introduce a different terminology to discuss local answers.

Local Answers
We replace the accept and reject states (q_a and q_r) by the new distinguished states,

$$q_Y, q_N \in Q_\infty.$$

We call them the *YES-state* and *NO-state*, respectively, and arrange our transition tables so configurations with these states are necessarily terminal. A terminal configuration C is called a **YES-configuration**, **NO-configuration** or a **YO-configuration**, depending on whether its state is q_Y, q_N or some other state. A complete computation path is called a **YES-path**, **NO-path** or **YO-path**, depending on whether it terminates in a YES, NO or otherwise. Thus a YO-path either terminates in a YO-configuration or is non-terminating. We also speak of a machine or configuration giving a "YES, NO or YO-answer": such locution would be self-evident.

The global decision is determined by the acceptance rules, to be defined below. Naturally, the global decision is some generalized average of the local answers. This global/local terminology anticipates the quantitative study of errors in a computation (see chapter 8). For now, it suffices to say that all error concepts are based on the discrepancies between global decisions and local answers. The seemingly innocuous introduction of indecision and YO-answers[8] is actually critical in our treatment of errors.

Let f be a function on INT, i.e., there is an $n \geq 0$ (n is the arity of f) such that

$$f : INT^n \to INT.$$

The 0-ary functions are necessarily *constant functions*, and the *identity function* $\iota(I) = I$ has arity 1.

Definition 2 *A set B of functions on INT is called a* basis set *if the functions in B are continuous and if B contains the identity function ι and the three* 0*-ary functions* $0, 1$ *and* $\perp$. ■

Definition 3 *Let B be a basis set. A B-acceptor is a pair* $\mathrm{M} = (\delta, \gamma)$ *where δ is a Turing transition table whose states are ordered by some total ordering* $\overset{\mathrm{M}}{<}$*, and γ associates a basis function $\gamma(q)$ to each state q,*

$$\gamma : Q \to B.$$

Moreover, δ has the property that if C is a configuration of δ in state q and $\gamma(q) = f$ has arity n, then C either is a terminal configuration or has exactly n immediate successors $C_1, \ldots, C_n$ such that the C_i's have distinct states. ■

[8] We owe the YO-terminology to the unknown street comedian in Washington Square Park who suggested that in a certain uptown neighborhood of Manhattan, "we say YO to drugs". Needless to say, this local answer is in grave error.

Name	Basis B	Mode Symbol
deterministic	$\emptyset$	D
nondeterministic	$\{\vee\}$	N
probabilistic	$\{\oplus\!\!\!\!-\}$	Pr
alternating	$\{\wedge, \vee, \neg\}$	A
interactive proofs	$\{\oplus\!\!\!\!-, \vee\}$	Ip
probabilistic-alternating	$\{\oplus\!\!\!\!-, \wedge, \vee, \neg\}$	PrA
stochastic	$\{\oplus\!\!\!\!-, \otimes, \oplus, \neg\}$	St
stochastic-alternating	$\{\oplus\!\!\!\!-, \otimes, \oplus, \wedge, \vee, \neg\}$	StA

Figure 7.2: Some Choice Modes and their Symbols.

We simply call M a *choice acceptor (or machine)* if B is understood.

Explanation. We describe how choice machines operate. If the immediate successors of a configuration C are $C_1, \ldots, C_n$ such that the state of C_i is less than the state of C_{i+1} (under the ordering $\overset{\text{M}}{<}$) for each $i = 1, \ldots, n-1$, then we indicate this by writing[9]

$$C \vdash (C_1, \ldots, C_n).$$

If q is the state of C, we also write $\gamma(C)$ or γ_C instead of $\gamma(q)$. We require that the $C_1, \ldots, C_n$ to have distinct states because the value of the node (labeled by) C in the computation tree is given by $\gamma_C(v_1, \ldots, v_n)$ where v_i is the value of the node (labeled by) C_i. Without an ordering such as $\overset{\text{M}}{<}$ on the children of C, we have no definite way to assign the v_i's as arguments to the function γ_C. But for basis sets that we study, the functions are symmetric in their arguments and so we will not bother to mention the ordering $\overset{\text{M}}{<}$. ■

It is useful to collect some notations for B-choice machines for some important bases B. Since every basis set contains the functions $\iota, 0, 1, \perp$, we may omit them when writing out B.

The mode symbol extends our previous notations for non-deterministic or deterministic classes. For instance, the notation $IpTIME(f(n))$ clearly refers to the class of languages accepted by "interactive proof machines" in time $f(n)$. Of course, the precise definition of time or space complexity for such computations is yet to be rigorously defined.

We shall say a B-machine makes *B-choices*. Thus, nondeterministic machines makes nondeterministic choices and alternating machines makes alternating choices. MIN- and MAX-choices are also called **universal choices** and **existential choices**;

[9]This notation could cause some confusion because we do not want to abandon the original meaning of "$C \vdash C'$", that C' is a successor of C. Hence "$C \vdash C'$" does not mean that C has only one successor; to indicate this, we need to write "$C \vdash (C')$".

Coin-tossing choices are also called random choices or probabilistic choices.

From the table, it is evident that we differentiate between the words 'probabilistic' and 'stochastic': the adjective 'probabilistic' applies only to coin-tossing concepts – a usage that conforms to the literature. The adjective 'stochastic' is more general and includes coin-tossing concepts.

We abbreviate a probabilistic-alternating machine to 'PAM', and a stochastic-alternating machine to 'SAM'.

If $\gamma(q) = \wedge$ (respectively, $\vee, \dot{\vee}, \otimes, \oplus, \neg$) then q is called an *MIN-state* (respectively, *MAX-, TOSS-, PrAND-, PrOR-, NOT-state*). If the state of C is an MIN-state (MAX-state, etc.), then C is an *MIN-configuration* (*MAX-configuration*, etc.).

Example 3 In chapter 2 we showed that a deterministic 1-tape Turing machine can accept the palindrome language

$$L_{pal} = \{w : w \in \{0,1\}^*, w = w^R\}$$

in (simultaneous) linear time and linear space or in (simultaneous) quadratic time and logarithmic space. Furthermore, if L_{pal} is accepted nondeterministically in time t and space s then $s(n) \cdot t(n) = \Omega(n^2)$. Now we show that an alternating machine M can accept L in linear time but using only logarithmic space. Hence the alternating mode is strictly more powerful than the fundamental mode. Of course, the formal definition of what it means for a choice machine to accept a word, and the notion of space and time complexity is yet to come. But if our machine halts on all paths, then these concepts are intuitively obvious: we rely on such intuitions for the reader to understand the example. The idea of the construction is that M accepts an input w provided for all $i = 1, \ldots, n$, $w[i] = w[n+1-i]$, provided $|w| = n$.

In phase 1, the machine M on input w of length n marks out some $m \geq 0$ cells using existential choice. It is not hard to show that the simple procedure P that repeatedly increments a binary counter from 0 to 2^m, taking $O(2^m)$ steps overall. In phase 2, M deterministically checks if $2^{m-1} < n \leq 2^m$ using this procedure P, answering NO otherwise. Hence phases 1 and 2 take linear time. In phase 3, M universally guesses a bit for each of the marked cells. This takes $O(\log n)$ steps. At the end of phase 3, M is armed with a binary number i between 0 and 2^{m+1}. Then it deterministically tests if $w[i] = w[n-i]$. If $i > n$ then this test, by definition, passes. In any case this test takes a linear number of steps, again using procedure P. This completes our description of M. It is clear that M accepts L_{pal} and uses $O(\log n)$ space. ■

We now want to define acceptance by choice machines. Basically we need to assign intervals $[u, v] \in INT$ to nodes in computation trees. The technical tool we employ is the concept of a 'valuation'.

Definition 4 *Let M$= (\delta, \gamma)$ be a choice machine. The set of configurations of δ is denoted Δ(M). A valuation of M is a function*

$$V : \Delta(\mathrm{M}) \to INT.$$

A partial ordering on valuations is induced from the $\sqsubseteq$ -ordering on INT as follows: for valuations V_1 and V_2, define $V_1 \sqsubseteq V_2$ if

$$V_1(C) \sqsubseteq V_2(C)$$

for all $C \in \Delta(\mathrm{M})$. The bottom valuation*, denoted $V_\perp$, is the valuation that always yield $\perp$. Clearly $V_\perp \sqsubseteq V$ for any valuation V.* ■

Definition 5 *Let $\Delta \subseteq \Delta(\mathrm{M})$. We define the following operator τ_Δ on valuations. If V is a valuation, then $\tau_\Delta(V)$ is the valuation V' defined by:*

$$V'(C) = \begin{cases} \perp & \text{if} \quad C \notin \Delta \text{ or } C \text{ is YO-configuration,} \\ 1 & \text{else if} \quad C \text{ is a YES-configuration,} \\ 0 & \text{else if} \quad C \text{ is a NO-configuration,} \\ \gamma_C(V(C_1), \ldots, V(C_n)) & \text{else if} \quad C \vdash (C_1, \ldots, C_n). \end{cases}$$

■

For instance, we may choose Δ to be the set of all configurations of M that uses at most space h (for some h).

Lemma 5 (Monotonicity) *$\Delta_1 \subseteq \Delta_2$ and $V_1 \sqsubseteq V_2$ implies $\tau_{\Delta_1}(V_1) \sqsubseteq \tau_{\Delta_2}(V_2)$.*

Proof. We must show $\tau_{\Delta_1}(V_1)(C) \sqsubseteq \tau_{\Delta_2}(V_2)(C)$ for all $C \in \Delta(\mathrm{M})$. If $C \notin \Delta_1$, then this is true since the left-hand side is equal to $\perp$. So assume $C \in \Delta_1$. If C is terminal, then $\tau_{\Delta_1}(V_1)(C) = \tau_{\Delta_2}(V_2)(C)$ $(= 0, 1 \text{ or } \perp)$. Otherwise, $C \vdash (C_1, \ldots, C_n)$ where n is the arity of γ_C. Then

$$\begin{aligned} \tau_{\Delta_1}(V_1)(C) &= \gamma_C(V_1(C_1), \ldots, V_1(C_n)) \\ &\sqsubseteq \gamma_C(V_2(C_1), \ldots, V_2(C_n)) \\ &= \tau_{\Delta_2}(V_2)(C). \end{aligned}$$

where the $\sqsubseteq$ follows from the monotonicity of γ_C. **Q.E.D.**

For any $\Delta \subseteq \Delta(\mathrm{M})$ and $i \geq 0$, let τ_Δ^i denote operator obtained by the i-fold application of τ_Δ, i.e.,

$$\tau_\Delta^0(V) = V, \quad \tau_\Delta^{i+1}(V) = \tau_\Delta(\tau_\Delta^i(V)).$$

As corollary, we get

$$\tau_\Delta^i(V_\perp) \sqsubseteq \tau_\Delta^{i+1}(V_\perp)$$

for all $i \geq 0$. To see this, use induction on i and the monotonicity lemma.

Definition 6 *From the compactness of the interval $[0, 1]$, we see that there exists a unique least upper bound Val_Δ defined by*

$$Val_\Delta(C) = \lim\{\tau_\Delta^i(V_\perp)(C) : i \geq 0\},$$

for all $C \in \Delta$. If $\Delta = \Delta(\mathrm{M})$, then we denote the operator τ_Δ by τ_M, and the valuation Val_Δ by Val_M. ■

A simple consequence of the monotonicity lemma is the following:

$$\Delta_1 \subseteq \Delta_2 \quad \Rightarrow \quad Val_{\Delta_1} \sqsubseteq Val_{\Delta_2}.$$

To see this, it is enough to note that for all $i \geq 0$, $\tau^i_{\Delta_1}(V_\perp) \sqsubseteq \tau^i_{\Delta_2}(V_\perp)$.

For any operator τ and valuation V, we say V is a *fixed point* of τ if $\tau(V) = V$.

Lemma 6 *Val_Δ is the least fixed point of τ_Δ, i.e.,*

(i) *It is a fixed point: $\tau_\Delta(Val_\Delta) = Val_\Delta$*

(ii) *It is the least such: for all valuations V, if $\tau_\Delta(V) = V$ then $Val_\Delta \sqsubseteq V$.*

Proof.

(i) If C is terminal then it is easy to see that $\tau_\Delta(Val_\Delta)(C) = Val_\Delta(C)$. For non-terminal C, if $C \vdash (C_1, \ldots, C_n)$ then

$$\begin{aligned}
\tau_\Delta(Val_\Delta)(C) &= \gamma_C(Val_\Delta(C_1), \ldots, Val_\Delta(C_n)) \\
&= \gamma_C(\lim_i\{\tau^i_\Delta(V_\perp)(C_1)\}, \ldots, \lim_i\{\tau^i_\Delta(V_\perp)(C_n)\}) \\
&= \lim_i\{\gamma_C(\tau^i_\Delta(V_\perp)(C_1), \ldots, \tau^i_\Delta(V_\perp)(C_n))\} \quad \text{(by continuity)} \\
&= \lim_i\{\tau^{i+1}_\Delta(V_\perp)(C)\} \\
&= Val_\Delta(C).
\end{aligned}$$

(ii) $V_\perp \sqsubseteq V$, so $\tau^i_\Delta(V_\perp) \sqsubseteq \tau^i_\Delta(V) = V$ for all $i \geq 0$. Hence $Val_\Delta \sqsubseteq V$.

Q.E.D.

Example 4 To see that a fixed point of τ_Δ need not be unique, consider a binary computation tree in which all paths, with a single exception, terminate at accepting configurations. The exception is the infinite path π that always branches to the right. We could make sure that each node in this tree has a distinct configuration. Assuming that all nodes are MIN-configurations, a fixed point valuation V_1 of the computation tree is where all nodes have value 1. Another fixed point valuation V_2 assigns each nodes in π to 0 but the rest has value 1. But the least fixed point valuation V_0 assigns to the value $\perp$ to each node on the path π and the value 1 to the rest. ■

Definition 7 *An interval $I \subseteq [0,1]$ is a* accepting *if it is contained in the half-open interval $(\frac{1}{2}, 1]$,* i.e., *$I \subseteq (\frac{1}{2}, 1]$. It is* rejecting *if $I \subseteq [0, \frac{1}{2})$; it is* undecided *if it is neither accepting nor rejecting.* ■

Note that I is accepting/rejecting iff each $v \in I$ is greater/less than $\frac{1}{2}$. Similarly I is undecided iff $\frac{1}{2} \in I$.

Definition 8 (Acceptance rule for choice machines)
(i) Let w be a word in the input alphabet of a choice acceptor M*, and Δ a set of configurations of* M*. The* Δ-value of w*, denoted $Val_\Delta(w)$, refers to $Val_\Delta(C_0(w))$ where $C_0(w)$ is the initial configuration of* M *on w. If $\Delta = \Delta(\mathrm{M})$, the set of all configurations of* M*, we write $Val_{\mathrm{M}}(w)$ instead of $Val_{\Delta(\mathrm{M})}(w)$.*
(ii) We say M accepts, rejects *or* is undecided on w *according as $Val_{\mathrm{M}}(w)$ is accepting, rejecting or undecided.*
(iii) A machine is said to be **decisive** *if every word is either accepted or rejected; otherwise it is* **indecisive**
(iv) The language accepted by M *is denoted $L(\mathrm{M})$. The language rejected by* M *is denote $\overline{L}(\mathrm{M})$. Thus,* M *is decisive iff*

$$\overline{L}(\mathrm{M}) = co\text{-} - L(M).$$

■

Convention. In the course of this section, we will introduce other types of fixed point valuations. It is helpful to realize that we will use '*Val*' (with various subscripts) only to denote valuations that are least fixed points of the appropriate operators.

7.3.1 Tree Valuations and Complexity

To discuss complexity in general, we need an alternative approach to valuations, called 'tree valuations'. To emphasize the difference, the previous notion is also called 'configuration valuations'.

> Configuration valuations allows us to define the notion of acceptance or rejection. They can also define space complexity: thus, we say that M accepts input w in space h if $Val_\Delta(w)$ is accepting with Δ comprising all those configurations of M that uses space $\leq h$, Unfortunately, configuration valuations are not suited for time complexity. To see why, note that they are unable to distinguish between different occurrences of the same configuration C in computation trees. That is, once we fix Δ, then the valuation of C is uniquely determined. But suppose C occurs in two positions (say, C_1 and C_2) of a computation tree. Assume the depth of C_1 is less than the depth of C_2. In a time-limited computation, we are interested in computation trees with bounded depths. We want "valuation" V of such trees which may distinguish between C_1 and C_2: thus, $V(C_1)$ may have more information than $V(C_2)$, since it is allowed a longer computation time in its subtree.

The following treatment is abbreviated since it imitates the preceding development.

Definition 9 *Fix a choice machine* M *and let* w *any input.*
(i) The complete computation tree $T_{\rm M}(w)$ *of* M *on* w *is an ordered tree whose nodes are labeled by configurations from* $\Delta_{\rm M}$ *such that the root is labeled with the initial configuration* $C_0(w)$*, and whenever a node* u *is labeled by some* C *and* $C \vdash (C_1, \ldots, C_n)$ *then* u *has* n *children* $u_1, \ldots, u_n$ *which are ordered so that* u_i *is labeled by* C_i*. We write* $u \vdash (u_1, \ldots, u_n)$ *in this case. By abuse of terminology, we sometimes identify a node* u *with its label* C*.*
(ii) A tree T' *is a prefix of another tree* T *if* T' *is obtained from* T *by pruning*[10] *some subset of nodes of* T*. In particular, if* T' *is non-empty then the root of* T' *is the root of* T*. If* T *is labeled, then* T' *has the induced labeling.*
(iii) A computation tree T *of* w *is a prefix of the complete computation tree* $T_{\rm M}(w)$*. We call* $T_{\rm M}(w)$ *the completion of* T*.*
(iv) A (tree) valuation *on a computation tree* T *is a function* V *that assigns a value* $V(u) \in INT$ *for each node* u *in the completion* $T_{\rm M}(w)$ *of* T*, with the property that nodes not in* T *are assigned* $\perp$*. We also call* V a tree valuation of w*. If* V, V' *are valuations of* w *then we define*

$$V \sqsubseteq V'$$

if $V(u) \sqsubseteq V'(u)$ *for all* $u \in T_{\rm M}(w)$*.*
(v) The bottom valuation*, denoted* $V_\perp$*, assigns each node of* $T_{\rm M}(w)$ *to* $\perp$*. Clearly* $V_\perp$ *is the* $\sqsubseteq$*-minimum tree valuation, for any given* w*.*
(vi) The operator τ_T *transforms a valuation* V *on* T *to a new valuation* $\tau_T(V)$ *on* T *as follows: for each node* $u \in T_{\rm M}(w)$*,*

$$\tau_T(V)(u) = \begin{cases} \perp & \text{if } u \text{ is a leaf of } T \text{ or } u \notin T, \\ 1 & \text{else if } u \text{ is a YES-node,} \\ 0 & \text{else if } u \text{ is a NO-node,} \\ \gamma_u(V(u_1), \ldots, V(u_n)) & \text{else if } u \vdash (u_1, \ldots, u_n). \end{cases}$$

Let the least fixed point of τ_T *be denoted by* Val_T*. In fact,* $\tau_T^i(V_\perp) \sqsubseteq \tau_T^{i+1}(V_\perp)$ *for all* $i \geq 0$ *and we have*

$$Val_T = \lim\{\tau_T^i(V_\perp) : i \geq 0\}.$$

(vii) An accepting/rejecting/undecided tree for w *is is any computation tree* T *of* w *such that* $Val_T(u_0)$ *is accepting/rejecting/undecided where* u_0 *is the root of* T*.* ■

We claim that Val_T is the least fixed point without proof because it is proved exactly as for configuration valuations; furthermore, the next section gives another approach.

Let us say a word w is accepted or rejected by M in the 'new sense' if there is an accepting/rejecting tree for w. We next show the new sense is the same as the

[10] To prune a node u from T means to remove from T the node u and all the descendents of u. Thus, if we prune the root of T, we an empty tree.

old. First we introduce a notation: if $\Delta \subseteq \Delta(\mathrm{M})$ then let

$$T_\Delta(w)$$

denote the largest computation tree T of w all of whose nodes are labeled by elements of Δ. It is not hard to see that this tree is uniquely defined, and is non-empty if and only if the initial configuration $C_0(w)$ is in Δ. Equivalence of the two senses of acceptance amounts to the following.

Lemma 7 *Fix* M *and input* w.
(a) If T *is an accepting/rejecting computation tree of* w *then* $Val_\Delta(w)$ *is also accepting/rejecting, where* Δ *is the set of labels in* T.
(b) Conversely, for any $\Delta \subseteq \Delta(\mathrm{M})$, *if* $Val_\Delta(w)$ *is accepting/rejecting then* $T_\Delta(w)$ *is also accepting/rejecting.*

Its proof is left as an exercise.

Definition 10 *(Acceptance Complexity) Let* r *be any extended real number.*
(i) We say that M *accepts* x *in time* r *if there is an accepting tree* T *on input* x *whose nodes are at level at most* r *(the root is level* 0*).*
(ii) We say M *accepts* x *in space* r *if there is an accepting tree* T *on input* x *whose nodes each uses space at most* r.
(iii) We say M *accepts* x *in reversal* r *if there is an accepting tree* T *on input* x *such that each path in the tree makes at most* r *reversals.*
(iv) A computation path $C_1 \vdash C_2 \vdash \cdots \vdash C_m$ *makes* (at least) r alternations *if there are* $k = \lfloor r \rfloor + 1$ *configurations*

$$C_{i_0}, C_{i_1}, \ldots, C_{i_k}$$

$1 \leq i_1 < i_2 < \cdots < i_k \leq m$ *such that each* C_{i_j} $(j = 0, \ldots, k)$ *is either a MIN- or a MAX-configuration. Furthermore,* $C_{i_{j-1}}$ *and* C_{i_j} $(j = 1, \ldots, k)$ *make different choices (one makes a MIN- and the other a MAX-choice) if and only if there are no NOT-configurations between them along the path.* M *accepts* x *in* r *alternations if there is an accepting tree* T *on input* x *such that no path in the tree makes* $1 + r$ *alternations.*
(v) Let r_1, r_2, r_3 *be extended real numbers. We say* M *accepts* x *in simultaneous time-space-reversal* (r_1, r_2, r_3) *if there is an accepting tree* T *that satisfies the requirements associated with each of the bounds* r_i $(i = 1, \ldots, 3)$ *for the respective resources.*
(vi) For complexity functions f_1, f_2, f_3, *we say that* M *accepts in simultaneous time-space-reversal* (f_1, f_2, f_3) *if for each* $x \in L(\mathrm{M})$, M *accepts* x *in simultaneous time-space-reversal* $(f_1(|x|), f_2(|x|), f_3(|x|))$. *This definition extends to other simultaneous bounds.* ■

We have just introduced a new resource 'alternation'. Unlike time, space and reversals, this resource is mode-dependent. For example, the machine in the palindrome example above has one alternation and nondeterministic machines has no alternations. We have a similar monotonicity property for tree valuations: if T is a prefix of T' then

$$Val_T \sqsubseteq Val_{T'}.$$

In consequence, we have:

Corollary 8 *If* M *accepts an input in time r then it accepts the same input in time r' for any $r' > r$. Similarly for the other resources.*

Our definition of accepting in time r is phrased so that the accepting tree T need not include all nodes at levels up to $\lfloor r \rfloor$. Because of monotonicity, it may be more convenient to include all nodes up to level $\lfloor r \rfloor$. But when other resource bounds are also being considered, we may no longer be free to do this.

The following result is fundamental:

Theorem 9 (Compactness) *If a choice machine* M *accepts a word x then it has a finite accepting tree on input x. Similarly, if* M *rejects a word, then there is a finite rejecting tree.*

The proof will be deferred to the next section. Thus, if an input is accepted, then it is accepted in finite amounts of time, space, etc. This result implies that the complexity measures such as time, space, reversals or alternation are Blum measures (chapter 6, section 8).

On rejection and running complexity. The above definitions of complexity is concerned with accepted inputs only, and no assumptions on the computation of M are made if $w \notin L(\mathrm{M})$. In other words, we have been discussing acceptance complexity. We now introduce running complexity whose general idea is that complexity bounds apply to rejected as well as accepted words. Should running complexity allow indecision on an input? Since indecision can always be achieved with the empty computation tree, we insist that there be no indecision in running complexity.

Definition 11 *(Running time complexity) Fix a choice machine* M.
(i) We say M rejects *an input w in k steps if there is a rejecting tree of* M *on w whose nodes have level at most k.*
(ii) For any complexity function $t(n)$, we say M rejects in time $t(n)$ *if for all rejected inputs w, M rejects w in time $t(|w|)$.*
(iii) M runs in time (t, t') *if each input of length n is either accepted in time $t(n)$ or rejected in time $t'(n)$. If $t = t'$, we simply say* M *runs in time t.* ■

This definition extends naturally to other resources. Note that if M has a running time that is finite, *i.e.*, $t(n) < \infty$ for all n, then it is decisive. Thus, we can alternatively say that M is halting if it is decisive.

Complexity classes. We are ready to define complexity classes for choice modes. Our previous convention for naming complexity classes extends in a natural way: First note that our notation for complexity classes such as $NTIME(F)$ or $D\text{-}TIME\text{-}REVERSAL(F, F')$ has the general format

$$Mode\text{-}Resources\ (\ Bounds\ )$$

where $Mode$ is either N or D, $Resources$ is a sublist of $time, space, reversal.$ and $Bounds$ is a list of (families of) complexity functions. The complexity class defined by choice machines can be named using the same format: we only have to add symbols for the new modes and resources. The new mode symbols appears in the last column of the table at the beginning of this section. They are

$$Pr, \quad A, \quad Ip, \quad PrA, \quad St, \quad StA$$

denoting (respectively) the probabilistic, alternating, interactive proof, probabilistic-alternating, stochastic, stochastic-alternating modes. We have one new resource, with symbols[11]

$$ALTERNATION \text{ or } ALT.$$

Example 5
(i) Thus $PrTIME(n^{O(1)})$ denotes the class of languages accepted in polynomial time by probabilistic machines. This class is usually denoted PP.
(ii) The class $IpTIME(n^{O(1)})$ contains the class usually denoted IP in the literature. If we introduce (see next chapter) the notion of bounded-error decision, indicated by the subscript 'b', then we have

$$IP = IpTIME(n^{O(1)}).$$

(iii) If F, F' are families of complexity functions, $PrA\text{-}TIME\text{-}SPACE\ (F, F')$ denotes the class of languages that can be accepted by PAMs in simultaneous time-space (t, s) for some $t \in F, s \in F'$.
(iv) We will write $A\text{-}TIME\text{-}ALT(n^{O(1)}, O(1))$ for the class of languages accepted by alternating machines in polynomial time in some arbitrary but constant number of alternations. This class is denoted PH and contains precisely the languages in the polynomial-time hierarchy (chapter 9). ■

Example 6 Note that $\{\otimes\}$-machines (respectively, $\{\oplus\}$-machines) are equivalent to $\{\wedge\}$-machines ($\{\vee\}$-machines). A more interesting observation is that the probabilistic mode is at least as powerful as nondeterministic mode:

$$N\text{-}TIME\text{-}SPACE\text{-}REVERSAL(t, s, r) \subseteq Pr\text{-}TIME\text{-}SPACE\text{-}REVERSAL(t + 1, s, r)$$

[11] Since "alternation" is the name of a mode as well as of a resource, awkward notations such as $A\text{-}ALT(f(n))$ arise.

for any complexity functions t, s, r. To see this, let N be any nondeterministic machine that accepts in time-space (t, s). Let M be the following probabilistic machine: on input w, first toss a coin. If tail, answer YES; otherwise simulate N and answer YES iff N answers YES. The jargon 'toss a coin and if tail then do X, else do Y' formally means that the machine enters a TOSS-state from which there are two next configurations: in one configuration it does X and in the other choice it does Y. The reader may verify that M accepts L(N) in time-space-reversal $(t+1, s, r)$. ■

7.4 Basic Results

In the last section, the least fixed point tree valuation Val_T for a computation tree T is obtained by repeated application of the operator τ_T to the bottom valuation $Val_\perp$. We now obtain Val_T in an alternative, top-down way.

For each integer $m \geq 0$, let T_m denote the prefix of T obtained by pruning away all nodes at level $m+1$. Thus T_0 consists of just the root of T. By monotonicity,

$$Val_{T_m} \sqsubseteq Val_{T_{m+1}}.$$

Lemma 10
(i) For any finite computation tree T, the fixed point of τ_T is unique (and, a fortiori*, equal to the least fixed point Val_T). Moreover, this fixed point is easily computed 'bottom-up' (e.g., by a postorder traversal of T).*
(ii) For any computation tree T (finite or not), the valuation

$$V_T^* := \lim\{\, Val_{T_m} : m \geq 0\}$$

is a fixed point of τ_T.
(iii) V_T^* is equal to the least fixed point, Val_T.
(iv) A computation tree T is accepting/rejecting if and only if it has a finite prefix that is accepting/rejecting.

Proof.(i) This is seen by a bottom-up examination of the fixed point values at each node.
(ii) If u is a leaf then clearly $\tau_T(V_T^*)(u) = V_T^*(u)$. Otherwise, let $u \vdash (u_1, \ldots, u_n)$.

$$\begin{aligned}
\tau_T(V_T^*)(u) &= \gamma_u(V_T^*(u_1), \ldots, V_T^*(u_n)) \\
&= \gamma_u(\lim_{m\geq 0}\{\, Val_{T_m}(u_1)\}, \ldots, \lim_{m\geq 0}\{\, Val_{T_m}(u_n)\}) \\
&= \lim_{m\geq 0}\{\gamma_u(\, Val_{T_m}(u_1), \ldots, \, Val_{T_m}(u_n))\} \\
&= \lim_{m\geq 0}\{\, Val_{T_m}(u)\} \\
&= V_T^*(u)
\end{aligned}$$

This proves that V_T^* is a fixed point of τ_T.
(iii) Since Val_T is the least fixed point, it suffices to show that $V_T^* \sqsubseteq Val_T$. This easily follows from the fact that $V_{T_m} \sqsubseteq Val_T$.
(iv) If a prefix of T is accepting/rejecting then by monotonicity, T is accepting/rejecting. Conversely, suppose T is accepting/rejecting. Then the lower bound of $Val_T(u_0)$ is greater/less than $\frac{1}{2}$, where u_0 is the root of T. By the characterization of Val_T in part (iii), we know that the lower/upper bound of $Val_{T_m}(u_0)$ is monotonically non-decreasing/non-increasing and is greater/less than $\frac{1}{2}$ in the limit as m goes to infinity. Then there must be a first value of m when this lower/upper bound is greater/less than $\frac{1}{2}$. This m gives us the desired finite prefix T_m of T. **Q.E.D.**

This lemma has a two-fold significance: First, part (iv) proves the compactness theorem in the last section. Second, part (iii) shows us an constructive way to compute Val_T, by approximating it from below by Val_{T_m} with increasing m. This method is constructive (in contrast to the $\tau_T^m(V_\perp)$ approximation) because T_m is finite for each m, and the proof of part (i) tells us how to compute Val_{T_m}.

The following lemma is useful for stochastic-alternating computations:

Lemma 11 *Let T be a computation tree of a choice machine* M *on w, and $i \geq 0$.*
(i) If M *is a probabilistic-alternating machine then $\tau_T^{i+1}(V_\perp)(u) = [x, y]$ implies $2^i x$ and $2^i y$ are integers.*
(ii) If M *is a stochastic-alternating machine then $2^{2^i} x$ and $2^{2^i} y$ are integers.*

We leave the proof as an exercise.

We now have the machinery to show that the language accepted by a B-choice machine M is recursively enumerable provided each function in B is computable. To be precise, assume a suitable subset X of $[0, 1]$ consisting of all the 'representable' numbers, and call an interval representable if its endpoints are representable. We assume $0, 1 \in X$. We require X to be dense in $[0, 1]$ (for example, $X \subseteq [0, 1]$ is the set of rational numbers or X is the set of "binary rationals" which are rationals with finite binary expansion). We say $f \in B$ is computable (relative to the representation of X) if it returns a representable value when given representable arguments; moreover this value is computable by some recursive transducer.

Theorem 12 *Let B be a basis set each of whose functions are computable.*
a) The class of languages accepted by B-machines is precisely the class RE of recursively enumerable languages.
b) The class of languages accepted by decisive B-machines is precisely the class REC.

Proof. a) Languages in RE are accepted by B-choice machines since B-choice machines are generalizations of ordinary Turing machines. Conversely, let M be a B-choice machine and w an input word. To show that L(M) is in RE, it is sufficient

to give a deterministic procedure for checking if $w \in L(\mathrm{M})$, where the procedure is required to halt only if w is in $L(\mathrm{M})$. The procedure computes, for successive values of $m \geq 0$, the value $Val_{T_m}(u_0)$ where T_m is the truncation of $T_{\mathrm{M}}(w)$ below level m and u_0 the root. If T_m is accepting for any m, the procedure answers YES. If T_m is non-accepting for all m, the procedure loops. Lemma 10 not only justifies this procedure, but it also shows how to carry it out: the values $Val_{T_m}(u)$ of each node $u \in T_m$ is computed in a bottom-up fashion. The computability of the basis functions B ensures this is possible.

b) We leave this as an exercise. **Q.E.D.**

One can view the theorem as yet another confirmation of Church's thesis. Our next result shows that negation $\neg$ can be avoided in stochastic-alternating machines at the cost of an increase in the number of states. The following generalizes a result for alternation machines in [3].

Theorem 13 *For any stochastic-alternating acceptor* M*, there is a stochastic-alternating acceptor* N *such that* N *has no NOT-states,* $L(\mathrm{M}) = L(\mathrm{N})$*, and for all* w *and* $t, s, r, a \geq 0$*:* M *accepts* w *in (time, space, reversal, alternation)* (t, s, r, a) *iff* N *accepts* w *in (time, space, reversal, alternation)* (t, s, r, a)*.*

Proof. The idea is to use de Morgan's law to move negation to the leaves of the computation tree. Let M be any SAM. We construct a SAM N satisfying the requirements of the theorem. For each state q of M, there are two states q^+ and q^- for N. For any configuration C of M, let C^+ (resp., C^-) denote the corresponding configuration of N where q^+ (resp., q^-) is substituted for q. In N, we regard q_0^+, q_Y^+ and q_N^+ (respectively) as the initial, YES and NO states. However, we identify q_Y^-, q_N^- (respectively) with the NO, YES states (note the role reversal). Of course, the technical restriction does not permit two YES- or two NO-states, so we will identify them ($q_Y^+ = q_N^-, q_N^+ = q_Y^-$). The functions $\gamma(q^+), \gamma(q^-)$ assigned to the states in N are defined from $\gamma(q)$ in M as follows:

$$\gamma(q^+) = \begin{cases} \gamma(q) & \text{if } \gamma(q) \neq \neg \\ \iota & \text{if } \gamma(q) = \neg \end{cases}$$

$$\gamma(q^-) = \begin{cases} \iota & \text{if } \gamma(q) = \neg \\ \wedge & \text{if } \gamma(q) = \vee \\ \vee & \text{if } \gamma(q) = \wedge \\ \textcircled{\scriptsize T} & \text{if } \gamma(q) = \textcircled{\scriptsize T} \\ \oplus & \text{if } \gamma(q) = \otimes \\ \otimes & \text{if } \gamma(q) = \oplus \end{cases}$$

Hence N has no NOT-states. We now state the requirements on transitions of N (this easily translates into an explicit description of the transition table of N).

Suppose that $C_1 \vdash C_2$. If C_1 is not a NOT-configuration then

$$C_1^+ \vdash C_2^+ \text{ and } C_1^- \vdash C_2^-.$$

If C_1 is a NOT-configuration then

$$C_1^+ \vdash C_2^- \text{ and } C_1^- \vdash C_2^+.$$

Our description of N is complete: there are no transitions besides those listed above.

Let T be an accepting computation tree of N for an input word w; it is easy to see that there is a corresponding computation tree $\widehat{T}$ for N with exactly the same time, space, reversal and alternation complexity. In fact there is a bijection between the nodes of T and $\widehat{T}$ such a node labeled C in T corresponds to one labeled C^+ or C^- in $\widehat{T}$. The fact that T and $\widehat{T}$ have identical alternating complexity comes from the carefully-crafted definition of N.

Our theorem is proved if we show that $\widehat{T}$ is accepting. Let T_m be the truncation of the tree T at levels below $m \geq 0$; $\widehat{T}_m$ is similarly defined with respect to $\widehat{T}$. For $h \geq 0$, let V_m^h denote the valuations on T_m given by h-fold applications of the operator τ_{T_m} to $\perp$:

$$V_m^h = \tau_{T_m}^h(V_\perp)$$

and similarly define $\widehat{V}_m^h = \tau_{\widehat{T}_m}^h(V_\perp)$. We now claim that for all m, h and $C \in T_m$,

$$V_m^h(C) = \begin{cases} \widehat{V}_m^h(C^+) & \text{if } C^+ \in \widehat{T} \\ \neg\widehat{V}_m^h(C^-) & \text{if } C^- \in \widehat{T} \end{cases}$$

Here, we have abused notation by identifying the configuration C with the node of T_m that it labels. But this should be harmless except for making the proof more transparent. If $h = 0$ then our claim is true

$$\perp = V_m^h(C) = \widehat{V}_m^h(C^+) = \neg\widehat{V}_m^h(C^-)$$

since $\neg\perp = \perp$. So assume $h > 0$. If C is a leaf of T, it is also easy to verify our claim. Hence assume C is not a leaf. Suppose $C^- \vdash (C_1^-, C_2^-)$ occurs in $\widehat{T}$. Then

$$\begin{aligned} V_m^h(C) &= \gamma(C)(V_m^{h-1}(C_1), V_m^{h-1}(C_2)) \\ &= \gamma(C)(\neg\widehat{V}_m^{h-1}(C_1^-), \neg\widehat{V}_m^{h-1}(C_2^-)) \quad \text{(by induction)} \\ &= \neg\gamma(C^-)(\widehat{V}_m^{h-1}(C_1^-), \widehat{V}_m^{h-1}(C_2^-)) \quad \text{(de Morgan's law for } \gamma(C)) \\ &= \neg\widehat{V}_m^h(C^-). \end{aligned}$$

Similarly, we can show $V_m^h(C) = \widehat{V}_m^h(C^+)$ if $C^+ \vdash (C_1^+, C_2^+)$ occurs in $\widehat{T}$. We omit the demonstration in case C is a NOT-configuration. Finally, noting that $V_m^{m+1} = Val_{T_m}$ and $\widehat{V}_m^{m+1} = Val_{\widehat{T}_m}$, we conclude that $Val_{T_m} = Val_{\widehat{T}_m}$. It follows $\widehat{T}$ is accepting. **Q.E.D.**

Consequence of eliminating negation. This proof also shows that negation can be eliminated in alternating machines and in PAMs. With respect to SAMs without negation, the use of intervals in valuations can be replaced by ordinary numbers in $[0,1]$ *provided we restrict attention to acceptance complexity.* A valuation is now a mapping from $\Delta \subseteq \Delta(\mathrm{M})$ into $[0,1]$. Likewise, a tree valuation assigns a real value in $[0,1]$ to each node in a complete computation tree. We now let $V_\perp$ denote the valuation that assigns the value 0 to each configuration or node, as the case may be. The operator τ_Δ or τ_T on valuations is defined as before. Their least fixed point is denoted Val_Δ or Val_T as before. The connection between the old valuation V and the new valuation V' is simply that $V'(C)$ or $V'(u)$ (for any configuration C or node u) is equal to the lower bound of the interval $V(C)$ or $V(u)$. When we discuss running complexity, we need to consider the upper bounds of intervals in order to reject an input; so we are essentially back to the entire interval.

Convention for this chapter. In this chapter, we only consider alternating machines, PAMs and SAMs with no NOT-states. We are mainly interested in *acceptance complexity.* In this case, we may restrict valuations take values in $[0,1]$ instead of in INT (we call these real values *probabilities*). With this convention, the acceptance rule becomes:

M accepts a word w iff the probability $Val_{\mathrm{M}}(w)$ is greater than $\frac{1}{2}$.

■

It is sometimes convenient to construct SAMs *with* NOT-states, knowing that they can be removed by an application of the preceding theorem.

Suppose we generalize SAMs by allowing the k-ary versions of the alternating-stochastic functions:

$$B_k := \{\max_k, \min_k, \textstyle\bigoplus_k, \otimes_k, \oplus_k\},$$

for each $k \geq 2$. For example,

$$\begin{aligned} \textstyle\bigoplus_3(x,y,z) &= (x+y+z)/3, \\ \oplus_3(x,y,z) &= 1-(1-x)(1-y)(1-z). \end{aligned}$$

Consider generalized SAMs whose basis set is $\cup_{k\geq 2} B_k$. Note that even though the basis set is infinite, each generalized SAM uses only a finite subset of these functions. It is easily seen that with this generalization for the alternating choices ($\max_k$, $\min_k$ and $\otimes_k$), the time complexity is reduced by at most a constant factor. It is a little harder (Exercise) to show the same for the stochastic choices ($\bigoplus_k, \oplus_k, \otimes_k$).

A useful technical result is tape reduction for alternating machines. The following is from Paul, Praus and Reischuk [18].

Theorem 14 *For any k-tape alternating machine* M *accepting in time-alternation $(t(n), a(n))$, there is a simple alternating machine* N *accepting the same language and time-alternation $(O(t(n)), a(n)+O(1))$. Here* N *is 'simple' as in 'simple Turing machines', with only one work-tape and no input tape.*

This leads, in the usual fashion, to a hierarchy theorem for alternating time:

Theorem 15 *Let $t(n)$ be constructible and $t'(n) = o(t(n))$. Then*

$$ATIME(t) - ATIME(t') \neq \emptyset.$$

We leave both proofs as exercises.

Theorem 16 (Space compression) *Let B be any basis set. Then the B-choice machines have the space compression property. More precisely, if* M *is a B-choice machine accepting in space $s(n)$ then there is another* N *which accepts the same language in space $s(n)/2$. Furthermore,* N *has only one work-tape.*

Proof. We only sketch the proof, emphasizing those aspects that are not present in the proof of original space compression theorem in chapter 2. As before, we compress 2 symbols from each of the k work-tapes of M into one *composite symbol* (with k tracks) of N. We show how N simulates one step of M: suppose M is in some configuration C and $C \vdash (C_1, \ldots, C_m)$. Assume that the tape head of N is positioned at the leftmost non-blank cell of its tape. By deterministically making a rightward sweep across the non-blank part of its work-tape, N can remember in its finite state control the two composite symbols adjacent to the currently scanned cell *in each track*: the cells of M corresponding to these remembered symbols constitute the *current neighborhood*. In the original proof, N makes a leftward sweep back to its starting position, updating the contents of the current neighborhood. The new twist is that there are now m ways to do the updating. N can use choice to ensure that each of these possibilities are covered. More precisely, before making the return sweep, N enters a state q such that $\gamma(q) = \gamma(C)$ and then N branches into m different states, each corresponding to a distinct way to update the current neighborhood. Then N can make a deterministic return sweep on each of the m branches. By making some adjustments in the finite state of N, we may ensure that N uses space $s(n)/2$. Further, N is also a B-machine by construction. **Q.E.D.**

For any family of functions B over INT, let B^* denote the *closure* of B (under function composition): B^* is the smallest class of functions containing B and closed under function composition.[12] For example, the function $h(x, y, z) = x \oplus (y \oplus z) = x/2 + y/4 + z/4$ is in the closure of the basis $\{\oplus\}$. The closure of a basis set is also a basis set (in fact, an infinite set).

Theorem 17 (Linear Speedup) *Let B be an admissible family which is closed under function composition, $B = B^*$. Then the B-choice machines have the linear speedup property. More precisely, if* M *is a B-choice machine accepting in time $t(n) > n$ then there is another* N *which accepts the same language in time $n + t(n)/2$.*

[12] More precisely, if $f \in B^*$ is a function of arity k and g_i $(i = 1, \ldots, k)$ are functions of arity m_i in B^* then $f(g_1(\bar{x}_1), \ldots, g_k(\bar{x}_2))$ is a function in B^* of arity $p \leq \sum_{i=1}^{k} m_i$ where $\bar{x}_i$ is a sequence of m_i variables and p is the number of distinct variables in $\bar{x}_1 \bar{x}_2 \cdots \bar{x}_k$.

Proof. The proof is similar to that for ordinary Turing machines in chapter 2, so we only emphasize the new aspects. The new machine N has $k+1$ work-tapes if M has k. Each tape cell of N encodes up to $d > 1$ (for some d to be determined) of the original symbols. N spends the first n steps making a compressed copy of the input. Thereafter, N uses 8 steps to simulates d steps of M. In general, suppose that M is in some configuration C and d moves after C, there are m successor configurations $C_1, \ldots, C_m$ (clearly m is bounded by a function of M and d only). Suppose that the complete computation tree in question is T. The value $Val_T(C)$ is given by

$$f(Val_T(C_1), \ldots, Val_T(C_m))$$

where $f \in B$ since B is closed under function composition. We show how N can determine this f: first N takes 4 steps to determine the contents of the 'current neighborhood' (defined as in the original proof). From its finite state constrol, N now knows f and each C_i $(i = 1, \ldots, m)$. So at the end of the fourth step, N could enters a state q where $\gamma(q) = f$ and such that q has m successors, corresponding to the C_i's. In 4 more steps, N deterministically updates its current neighborhood according to each C_i. It is clear that by choosing $d = 16$, N accepts in time $n+t(n)/2$. One minor difference from the original proof: previously the updated tapes represent the configuration at the first time some tape head leaves the current neighborhood, representing at least d steps of M. Now we simply simulate *exactly* d steps and so it is possible that a tape head remain in the current neighborhood after updating. **Q.E.D.**

As corollary, if we generalize alternating machines by replacing the usual basis set $B = \{\wedge, \vee, \neg\}$ by its closure B^*, then the generalized alternating time classes enjoy the time speedup property. A similar remark holds for the other modes.

7.5 Alternating Time versus Deterministic Space

This section points out strong similarities between alternating time and deterministic space. This motivates a variation of choice machines called the addressable-input model. We first prove the following result of Chandra, Kozen and Stockmeyer:

Theorem 18 *For all t, $ATIME(t) \subseteq DSPACE(t)$.*

Proof. Let M be an alternating machine accepting in time t. We describe a deterministic N that simulates M in space t. Let w be the input and N computes in successive stages. In the mth stage $(m = 1, 2, 3, \ldots)$, N computes Val_{T_m} where T_m is the truncation of the complete computation tree $T_{\mathrm{M}}(w)$ at levels below m. For brevity, write Val_m for Val_{T_m}. If T_m is accepting then N accepts, otherwise it proceeds to the next stage. So if w is not in $L(\mathrm{M})$ then N will not halt.

To complete the proof, we show that the mth stage can be carried out using $O(m)$ space. We describe a procedure to compute the values $Val_m(C)$ of the nodes

C in T_m in a post-order manner. The structure of this search is standard. We inductively assume that the tapes of N contain three pieces of information when we visit a configuration C:

(a) The configuration C. This requires $O(m)$ space. In particular, the input head position can be recorded in $O(\log m)$ space (rather than $O(\log |w|)$ space).

(b) A representation of the path $\pi(C)$ from the root of T_m to C. If the path has length k, we store a sequence of k tuples from the transition table of M where each tuple represents a transition on the path $\pi(C)$. The space for storing the k tuples is $O(m)$ since $k \leq m$. These tuples allow us to "backup" from C to any of its predecessors C' on the path (this simply means we reconstruct C' from C).

(c) All previously computed values $Val_m(C')$ where C' is the child of some node in the path $\pi(C)$. The space is $O(m)$, using the fact that $Val_m(C')$ is 0 or 1.

We maintain this information at each "step". A step (at current configuration C) either involves descending to a child of C or backing up from C to its parent. We descending to a child of C provided C is not a leaf and at least one child of C has not yet been visited. Maintaining (a)-(c) is easy in this case. So suppose we want to backup from C to its parent C'. We claim that $Val_m(C)$ can be determined at this moment. This is true if C is a leaf of T_m. Otherwise, the reason we are backing up to C' is because we had visited both children C_1, C_2 of C. But this meant we had just backed up from (say) C_2 to C, and inductively by our claim, we have determined $Val_m(C_2)$. From (c), we also know $Val_m(C_1)$. Thus we can determine $Val_m(C)$, as claimed. Eventually we determine the value of Val_m at the root. **Q.E.D.**

Discussion. (A) This theorem shows that *deterministic space is at least as powerful as alternating time*. This suggests new results as follows: take a known simulation by deterministic space and ask if it can be improved to an alternating time simulation. This methodology has proven fruitful and has resulted in a deeper understanding of the space resource. Thus, in section 8, a known inclusion $DTIME(t) \subseteq DSPACE(t/\log t)$ was sharped to $DTIME(t) \subseteq ATIME(t/\log t)$. This strengthening apparently lead to a simplification of the original proof. This paradox is explained by the fact that the control mechanism in alternating computation is "in-built"; an alternating simulation (unlike the original space simulation) need not explicitly describe this mechanism.

(B) In fact there is evidence to suggest that alternating time and deterministic space are very similar. For instance, we prove (§7.7) a generalization of Savitch's result, which yields the corollary

$$NSPACE(s) \subseteq ATIME(s^2).$$

This motivates another class of new results: given a known result about deterministic space, try to prove the analogue for alternating time, or vice-versa. For instance, the

last section shows a tape-reduction and a hierarchy theorem for alternating-time; a motivation for these results is that we have similar results for deterministic space. We now give another illustration. In chapter 2, we show that $DSPACE_r(s)$ is closed under complementation for all s finite (*i.e.*, $s(x) < \infty$ whenever defined). We ask for a corresponding result for $ATIME_r(t)$. (Note that the subscript 'r' indicates running complexity.) As it turns out, this result[13] is rather easy for alternating time:

Theorem 19 *For all complexity function $t(n)$,*

$$ATIME_r(t) = co\text{-}ATIME_r(t).$$

Similarly, the following time classes are closed under complementation

$$PrTIME_r(t), StTIME_r(t), PrA\text{-}TIME_r(t), StA\text{-}TIME_r(t).$$

Proof. Recall the construction in theorem 13 of a machine N without negation from another machine M that may have negation. Now let M be an alternating machine. Suppose that we make q_0^- (instead of q_0^+) the start state of N but $q_Y^+ = q_N^-$ remains the YES state. On re-examination of the proof of theorem 13, we see that this N accepts *co-*L(M). The proof for the other time classes are similar. **Q.E.D.**

(C) Continuing our discussion: by now it should be realized that the fundamental technique for space simulation is to 'reuse space'. This usually amounts to cycling through an exponential number of possibilities using the same space. In alternating time, the corresponding technique is to make exponentially many universal or existential choices. While a deterministic space search proceeds from what is known to what is unknown, alternating time search proceeds in reverse direction: it guesses the unknown and tries to reduce it to the known. This remark may be clearer by the end of this chapter.

(D) We should caution that the research programs (A) and (B) have limitations: although the deterministic space and alternating time are similar, it is unlikely that they are identical. Another fundamental difficulty is that whereas sublinear deterministic space classes are important, it is easy to see that alternating machines do not allow meaningful sublinear time computations. This prompted Chandra, Kozen and Stockmeyer to suggest[14] a variation of alternating machines which we now extend to choice machines:

Definition 12 (Addressable-input Machine Model) *An* addressable-input *choice machine* M *is one that is equipped with an extra* address *tape and two distinguished states called the* READ *and* ERROR *states. The address tape has a binary alphabet whose content is interpreted as an integer. Whenever* M *enters the READ state,*

[13] Paul and Reischuk show that if t is time-constructible then $ATIME(t) = co\text{-}ATIME(t)$.

[14] In this suggestion, they are in good company: historically, the read-only input tape of Turing machines was invented for a similar purpose.

the input head is instantaneously placed at the (absolute) position indicated by the address tape. If this position lies outside the input word, the ERROR state will be entered and no input head movement occurs. The machine can still move and use the input head in the usual fashion.

We assume the address tape is one-way (and hence is write-only). Hence for complexity purposes, space on the address tape is not *counted. We also assume that after exiting from the READ state, the contents of the address tape is erased, in an instant.* ■

The addressable-input model is defined so that such machines are at least as powerful as *ordinary* choice machines. However, it is not excessively more powerful; for instance the preceding theorem 18 holds even with this model of alternation (Exercises). This addressable-input model now admits interesting alternating time classes with complexity as small as $\log n$ (not $\log \log n$, unfortunately). We will be explicit whenever we use this version of choice machines instead of the ordinary ones.

7.6 Simulations by Alternating Time

We present efficient simulations of other complexity resources by alternating time. We begin with an alternating time simulation of deterministic space and reversals.

Theorem 20 *Let t, s be complexity functions such that $t(n) \geq 1 + n$. Under the addressable-input model,*

$$\textit{D-TIME-REVERSAL}(t, r) \subseteq \textit{ATIME}(O(r \log^2 t)).$$

If $r(n) \log^2 t(n) \geq n$, then this result holds under the ordinary model.

Proof. Given a deterministic M accepting in time-reversal (t, r), we show an alternating machine N accepting the same language $L(\mathrm{M})$ in time $O(r \log^2 t)$.

Recall the concept of a (full) trace[15] in the proof of chapter 2. On any input w, N existentially chooses some integer $r \geq 1$ and writes r full traces

$$\tau_1, \tau_2, \ldots, \tau_r,$$

on tape 1. These are intended to be the traces at the beginning of each of the r phases. On tape 2, N existentially chooses the time t_i (in binary) of each τ_i,

$$t_1 < t_2 < \cdots < t_r.$$

We may assume τ_r is the trace when the machine accepts. Note that τ_1 (which we may assume correct) is simply the trace of the initial configuration and so $t_1 = 0$.

[15] Briefly, the trace of a configuration in a computation path records its state and for each tape, the scanned symbol, the absolute head position and the head tendencies.

Relative to this sequence, we say that an integer $t \geq 0$ *belongs to phase* j if $t_j \leq t < t_{j+1}$.

Then N proceeds to verify τ_r. To do this, it writes on tape 3 the pairs (τ_{r-1}, t_{r-1}) and (τ_r, t_r) and invokes a procedure *TRACE*. N accepts if and only if this invocation is *successful* (i.e., the procedure accepts). In general, the arguments to *TRACE* are placed on tape 3, and they have the form

$$(\sigma, s_0), (\tau, t_0)$$

where $s_0 < t_0$ are binary integers lying in the range

$$t_i \leq s_0 < t_0 \leq t_{i+1}$$

for some $i = 1, \ldots, r-1$, and σ, τ are traces such that the head tendencies in σ and in τ_i agree, and similarly the head tendencies in τ and in τ_i agree (with the possible exception of $t_0 = t_{i+1}$ and $\tau = \tau_{i+1}$). Intuitively, $TRACE(\sigma, s_0, \tau, t_0)$ accepts if σ is the trace at time s_0, τ is the trace at time t_0, and there is a (trace of a) path from σ to τ. *TRACE* does one of two things:

(i) Suppose $s_0 + 1 < t_0$. Let $t' = \lfloor (s_0 + t_0)/2 \rfloor$. Now *TRACE* existentially chooses a trace τ' where the head tendencies in τ' agree with those of σ. Then it universally chooses to recursively call $TRACE(\sigma, s_0, \tau', t')$ and $TRACE(\tau', t', \tau, t_0)$.

(ii) Suppose $s_0 + 1 = t_0$. *TRACE* verifies τ can be derived from σ in one step of M, and any head motion is consistent with the head tendencies in σ. Of course, we allow the head tendencies to be different but only when σ is the last trace in a phase (it is easy to determine if this is the case). Any head motion in the σ to τ transition causes the corresponding tape cell in τ to be 'marked'. Note that the marked cells were written in some previous phase (unless they are blanks), and our goal is to verify their contents. Suppose that the tape symbols and head positions in the $k+1$ tapes of τ are given by

$$b_0, \ldots, b_k, \qquad n_0, \ldots, n_k.$$

Then *TRACE* universally chooses to call another procedure *SYMBOL* with arguments (i, b_i, n_i, t_0) for each cell n_i in tape i that is marked. Intuitively, $SYMBOL(i, b_i, n_i, t_0)$ verifies that just before time t_0, the tape symbol in cell n_i of tape i is b_i.

We now implement $SYMBOL(i', b', n', t_0)$. If $i' = 0$ then we want to check that the input symbol at position n' is b'. This can be done in $O(\log n)$ steps, using the input addressing ability of our alternating machines (note that $r \log^2 t > \log n$). Otherwise, suppose that t_0 belongs to phase j_0. We then existentially choose some j,

$$j = 0, \ldots, j_0 - 1,$$

some t' and a trace σ'. Intuitively, this means that cell n' in tape i' was last visited by σ' which occurs at time t' in phase j. We want the following to hold:
(a) $t_j \leq t' < t_{j+1} \leq t_0$.
(b) The head tendencies σ' and in τ_j agree.
(c) The head i' is in position n' scanning symbol b' in σ'.
(d) On each tape, the head position in σ' lies in the range of possible cell positions for that phase j.
(e) On tape i', cell n' is not visited in any of phases $j+1, j+2, \ldots, j_0$.
Conditions (a)-(e) can be directly verified using the information on tapes 1 and 2. Armed with σ' and t', we then universally choose one of two actions: either call $TRACE(\tau_j, t_j, \sigma', t')$ or $TRACE(\sigma', t', \tau_{j+1}, t_{j+1})$. If $t_j = t'$ then the first call is omitted.

Correctness. Let us show that *TRACE* and *SYMBOL* are correct. Suppose input w is accepted by M. Then it is not hard to see that N accepts. To show the converse, suppose N accepts w relative to some choice of traces $\tau_1, \ldots, \tau_r$ in tape 1 and times $t_1, \ldots, t_r$ on tape 2. Suppose the call $TRACE(\sigma, s_0, \tau, t_0)$ is successful and t_0 belongs to phase j. Then this call generates a trace-path

$$(\sigma_0, \ldots, \sigma_m) \tag{7.4}$$

from $\sigma_0 = \sigma$ to $\sigma_m = \tau$ (where $m = t_0 - s_0$) with the following properties:

1. σ_{i-1} derives σ_i for $i = 1, \ldots, m$ according to the rules of M.

2. Each pair (σ_{i-1}, σ_i) in turn generates at most $k+1$ calls to *SYMBOL*, one call for each "marked" cell in σ_i. Each of these calls to *SYMBOL* leads to acceptance. For this reason we call (7.4) a 'successful' trace-path for this call to *TRACE*.

3. Some of these calls to *SYMBOL* in turn calls *TRACE* with arguments belonging some phase ℓ $(1 \leq \ell < j)$. We call any such phase ℓ a *supporting phase* of $TRACE(\sigma, s_0, \tau, t_0)$.

Notice that if phase ℓ is a supporting phase for some accepting call to *TRACE* then there must be two successful calls of the form

$$TRACE(\tau_\ell, t_\ell, \sigma', t') \quad \text{and} \quad TRACE(\sigma', t', \tau_{\ell+1}, t_{\ell+1}) \tag{7.5}$$

for some σ', t'. We claim:

> (a) If $TRACE(\sigma, s_0, \tau, t_0)$ accepts and phase ℓ is a supporting phase of $TRACE(\sigma, s_0, \tau, t_0)$, then the traces $\tau_1, \ldots, \tau_{\ell+1}$ on tape 1 and times $t_1, \ldots, t_{\ell+1}$ on tape 2 are correct (i.e., τ_i is the trace at the beginning of the ith phase at time t_i for $i = 1, \ldots, \ell+1$.)

(b) If, in addition, we have that $\sigma = \tau_j$ and $s_0 = t_j$ for some $j = 1, \ldots, r$, then τ is indeed the trace of the t_0th configuration in the computation path of M on input w. (Note that (b) implies the correctness of *SYMBOL*.)

We use induction on ℓ. Case $\ell = 1$: τ_1 is always correct and $t_1 = 0$. By (7.5), we see directly that there must be two successful calls of the form $TRACE(\tau_1, t_1, \sigma', t')$ and $TRACE(\sigma', t', \tau_2, t_2)$. One immediately checks that this implies τ_2, t_2 are correct. This proves (a). Part (b) is immediate.

Case $\ell > 1$: for part (a), again we know that there are two successful calls of the form (7.5). But notice that phase $\ell - 1$ is a supporting phase for the first of these two calls: this is because in τ_ℓ, some tape head made a reversal that this means that this head scans some symbol last visited in phase $\ell - 1$. Hence by induction hypothesis, the traces $\tau_1, \ldots, \tau_\ell$ and times $t_1, \ldots, t_\ell$ are correct. Furthermore, as in (7.4), we have a successful trace-path from τ_ℓ to $\tau_{\ell+1}$. Each trace (except for τ_ℓ) in the trace-path in turn generates a successful call to *SYMBOL* with arguments belonging to some phase less than ℓ, and by induction (b), these are correct. Thus $\tau_{\ell+1}$ and $t_{\ell+1}$ are correct. For part (b), we simply note that $j - 1$ is a support phase for such a call to *TRACE* by the preceding arguments. So by part (a), τ_j and t_j are correct. Then we see that there is a trace-path starting from τ_j as in (7.4) that is correct. Part (b) simply asserts that the last trace in (7.4) is correct. This completes our correctness proof.

Complexity. The guessing of the traces τ_i and times t_i on tapes 1 and 2 takes alternating time $O(r \log t)$. If the arguments of *TRACE* belongs to phase j, then *TRACE* may recursively call itself with arguments belonging to phase j for $O(\log t)$ times along on a computation path of N. Then *TRACE* calls *SYMBOL* which in turn calls *TRACE* but with arguments belonging to phase $< j$. Now each call to *TRACE* takes $O(\log t)$ alternating steps just to set up its arguments (just to write down the head positions). Thus it takes $O(\log^2 t)$ alternating steps between successive calls to *SYMBOL*. In the complete computation tree, we make at most r calls to *SYMBOL* along any path. This gives an alternating time bound of $O(r \log^2 t)$. **Q.E.D.**

Corollary 21 $DREVERSAL(r) \subseteq ATIME(r^3)$

We next show that alternating time is at least as powerful as probabilistic time. The proof is based on the following idea: suppose a probabilistic machine accepts an input x in $m \geq 0$ steps and T is the computation tree. If T is finite and all its leaves happen to lie a fixed level $m \geq 0$ ($m = 0$ is the level of the root) then it is easily seen that T is accepting iff the number of accepting leaves is more than half of the total (i.e. more than 2^{m-1} out of 2^m). In general, T is neither finite nor will all the leaves lie in one level. But we see that if T_m is the truncation of T to level m, then T_m if accepting iff the sum of the "weights" of accepting leaves in T_m is

more than 2^{m-1}. Here we define a leaf at level i ($0 \leq i \leq m$) to have a weight of 2^{m-i}. It is now easy to simulate a probabilistic machine M that uses time $t(n)$ by a deterministic machine N using space $t(n)$, by a post-order traversal of the tree $T_{t(n)}$. But we now show that instead of deterministic space $t(n)$, alternating time $t(n)$ suffices.

Theorem 22 *For all complexity functions t, $PrTIME(t) \subseteq ATIME(t)$.*

Proof. Let M be a probabilistic machine that accepts in time t. We describe an alternating machine N that accepts in time t. Let x be an input and T_m be the computation tree of M on x restricted to configurations at level at most m. For any configuration C in T_m, define

$$VAL_m(C) = 2^{m-level(C)}\, Val_{T_m}(C)$$

where Val_{T_m} is, as usual, the least fixed point valuation of T_m. We abuse notation with the usual identification of the nodes of T_m with the configurations labeling them. Thus if C is the root then $VAL_m(C) = 2^m\, Val_\Delta(C)$. If C is not a leaf, let C_L and C_R denote the two children of C. Observe that

$$VAL_m(C) = VAL_m(C_L) + VAL_m(C_R).$$

Regarding $VAL_m(C)$ as a binary string of length $m+1$, we define for $i = 0, \ldots, m$,

$$\begin{aligned} BIT_m(C,i) &:= i^{th} \text{ bit of } VAL_m(C) \\ CAR_m(C,i) &:= i^{th} \text{ carry bit of the summation } VAL_m(C_L) + VAL_m(C_R) \end{aligned}$$

where we assume that $i = 0$ corresponds to the lowest order bit. It is easy to see that the following pair of mutually recursive formulas hold:

$$\begin{aligned} BIT_m(C,i) &= BIT_m(C_L,i) \overline{\oplus} BIT_m(C_R,i) \overline{\oplus} CAR_m(C,i-1) \\ CAR_m(C,i) &= \lfloor \frac{BIT_m(C_L,i)) + BIT_m(C_R,i) + CAR_m(C,i-1)}{2} \rfloor \end{aligned}$$

Here, $\overline{\oplus}$ denotes the exclusive-or Boolean operation: $b \overline{\oplus} b' = 1$ iff $b \neq b'$. If $i = 0$, $CAR_m(C, i-1)$ is taken to be zero.

If C is a leaf, we define $CAR_m(C,i) = 0$ and

$$BIT_m(C,i) = \begin{cases} 1 & \text{if } i \text{ is equal to } m - level(C) \text{ and } C \text{ answers YES;} \\ 0 & \text{otherwise.} \end{cases}$$

To simulate M on input x, N first guesses the value $m = t(|x|)$ in unary in tapes 1 and 2. Note that M accepts iff $VAL_m(x) > 1/2$, iff there exists an i, $0 \leq i < m-1$, such that

$$\text{Either} \quad BIT_m(C_0(x), m) = 1 \tag{7.6}$$
$$\text{or} \quad BIT_m(C_0(x), m-1) = BIT_m(C_0(x), i) = 1. \tag{7.7}$$

N checks for either condition (7.6) or (7.7) by an existential choice. In the latter case, N makes a universal branch to check that $BIT_m(C_0(x), m-1) = 1$ and, for some existentially guessed unary integer $0 < i < m-1$, $BIT_m(C_0(x), i) = 1$.

It remains to describe the subroutine to verify $BIT_m(C, i) = b$ for any arguments C, i, b. It is assumed that just before calling this subroutine the following setup holds. N has the first m cells on tapes 1,2 and 3 marked out. Head 1,2 and 3 are respectively keeping track of the integers i, $level(C)$ and $i+level(C)$, in unary. Moreover, because of the marked cells, it is possible to detect when these values equals 0 or m. The configuration C is represented by the contents and head positions of tapes 4 to $k+3$ (k is the number of work-tapes of M) and the input tape. This setup is also assumed when calling the subroutine to verify $CAR_m(C, i) = b$.

With this setup, in constant time, N can decide if C is a leaf of T_m (i.e. either C is terminal or $level(C) = m$) and whether $i = m - level(C)$. Hence, in case C is a leaf, the subroutine can determine the value of $BIT_m(C, i)$ in constant time. If C is not a leaf, say $C \vdash (C_L, C_R)$, then N guesses three bits, b_1, b_2 and c such that

$$b = b_1 \overline{\oplus} b_2 \overline{\oplus} c.$$

It then universally branches to verify

$$BIT_m(C_L, i) = b_1,\ BIT_m(C_R, i) = b_2,\ CAR_m(C, i-1) = c.$$

It is important to see that N can set up the arguments for these recursive calls in constant time. A similar subroutine for CAR_m can be obtained.

It remains to analyze the time complexity of N. We first define the function t_m to capture the complexity of BIT_m and CAR_m:

$$t_m(d, i) = \begin{cases} 1 & \text{if } d = m \\ 1 + t_m(d+1, 0) & \text{if } (d < m) \wedge (i = 0) \\ 1 + \max\{t_m(d+1, i), t_m(d, i-1)\} & \text{else.} \end{cases}$$

An examination of the recursive equations for BIT_m and CAR_m reveals that the times to compute $BIT_m(C, i)$ and $CAR_m(C, i)$ are each bounded by $O(t_m(d, i))$ where $d = level(C)$. On the other hand, it is easily checked that the recursive equations for t_m satisfy

$$t_m(d, i) \leq m - d + i + 1 = O(m)$$

since i and d lie in the range $[0..m]$. This proves that the time taken by N is $O(m) = O(t(|x|))$. **Q.E.D.**

This theorem, together with that in section 5, imply that deterministic space is at least as powerful as probabilistic or alternating time separately:

$$PrTIME(t) \cup ATIME(t) \subseteq DSPACE(t)$$

It is not clear if this can be improved to showing that deterministic space is at least as powerful as probabilistic-alternating time. If this proves impossible, then the combination of probabilism and alternation is more powerful than either one separately. This would not be surprising since probabilism and alternation seems to be rather different computing concepts. Our current best bound on simulating probabilistic-alternating time is given next.

Theorem 23 *For all complexity functions t, PrA-TIME$(t) \subseteq$ ATIME$(t \log t)$.*

Proof. We use the same notations as the proof of the previous theorem. Let M be a PAM that accepts in time t. Fix any input x and let T_m be the complete computation tree of M on x restricted to levels at most m, and define VAL_m, BIT_m and CAR_m as before. There is one interesting difference: previously, the values m and i in calls to the subroutines to verify $BIT_m(C,i) = b$ and $CAR_m(C,i) = b$ were encoded in unary. We now store these values in binary (the reader is asked to see why we no longer use unary).

Consider the verification of $BIT_m(C,i) = b$ for any inputs C, i, b, using alternating time. If C is a leaf this is easy. Suppose C is a TOSS-configuration. Then we must guess three bits b_1, b_2, c and verify that $BIT_m(C_L,i) = b_1$, $BIT_m(C_R,i) = b_1$ and $CAR_m(C,i-1) = c$. Here we use an idea from the design of logic circuits: in circuits for adding two binary numbers, the carry-bits can be rapidly generated using what is known as the 'carry-look-ahead' computation. In our context, this amounts to the following condition:

$$CAR_m(C,i) = 1 \iff \text{if there is a } j \ (j = 0, \ldots, i) \text{ such that } BIT_m(C_L,j) = BIT_m(C_R,j) = 1 \text{ and for all } k = j+1, \ldots, i, \text{ either } BIT_m(C_L,k) = 1 \text{ or } BIT_m(C_L,k) = 1.$$

In $O(\log m)$ alternating steps, we can easily reduce these conditions to checking $BIT_m(C',j) = b'$ for some j, b' and C' a child of C. (We leave this detail as exercise.)

Finally, suppose C is a MIN-configuration (a MAX-configuration is handled similarly). By definition $BIT_m(C,i) = BIT_m(C_L,i)$ iff $VAL_m(C_L) < VAL_m(C_R)$, otherwise $BIT_m(C,i) = BIT_m(C_R,i)$. Now $VAL_m(C_L) < VAL_m(C_R)$ iff there exists a j $(0 \le j \le m)$ such that

$$\begin{aligned} BIT_m(C_L,h) &= BIT_m(C_R,h), \text{ (for } h = j+1, j+2, \ldots, m), \\ BIT_m(C_L,j) &= 0, \\ BIT_m(C_R,j) &= 1. \end{aligned}$$

Again, in $O(\log m)$ time, we reduce this predicate to checking bits of $VAL_m(C')$, C' a child of C.

To complete the argument, since each call to check a bit of $VAL_m(C)$ is reduced in $O(\log m)$ steps to determining the bits of $VAL_m(C')$ where $level(C') = level(C)+1$, there are at most m such calls on any computation path. To generate a call to C', we use $O(\log m)$ time, so that the length of each path is $O(m \log m)$. **Q.E.D.**

7.7 Further Generalization of Savitch's Theorem

Savitch's theorem says that for all $s(n) \geq \log n$, $NSPACE(s) \subseteq DSPACE(s^2)$. Chapter 2 gives a generalization of Savitch's theorem. In this section, we further improve the generalization in three ways: (i) by using alternating time, (ii) by allowing small space bounds $s(n)$, i.e., $s(n) < \log n$, and (iii) by extending the class of simulated machines from nondeterministic to alternating machines.

Consider what happens when $s(n) < \log n$. Savitch's proof method gives only the uninteresting result $NSPACE(s) \subseteq DSPACE(\log^2 n)$. Monien and Sudborough [14] improved this so that for $s(n) < \log n$,

$$NSPACE(s) \subseteq DSPACE(s(n) \log n).$$

Using addressable-input alternating machines, Tompa [21] improved the Monien-Sudborough construction to obtain:

$$NSPACE(s) \subseteq ATIME(s(n)[s(n) + \log n])$$

for all $s(n)$. Incorporating both ideas into the generalized Savitch's theorem of chapter 2, we get:

Theorem 24 *For all complexity functions $t(n) > n$,*

$$N\text{-}TIME\text{-}SPACE(t, s) \subseteq ATIME(s(n) \log \frac{n \cdot t(n)}{s(n)})$$

where the alternating machine here is the addressable-input variety.

Proof. Let M be a nondeterministic machine accepting in time-space (t, s). We describe a addressable-input alternating machine N to accept L(M). Let x be any input, $|x| = n$. M begins by existentially guessing $t = t(n)$ and $s = s(n)$ and marking out s cells on each work tape.

We number the n cells of the input tape containing x as $0, 1, \ldots, n-1$ (rather than the conventional $1, \ldots, n$). We will divide the cells $0, 1, \ldots, n-1$ of the input tape into intervals I_w (subscripted by words $w \in \{L, R\}^*$) defined as follows:

$$i \in I_w \iff \text{the most significant } |w| \text{ bits in the binary representation of } i \text{ corresponds to } w$$

where the correspondence between words in $\{L, R\}^*$ and binary strings is given by $L \leftrightarrow 0$ and $R \leftrightarrow 1$. It is also assumed here that the binary representation of i is

expanded to exactly $\lceil \log n \rceil$ bits. Clearly I_w is an interval of consecutive integers. For example: with $n = 6$,

$$I_\epsilon = [0..5],\ I_L = [0..3],\ I_R = [4..5],\ I_{RR} = \emptyset.$$

Observe that

$$I_w = I_{wL} \cup I_{wR} \text{ and } |I_{wL}| \geq |I_{wR}| \geq 0.$$

The *L-end* (resp., *R-end*) *cell* of a non-empty interval is the leftmost cell (resp., rightmost) cell in that interval.

A storage configuration is a configuration in which the contents as well as head position of the input tape are omitted. Let S, S' be storage configurations of M, $d, d' \in \{L, R\}$, $w \in \{L, R\}^*$. Let $conf(S, d, w)$ denote the configuration in which the contents and head positions in the work-tapes are specified by S, with input tape containing the fixed x and the input head scanning the d-end cell of interval I_w. In the course of computation, N will evaluate the two predicates $REACH$ and $CROSS$ defined next. The predicate

$$REACH(S, S', d, d', w, m)$$

holds if there is a computation path π of length at most m from $conf(S, d, w)$ to $conf(S', d', w)$ where the input head is restricted to the interval I_w throughout the computation, and the space used is at most s. Recall that s is the guessed value of the maximum space usage $s(|x|)$. Let $\overline{L}$ denote R and $\overline{R}$ denote L. Then the predicate

$$CROSS(S, S', d, d', w, m)$$

holds if there is a computation path of length at most m from $conf(S, \overline{d}, wd)$ to $conf(S', \overline{d'}, wd')$ where the input head is restricted to the interval I_w throughout the computation, and the space used is at most s. Observe that the intervals I_{wd} and $I_{wd'}$ used in this definition are adjacent and $I_{wd} \cup I_{wd'} \subseteq I_w$ (if $d = d'$ then this inclusion is proper). We assume in this definition I_{wR} is non-empty; this automatically implies I_{wL} is non-empty. For instance: $CROSS(S, S', L, R, RLR, m)$ holds means there is a path from $conf(S, R, RLRL)$ to $conf(S', L, RLRR)$ of length at most m, as illustrated in the following figure.

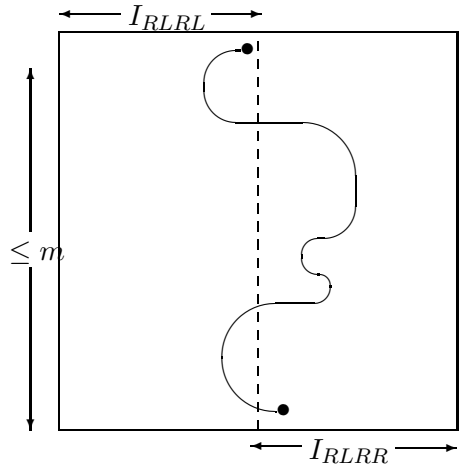


Figure 7.1 The input head positions implied by $CROSS(S, S', L, R, RLR, m)$

We may assume that when M halts, its input head returns to cell 0. The simulation of M amounts to the evaluation of $REACH(S_0, S_f, L, L, \epsilon, t)$ where S_0 is the (trivial) initial storage configuration (independent of x) and S_f is some existentially guessed accepting storage configuration that uses at most s space, λ being the empty string.

We describe the recursive evaluation of $REACH(S, S', d, d', w, m)$. It is assumed that the following setup holds at the time of calling the procedure: S and S' are represented by the contents and head positions on designated tapes of N (if M has k work-tapes, N uses $2k$ tapes for S and S'). The argument w is on the address tape of the N. The value m is written in binary on another tape. One of the following three cases hold:

(i) $|w| = \lceil \log n \rceil$: this condition can be checked in constant time (for instance, by keeping a tally of the length of w) provided that before the simulation begins, the unary representation of $\lceil \log n \rceil$ is guessed and verified once and for all. In this case $|I_w| = 1$ and N enters the read state to read the input symbol $x[w]$ indexed by w. From the definition of $REACH$, from now on N can ignore its input tape and call the predicate

$$REACHABLE(conf(S, d, w), conf(S', d', w), m)$$

where $REACHABLE$ is the predicate in the original proof of the generalized Savitch's theorem (chapter 2, section 7). Recall that the predicate $REACHABLE(C, C', m)$ holds if there is a computation path from configuration C to configuration C' of at most m steps using at most s space. The original simulation in chapter 2 uses $O(s \log \frac{m}{s}) = O(s \log \frac{t}{s})$ space, but this is easily modified to give us alternating time $O(s \log \frac{t}{s})$ (Exercises). Another slight modification is to make $REACHABLE$reject at once if input head ever moves at any time during the simulation.

(ii) $|I_{wR}| = 0$: then call $REACH(S, S', d, d', wL, m)$. Note that $|I_{wR}| = 0$ iff the binary number corresponding to wR is greater than $n - 1$. This is easily checked, for instance, by entering the READ state and seeing if we next enter the ERROR state.

(iii) $|I_{wR}| \geq 1$ (so $|w| < \lceil \log n \rceil$): N existentially guesses whether there is a computation path π from $conf(S, d, w)$ to $conf(S', d', w)$ with the input head restricted to I_{wd}. If it guesses 'no' (and it will not make this guess unless $d = d'$) then it next calls

$$REACH(S, S', d, d, wd, m)$$

If it guesses 'yes' (it could make this guess even if $d = d'$) then it chooses existentially two storage configurations S'', S''' in the computation path π and then chooses universally to check one of the following:

$$REACH(S, S'', d, \overline{d}, wd, m),$$
$$CROSS(S'', S''', d, d', w, m),$$
$$REACH(S''', S', \overline{d'}, d', wd', m).$$

N existentially guesses one of the cases (i)-(iii), and then universally checks that its guess is correct as well as performs the respective actions described under (i)-(iii). This completes the description of *REACH*.

The subroutine for $CROSS(S, S', d, d', w, m)$ has two cases:

(i)' $m \leq s$: in this case, N can check the truth of the predicate in time s (since a nondeterministic machine is just a special case of alternation).

(ii)' $m > s$: N guesses two storage configurations S'', S''' and a value $d'' \in \{L, R\}$ and universally branches to check

$$CROSS(S, S'', d, d'', w, m/2),$$
$$REACH(S'', S''', \overline{d''}, \overline{d''}, wd'', m) and$$
$$CROSS(S''', S', d'', d', w, m/2).$$

We should explain this choice of S'', S''', d'': if there is a computation path from $conf(S, \overline{d}, wd)$ to $conf(S', \overline{d'}, wd')$ that makes $CROSS(S, S', d, d', w, m)$ true then this path can be broken up into several disjoint portions where the input head is confined to I_{wL} or to I_{wR} in each portion. Consider the portion π of the path that contains the configuration at time $m/2$: let π be confined to the interval $I_{wd''}$ for some d'', and let the storage configurations at the beginning and end of π be S'' and S''', respectively. With this choice of d'', S'', S''', it is clear that the recursion above is correct.

Note that it is unnecessary to check for $m \leq s$ in *REACH* (since *REACH* does not reduce m in making recursive calls); likewise it is unnecessary to check for $|I_w| = 1$ in *CROSS* (since it is called by *REACH* only with $|I_w| \geq 2$). Observe that every two successive calls to *REACH* or *CROSS* result in either $|w|$ increasing by at least one or m decreasing to at most $m/2$. Hence in $2(\lceil \log n \rceil + \log(t/s)) = O(\log(nt/s))$ recursive calls, we reduce the input to the 'basis' cases where either $m \leq s$ or $|I_w| = 1$. Each recursive call of *REACH* or *CROSS* requires us to guess the intermediate storage configurations S'', S''' in time $O(s)$. Hence in $O(s \log(nt/s)])$ steps we reach the basis cases. In these basis cases, the time used is either $O(s)$ time or that to compute $REACHABLE(C, C', m)$. The latter is $O(s \log(t/s))$ as noted before. The total time is the sum of the time to reache the basis cases plus the time for basis cases. This is $O(s \log(nt/s)])$. **Q.E.D.**

The structure of the preceding proof involves dividing at least one of two quantities in half until the basis case. An immediate consequence of the above results is this:

Corollary 25
(i) $NSPACE(s) \subseteq ATIME(s^2)$.
(ii) $PrTIME(n^{O(1)}) \subseteq ATIME(n^{O(1)}) = PrA\text{-}TIME(n^{O(1)}) = PSPACE$.

Borodin [3] observed that Savitch's theorem is capable of generalization in another direction. Incorporating Borodin's idea to the previous theorem yields the following "super" Savitch's theorem. Recall the definition of alternating complexity in section 3.

Theorem 26 *Let $t(n) > n, s(n)$ and $a(n)$ be any complexity functions. Then*

$$A\text{-}TIME\text{-}SPACE\text{-}ALTERNATION(t, s, a) \subseteq ATIME(s(n)[a(n) + \log \frac{n \cdot t(n)}{s(n)}])$$

where alternating machines are the addressable-input variety.

Proof. Suppose an alternating machine M accepts in time, space and alternating complexity of $t(n), s(n)$ and $a(n)$. On input x, the machine N begins by guessing the values $t_0 = t(|x|)$ and $s_0 = s(|x|)$. Let $T(x) = T_{t_0,s_0}(x)$ be the computation tree on x restricted to nodes at level $\leq t_0$ and using space $\leq s_0$. There are two procedures involved: The main procedure evaluates a predicate $ACCEPT(C)$ that (for any configuration C as argument) evaluates to true if $C \in T(x)$ and the least fixed point value $Val_T(C)$ of C is equal to 1. The other procedure we need is a variation of the predicate $REACH(S, S', d, d', w, m)$ in the proof of theorem 24. Define the new predicate

$$REACH'(S, S', v, v', w, m)$$

where S, S' are storage configurations, $v, v', w \in \{L, R\}^*$ and $m \geq 1$ such that $|wv| = |wv'| = \lceil \log n \rceil$. Let $conf(S, w)$ where $|w| = \lceil \log n \rceil$ denote the configuration whose storage configuration is S and input tape contains x and the input head is at position indicated by w. The predicate $REACH'$ evaluates to true provided there is a computation path π of length $\leq m$ from $conf(S, wv)$ to $conf(S', wv')$ such that all the intermediate configurations C use space $\leq s$ and has input head restricted to the interval I_w. We further require that

(a) C and $conf(S, wv)$ have opposite types, i.e., C is a MIN-configuration if and only if $conf(S, wv)$ is a MAX-configuration. Note that we assume M has no NOT-configurations.

(b) The computation path π we seek must have only configurations of the same type as C with the sole exception of its last configuration (which is equal to $conf(S, wv)$, naturally).

It is clear that we can compute $REACH'$ in alternating time $O(s \log t/s)$ as in the case of $REACH$.

The procedure $ACCEPT(C)$ proceeds as follows: suppose C is an MAX-configuration (resp., MIN-configuration). Then the algorithm existentially (resp., universally) chooses in time $O(s)$ a configuration C' with opposite type than C. Let $C = conf(S, v)$ and $C' = conf(S', v')$. Regardless of the type of C, the algorithm existentially chooses to call the following subroutines:

(1) $ACCEPT(C')$

(2) $\neg REACH'(S, S', v, v', \epsilon, t_0)$ where the values S, S', v, v' are related to C, C' as above. Of course, by $\neg REACH'$ we mean that the procedure first enters a NOT-state and then calls $REACH'$. (Here is an occasion where it is convenient to re-introduce NOT-states.) The reader should easily see that the procedure is correct.

We now analyze the complexity of the procedure $ACCEPT$. For any configuration C let T_C be the subtree of configurations reachable from C. Define $depth(C)$ to be the minimum k such that there is prefix T' of T_C such that T' is accepting and each path in T' has at most k alternation. In particular, observe that if x is accepted by M then $depth(C_0(x))$ is at most $a(|x|)$, with $C_0(x)$ the initial configuration. Let $W(k)$ be the (alternating) time required by the procedure for $ACCEPT(C)$ on input C with depth k. Then we have

$$W(k) = O(s) + \max\{s \log \frac{t}{s}, W(k-1)\}.$$

To see this, suppose C is an MAX- (resp., MIN-) configuration. Then $O(s)$ is the time to existentially (resp., universally) choose the configurations C' reachable from C; $s \log t/s$ is the time to decide the predicate $REACH'$; and $W(k-1)$ is the time

to recursively call $ACCEPT(C')$. It is easy to deduce that $W(k)$ has solution given by:

$$W(k) = O(s \cdot [k + \log t/s]).$$

The theorem follows immediately. **Q.E.D.**

This is still not the last word on extensions of Savitch's theorem! We return to this in the next chapter.

7.8 Alternating Time versus Deterministic Time

The main result of this section is the following theorem:

Theorem 27 *For all t, $DTIME(t) \subseteq ATIME(\frac{t}{\log t})$.*

Tompa and Dymond [6] obtained this result by adapting the result of Hopcroft, Paul and Valiant [12] showing $DTIME(t) \subseteq DSPACE(t/\log t)$. Adleman and Loui [1] gave an interesting alternative proof of the Hopcroft-Paul-Valiant result. The Hopcroft, Paul and Valiant achievement showed for the first time that space is a more powerful resource than time in a "sufficiently" powerful model of computation (multi-tape Turing machines). Earlier results by Paterson [17] already established such results for simple Turing machines, but the techniques were special to simple Turing machines. Paul and Reischuk extended the Hopcroft, Paul and Valiant result to alternating time, but their simulation needed alternating time $t \log\log t/\log t$. We shall assume the addressable-input model of alternation in case $t/\log t = o(n)$ in the theorem, but otherwise, the regular model suffices.

An interesting corollary of this result, in conjunction with the alternating time hierarchy theorem at the end of section 4, is that there are languages in $DLBA$ that cannot be accepted in deterministic linear time.

7.8.1 Reduction of Simulation to a Game on Graphs.

First consider the simpler problem of simulating a deterministic Turing machine using as little deterministic space as possible. A key step is the reduction of this problem to a combinatorial question on graphs. Suppose a deterministic k-tape machine M accepts an input in $t > 0$ steps. Our goal is to describe a deterministic machine N that simulates M using as little space as possible.

Let $B = B(t) > 0$ be the *blocking factor*, left unspecified for now. For $i = 0, 1, \ldots, \lceil t/B \rceil$, let

$$t_i := iB$$

be *time samples*, and let the cells of each work-tape be grouped into *blocks* consisting of B consecutive cells. For each block b, let *neighborhood*(b) denote the set of 3

blocks consisting of b and the two adjacent blocks on either side of b. We construct a directed acyclic graph $G = (V, E)$ with node set

$$V = \{0, 1, \ldots, \lceil t/B \rceil\}$$

and *labels* for each node. The label for a node i consists of the following two pieces of information:

(i) positions $h_0, \ldots, h_k$ of the $k+1$ tape heads and

(ii) a state q.

We say that this label of node i is *correct* if at time sample t_i, the machine is in state q and the heads are at positions given by $h_0, \ldots, h_k$. We may say that block b is *visited* in time sample t_j if the label of node j says that there is a tape head somewhere in b. Note that this definition is relative to the labeling, regardless of its correctness. Once the labels are given, we can define an edge set E as follows. The edges in E are those (i, j) satisfying one of two requirements:

> (a) There is a tape block b visited at time sample t_j, another tape block b' visited time sample t_i such that *neighborhood*$(b) \cap$ *neighborhood*(b') is non-empty and, previous to sample time t_j, b' is last visited in time sample t_i.
> (b) There is a tape block b visited in time sample t_j such that *neighborhood*(b) contains a block that has never been visited in time samples before t_j, and $i = 0$.

If $(i, j) \in E$ then necessarily $i < j$, and if the labeling is correct then $(i, i+1)$ must be in E. Let *neighborhood*(j) denote the union of the blocks in *neighborhood*(b) where b range over all blocks visited in time sample t_j. Clearly *neighborhood*(j) has exactly $3k$ blocks. Let b be visited in time sample t_j. Then there 5 blocks b' such that *neighborhood*$(b') \cap$ *neighborhood*(b) is non-empty. Each such b' contributes an edge of the form $(i, j) \in E$ for some i. This implies that the indegree of each node in G is at most $5k$. (The outdegree of G is similarly bounded by $5k$; but this fact is not needed.)

A description of G together with its labels can be written down using at most

$$\frac{t \log t}{B}$$

space. This space is less than t if B is larger than $\log t$. We attempt to find such a graph G by testing successively larger values of t, and for each t, cycling through all ways of assigning labels. It remains to show how to verify a proposed labelling. The idea is that each node in G can be 'expanded' in the following sense: the *expansion* of a node $i \in G$ consists of the contents of the blocks in *neighborhood*(i) in time sample t_i. Note that the expansion of i can be encoded using $O(B)$ space. The edges of G define a predecessor-successor relationship: (i, j) is an edge mean that i is a *predecessor* of j, and j the *successor* of i. Next we make an important but elementary observation:

> (*) If we already have the expansions of all the predecessors of node $i \geq 1$ then we may expand node i simply by simulating the machine starting from the moment $(i-1)B$.

To do this, we first reconstruct the contents of blocks in *neighborhood*$(i-1) \cup$ *neighborhood*(i), using the expansions of the predecessors of i. (There could be overlap among the predecessor expansions, but it is easy to only use the contents of the most recent version of a block.) Now simulate M starting from time sample t_{i-1} to time sample t_i. At the end of the simulation, we may assume that the expansion of node i is now available, in addition to the previous expansions. Details can be filled in by the reader. Let us say that a node i is *verified* if we confirm that its label (i.e., head positions and state) is correct.

> (**) If the predecessors of node i are expanded and verified then we can also expand and verify node i.

This is because we can compare the state and head positions in the expansion of i with the labels of node i.

Now we can give a nondeterministic procedure to verify G: nondeterministically expand nodes, one at a time. At any moment, the tapes of the simulator contain some number of expanded nodes. Those nodes whose only predecessor is node 0 can be expanded at any moment; for any other node i, we can only expanded i if all its predecessors are expanded. At the end of expanding node i, we verify the label of i. We may nondeterministically *contract* any previous expansion if we wish; contraction is just the inverse of expansion. Of course we may contract a node only to re-expand it later. The space used by this procedure is $O(B)$ times the maximum number of expanded nodes at any moment. So to minimize space usage, we should contract nodes "at suitable moments". The graph G is said to be verified if its final node $\lceil t/B \rceil$ is verified in this process; we might as well assume that the label of $\lceil t/B \rceil$ always contains the accept state.

It is not hard to see that M accepts its input x iff there is a graph G that is verified by this procedure. We can make this procedure deterministic by cycling through all nondeterministic choices used in the expansion/contraction above. For a space-efficient method of verifying G, Hopcroft, Paul and Valiant showed a general strategy that never store more than $\frac{t}{B \log t}$ expanded nodes at any moment during the verification process. This means that the strategy never use more than $\frac{t}{\log t}$ space since each expanded node uses $O(B)$ space. This proves that $DTIME(t) \subseteq DSPACE(t/\log t)$. The details of this will not be explicitly described since it is essentially subsumed in the Tompa-Dymond alternating time implementation of the strategy, shown next.

7.8.2 A Pebble Game.

Now we transcribe the previous expansion and contraction process for verifying G into a combinatorial game on graphs. We are given a directed acyclic graph

$G = (V, E)$ together with a *goal* node $i_0 \in V$. There is only one player in this game. There is an infinite supply of indistinguishable pebbles and each node of G can hold a single pebble. A node is said to be *pebbled* if there is a pebble in it; it is *empty* otherwise. Initially, all the nodes are empty. A *pebbling step* consists of placing a pebble on an empty node u, provided all predecessors of u are already pebbled. In particular, we can always pebble an empty *source node* (i.e., a node with no predecessors). An *unpebbling step* consists of removing a pebble from a pebbled node. A *play* is simply a sequence of pebbling or unpebbling steps, with the last step being the pebbling of the goal node i_0. At any moment during the play, there is some number of pebbles on the graph, and our aim (as the player) is to choose the steps in a play in order to minimize the maximum number k of pebbled nodes at any time during the play. This number k is called *the pebble cost* of the play.

The reader will have no difficulty making the connection between this pebble game and the simulation described earlier: pebbling (unpebbling) a node corresponds to expansion (contraction) of nodes.

A *game strategy* is a rule to play the game for any graph. A trivial game strategy is to pebble the nodes in topological order and never to unpebble any nodes. On an n node graph, the pebble cost is n with this strategy. Can we do better in general? The key result here says: *for any directed acyclic graph G on n nodes with indegree at most d, the strategy yields a play with pebble cost $O_d(n/\log n)$.*

We want an 'alternating version' of playing this pebbling game. As usual, alternation turns the problem inside-out (or rather, bottom-up): instead of proceeding from the source nodes to the pebbling of the goal node i_0, we ask how can we pebble the goal node. This is viewed as a 'challenge' at node i_0. The challenge at a node u in turn spawns challenges at other nodes (which must include all predecessors of u). This is roughly the idea for our key definition:

Definition 13 *A pebbling tree for a directed acyclic graph G with goal node i_0 is a finite tree T satisfying the following.*[16] *Each vertex u of T is associated with a triple $[i, X, Y]$ where X and Y are subsets of nodes of G, and i is a node of G. We called i the challenged node, X the pebbling set, Y the unpebbling set (at vertex u). At each vertex u, define the set $C(u)$ of (currently) pebbled nodes at u by induction on the level of u: if u is the root then $C(u)$ is simply the pebbling set at u; otherwise if u is a child of u' then $C(u) = (C(u') - Y) \cup X$ where X (resp., Y) is the pebbling (resp., unpebbling) set at u. We require these properties:*

(i) *The challenged node at the root is the goal node i_0.*

(ii) *At each vertex u associated with $[i, X, Y]$, either $i \in X$ or else all the predecessors of i are contained in the currently pebbled nodes $C(u)$.*

(iii) *If the pebbling set at vertex u is X, then u has $|X|$ children, and the set comprising the challenged nodes at these children is precisely X.*

[16] To avoid confusing the nodes of T with those of G, we will refer to the nodes of T as 'vertices'.

■

Remark: note that (iii) implies that the pebbling set at a leaf must be empty; then (ii) implies that the predecessors of a challenged node at a leaf u is in $C(u)$.

Interpretation: This tree is an abstract description of an alternating computation tree that verifies the labels of a graph G in the sense of the Hopcroft-Paul-Valiant simulation of a deterministic time t machine M. To make this precise, we first describe an alternating machine N that on input a labeled graph G with goal node i_0 behaves as follows: initially, N existentially guesses some expansion of node i_0 and writes this onto tape 1; tape 2 is empty. In general, N is in the following 'inductive stage':

> Tape 1 contains the expansion $e(i')$ some node i',
> Tape 2 holds some number of expansions of nodes in G.

Then N existentially deletes some expansions in tape 2 and existentially writes some (possibly zero) new expansions in tape 3. If no expansion of node i' is found in tapes 2 and 3 then N tries to produce one: first N checks that all the predecessors of i' are in tapes 2 and 3 (otherwise it rejects) and then simulate M from sample time $t_{i'-1}$ to sample time $t_{i'}$ and, as usual, assume that we now have an expansion $d(i')$ of i'. N can now verify if the expansion $e(i')$ agrees with the found or newly constructed $d(i')$ (if not, N rejects). To continue, either tape 3 is empty (in which case N accepts) or else N universally chooses to copy one of the expanded nodes from tape 3 to tape 1 and the rest onto tape 2. Now we are back to the 'inductive stage'.

We claim that N accepts iff there is a pebbling tree T. Suppose T exists. To show that N accepts, we describe an accepting computation tree T' of N that is modeled after T: each inductive stage of N corresponds to a vertex u of T. If u is associated with the triple $[i, X, Y]$ then tape 1 contains the expansion of node i and tape 2 contains the expansions of the set $C(u)$ of pebbled nodes at u. Then the sets Y, X corresponds (respectively) to the expansions that are deleted from tape 2 or existentially guessed in tape 3. If we choose T' in such a way that the guessed expansions in tape 3 are always correct, then T' would be an accepting computation tree. Conversely, if N accepts, then we can construct the pebbling tree by reversing the above arguments. vertex of T. ■

With this interpretation, it is easy to understand the following definition of complexity. The *pebbling time* at any vertex u is the number of nodes in the union of the pebbling set and the unpebbling set. The pebbling time of a path of T is the sum of the pebbling times of nodes along the path. The *pebbling time* of T is the maximum pebbling time of any path in T. The *pebbling space* of T is the maximum of $|C(u)|$ over all vertices u. (Since we do not care about minimizing alternating space in our proofs, we can assume the unpebbling set Y is empty at each vertex u.)

Lemma 28 *For any directed acyclic graph G on n nodes with indegree at most d, there exists a pebbling tree with pebbling time of $O_d(n/\log n)$.*

Proof. We show that for any graph G and any goal node i_0, with indegree at most d and m edges, there is a pebbling tree for (G, i_0) whose pebbling time is at most $P(m)$, where $P(m) = O_d(m/\log m)$. Since $m \leq dn$ the lemma follows from this bound. We may suppose m is sufficiently large. First partition the nodes V of G into two disjoint sets, $V = V_a \cup V_b$ such that

(i) There are no edges from V_b ('nodes below') to V_a ('nodes above'). (So edges of G descend from above to below.) Let G_a, G_b be the induced subgraphs with nodes V_a, V_b, respectively.

(ii) The total indegree m_b of all nodes in V_b satisfies

$$\frac{m}{2} + \frac{m}{\log m} - d \leq m_b < \frac{m}{2} + \frac{m}{\log m}.$$

To see that such a partition exists, we offer a construction: starting with V_b as the empty set, keep putting nodes into V_b in topological order until the sum of indegrees of the nodes in V_b satisfies the above inequalities.

Let $A \subseteq V_a$ be the set of nodes that each has at least one successor in V_b. Consider the goal node i_0.

(iii) If $i_0 \in V_a$ then a pebbling tree for (G_a, i_0) is also a pebbling tree for (G, i_0). This tree has a pebbling time at most

$$P(m - m_b) \leq P(\frac{m}{2} - \frac{m}{\log m} + d).$$

(iv) Assume $i_0 \in V_b$ in the remaining cases. If $|A| < 2m/\log m$ then let the pebbling set at the root vertex be $A \cup \{i_0\}$. At each child u of the root vertex we inductively construct a pebbling tree according to two cases. If the challenged node at u is $i \in A$, then inductively construct a pebbling tree for (G_a, i) with pebbling time $P(m/2 - m/\log m + d)$. Otherwise the challenged node is i_0 and we inductively construct a pebbling tree for (G_b, i_0) with pebbling time $P(m/2 + m/\log m)$. It is easy to see that the result is a pebbling tree for (G, i_0) with pebbling time at most

$$\frac{2m}{\log m} + P(\frac{m}{2} + \frac{m}{\log m}).$$

(v) If $|A| \geq 2m/\log m$ we first construct a pebbling tree T for (G_b, i_0). The pebbling time for T is $P(m/2 - m/\log m)$ since there are at least $2m/\log m$ edges from V_a to V_b are not counted in G_b. We now convert T into a pebbling tree for (G, i_0). Let u be any leaf of T with challenged node i. The predecessors of i in G_b are contained in $C(u)$. Let $X(i)$ be the predecessors in G but not in G_b. We make $X(i)$ the pebbling set at u and create $|X(i)| \leq d$ children for

u. Each child u' of u has a challenged node $i' \in V_a$. We can attach to u' a pebbling tree T' for (G_a, i'). Observe that the pebbling time for T' is at most $P(m/2 - m/\log m + d)$. This completes our description of the pebbling tree for (G, i_0). The pebbling time of this tree is given by the pebbling time of T plus the pebbling time of T' plus at most d. This is at most

$$2P(\frac{m}{2} - \frac{m}{\log m} + d) + d.$$

Taking the worst of these three cases, we obtain

$$P(m) \leq \max\{P(\frac{m}{2} + \frac{m}{\log m}) + \frac{2m}{\log m}, 2P(\frac{m}{2} - \frac{m}{\log m} + d) + d\}$$

We want to show that there is a constant $c = c(d) \geq 5$ such that for m', $P(m') \leq cm'/\log m'$. By making c sufficiently large, we may assume that the truth has been established for m large enough. Inductively, we have the the following derivation:

$$\begin{aligned} P(\frac{m}{2} + \frac{m}{\log m}) + \frac{2m}{\log m} &= P(\alpha m) + \frac{2m}{\log m} \quad (\text{where } \alpha = \tfrac{1}{2} + \tfrac{1}{\log m}) \\ &\leq \frac{c\alpha m}{\log(\alpha m)} + \frac{2m}{\log m} \\ &\leq \frac{cm}{\log m}\left(\frac{\alpha \log m}{\log(\alpha m)} + \frac{2}{c}\right) \\ &\leq \frac{cm}{\log m}. \end{aligned}$$

We also obtain

$$\begin{aligned} 2P(\frac{m}{2} - \frac{m}{\log m} + d) + d &\leq \frac{2c\left(\frac{m}{2} - \frac{m}{\log m} + d\right)}{\log\left(\frac{m}{2} - \frac{m}{\log m} + d\right)} + d \\ &\leq \frac{cm\left(1 - \frac{2}{\log m} + \frac{2d}{m}\right)}{\log m + \log\left(\frac{1}{2} - \frac{1}{\log m} + \frac{d}{m}\right)} + d \\ &\leq \frac{cm\left(1 - \frac{1.9}{\log m}\right)}{\log m - 1.1} + d \\ &\leq \frac{cm}{\log m}\left(\frac{\log m}{\log m - 1.1}\right)\left(\frac{\log m - 1.9}{\log m}\right) + d \\ &\leq \frac{cm}{\log m}. \end{aligned}$$

Q.E.D.

Finally, we complete the proof of the main theorem by giving an alternating machine to simulate a deterministic M that accepts an input w in time t.

(1) Reserve tapes 1, 2 and 3 for later use. First we existentially choose the time t (tape 4) and blocking factor B (tape 5). Then we existentially choose a labeling of the graph G with nodes $V = \{0, \ldots, t/B\}$ (tape 6), and an edge set E (tape 7). Since each label uses $O(\log t)$ space, all this (when choice is correct) takes time $O(\frac{t \log t}{B})$.

(2) Universally choose to verify that E is correct relative to node labels, and to verify that the label of node t/B is correct. It takes time $O(\frac{t \log t}{B})$ to verify E. Verification of the label at node t/B is recursive as shown next.

(3) The verification of node $\lceil t/B \rceil$ amounts to simulating a pebbling tree T for $(G, \lceil t/B \rceil)$ (i.e., with $\lceil t/B \rceil$ as the goal node of G). We do this along the lines given by the "Interpretation" above. As before, each 'inductive stage' of our simulation of T corresponds to a vertex u of T: if $[i, X, Y]$ is associated with u then an expansion of the challenged node i is available on tape 1. The set of nodes previously expanded are available on tape 2. Since the pebbling time of T can be assumed to be $t/(B \log(t/B))$, we may assume that tape 2 has at most $t/(B \log(t/B))$ nodes. Since each expansion uses $O(B)$ space, tape 2 uses $O(t/\log(t/B))$ space. We existentially choose the pebbling set X at u and also their expansions, writing down these guesses on tape 3. (As noted before, we may assume Y is empty.) We then verify the challenged node i (it must either appear in tapes 2 or 3 or has all its predecessors expanded so that it can be simulated directly). This non-recursive verification of node i takes time $O(t/\log(t/B))$. To get to the next inductive stage, we universally choose to transfer one of the expanded nodes on tape 3 to tape 2, which is now the challenged node.

We have seen that the non-recursive parts of step (3) takes alternating time $O(t/\log(t/B))$. This time must be added to the total alternating time in the recursive parts of the computation. The recursive part of the computation, we claim is $O(B)$ times the pebbling time P. This is because each unit of pebbling time P can be associated with the guessing of an expanded node. But each expansion, when correct, takes space $O(B)$. It follows that the recursive part of the computation takes time $O(t/\log(t/B))$ since the pebbling time for the optimal tree is at most $O(t/[B \log(t/B)])$ (by preceding lemma, with $n = t/B$). Finally, if B is chosen to be $\log^2 t$, we get a time bound of $O(t/\log t)$ for steps (1),(2) and (3). (We could choose B as large as t^ϵ for any constant $\epsilon > 0$.) This proves our main theorem. **Q.E.D.**

In conclusion, it should be noted the space bound just obtained is the best possible in the sense that $\Omega(t/\log t)$ is a lower bound on the worst case pebbling time for the class bounded in-degree graphs [19].

7.9 Alternating Space

We show two results from Chandra, Kozen and Stockmeyer that relate alternating space and deterministic time.

Note that for a nondeterministic machine M, if there is an accepting path then there is one in which no configuration is repeated. The next lemma shows an analogous result for alternating machines.

Lemma 29
(a) Let M *be any choice machine. If T is an accepting computation tree for an input w then there is an accepting computation tree T' for w with the following properties:*

- *each computation path in T' is a (prefix of a) computation path in T*
- *if $u, v \in T'$ are vertices such that u is a proper ancestor of v and both u and v are labeled by the same configuration, then $Val_{T'}(v) \sqsubset Val_{T'}(u)$ (strict ordering).*

(b) If, in addition, M *is an alternating machine then we can assume that v is a leaf of T'.*

Proof. (a) The result follows if we show another accepting computation tree T' with fewer vertices. Let T_v denote the subtree of T rooted at v consisting of all descendents of v. There are two cases: if $Val_T(u) \sqsubseteq Val_T(v)$ then we can form T' from T by replacing T_u with T_v. By monotonicity, T' is still accepting.
(b) This simply follows from part (a) since for alternating machines, $Val_T(v) \sqsubset Val_T(u)$ implies that $Val_T(v) = \bot$. In that case, we might as well prune away all proper descendents of v from T.
Q.E.D.

Theorem 30 *For all complexity functions s,*

$$ASPACE(s) \subseteq DTIME(n^2 \log n O(1)^s).$$

Proof. Let M be an ordinary alternating machine accepting in space s. Later we indicate how to modify the proof if M has the addressable-input capability. We will construct a deterministic N that accepts $L(\mathrm{M})$ in the indicated time. On an arbitrary input w, N proceeds in stages: in the mth stage, N enumerates (in tape 1) the set Δ_m defined to be all the configurations C of M on input w where C uses at most m space. Note that each configuration can be stored in $m + \log n$ space, and there are $nO(1)^m$ configurations, so we use

$$(m + \log n) nO(1)^m = n \log n O(1)^m$$

space on tape 1. Then N enumerates (in tape 2) the edges of the computation tree T_m whose nodes have labels from Δ_m and where no node $u \in T_m$ repeats a configuration

that lie on the path from the root to u, except when u is a leaf. Clearly this latter property comes from the previous lemma. Using the information in tape 1, it is not hard to do this enumeration of edges in a 'top-down' fashion (we leave the details to the reader). Furthermore each edge can be produced in some constant number of scans of tape 1, using time $n \log nO(1)^m$. Thus the overall time to produce all $nO(1)^m$ edges is $n^2 \log nO(1)^m$. Now we can compute the least fixed point $Val_{T_m}(u)$ value at each node $u \in T_m$ in a bottom-up fashion, again $O(n \log nO(1)^m)$ per node for a total time of $n^2 \log nO(1)^m$. This completes our description of the mth stage.

The previous lemma shows that if M accepts w then at some mth stage, $m \leq s(|x|)$, T_m is accepting. Since the time for the mth stage is $n^2 \log nO_1(1)^m$, the overall time over all stages is

$$\sum_{m=1}^{s(n)} n^2 \log nO_1(1)^m = n^2 \log nO_2(1)^{s(n)}.$$

It remains to consider the case where M has the addressable-input capability. We first note that we never have to use more than $O(\log n)$ space to model the address tape (if M writes more than $\log n$ bits, we ignore the tape from that point onwards since it will lead to error when a READ is attempted). Hence the above space bounds for storing a configuration holds. Furthermore, the time to generate the contents of tapes 1 and 2 remains asymptotically unchanged. Similarly for computing the least fixed point Val_{T_m}. This concludes the proof. **Q.E.D.**

Theorem 31 *For all $t(n) > n$, $DTIME(t) \subseteq ASPACE(\log t)$.*

Proof. Let M accept in deterministic time t. We describe an alternating machine N to accept $L(\mathrm{M})$ in space $\log t$. For this simulation, N can be the ordinary variety of alternating machine. We may assume that M is a simple Turing machine and M never moves its tape head to the left of its initial position throughout the computation. (Otherwise, we first convert M into a simple Turing machine accepting in time $t(n)^2$ with these properties. How?) Let x be any word accepted by M. Let $C_0, C_1, \ldots, C_m$, $m = t(|x|)$, be the unique accepting computation path of M on x. We assume that the final accepting configuration is repeated as many times as needed in this path. Let each C_i be encoded as a string of length $m + 2$ over the alphabet

$$\Gamma = \Sigma \cup [Q \times \Sigma] \cup \{\Box\}$$

where Σ is the tape alphabet of M, Q the state set of M, and $[Q \times \Sigma]$ is the usual composite alphabet. Furthermore, we may assume the the first and last symbol of the string is the blank symbol $\Box$. Let $\alpha_{i,j}$ denote the jth symbol in configuration C_i $(i = 0, \ldots, m; j = 1, \ldots, m + 2)$.

N will be calling a subroutine $CHECK(i, j, b)$ that verifies whether $\alpha_{i,j} = b$ where $b \in \Gamma$. N begins its overall computation by existentially choosing the integer

m, the head position h ($1 \leq h \leq m+2$) and a symbol $b' = [q_a, c]$ and then it calls $CHECK(m, h, b')$. The subroutine is thus verifying that that M is scanning the symbol c at position h when M enters the accept state q_a. All integer arguments are in binary notation.

In general, the subroutine to verify if $\alpha_{i,j} = b$ (for any i, j, b) operates as follows: if $i = 0$ or $j = 1$ or $j = m$, N can directly do the checking and accept or reject accordingly. Otherwise, it existentially chooses the symbols b_{-1}, b_0, b_{+1} such that whenever b_{-1}, b_0, b_{+1} are consecutive symbols in some configuration of M then in the next instant, b_0 becomes b. Now N universally chooses to call

$$CHECK(i-1, j-1, b_{-1}), CHECK(i-1, j, b_0), CHECK(i-1, j+1, b_{+1}).$$

It is important to realize that even if b does not contain the tape head (i.e., $b \notin [Q \times \Sigma \times I]$), it is possible for b_{-1} or b_{+1} to contain the tape head. The space used by N is $O(\log m)$.

Correctness. If M accepts then it is clear that N accepts. The converse is not obvious. To illustrate the subtlety, suppose $CHECK(i, j, b)$ accepts because $CHECK(i-1, j-\epsilon, b_\epsilon)$ accepts for $\epsilon = -1, 0, 1$. In turn, $CHECK(i-1, j-1, b_{-1})$ accepts because $CHECK(i-2, j-1, b')$ (among other calls) accepts for some b'; similarly $CHECK(i-1, j, b_0)$ accepts because $CHECK(i-2, j-1, b'')$ (among other calls) accepts for some b''. But there is no guarantee that $b' = b''$ since these two calls occur on different branches of the computation tree. Another source of inconsistency is that the existential guessing of the b_j's may cause more than one tape head to appear during one configuration. Nevertheless, we have a correctness proof that goes as follows. First observe that if $CHECK(i, j, b)$ is correct if $i = 0$. Moreover, given j there is a unique b that makes $CHECK(0, j, b)$ accept. Inductively, assume $CHECK(i, j, b)$ is correct for all j, b and that $CHECK(i, j, b)$ and $CHECK(i, j, b')$ accept implies $b = b'$. Then it is easy to see that $CHECK(i+1, j, b)$ must be correct for all j, b; moreover, because of determinism, $CHECK(i+1, j, b)$ and $CHECK(i+1, j, b')$ accept implies $b = b'$. [This is why *CHECK* must universally call itself three times: for instance, if *CHECK* only makes two of the three calls, then the correctness of these two calls does not imply the correctness of the parent.] So we have shown that the symbols $\alpha_{i,j}$ are uniquely determined. In particular $\alpha_{m,h} = b'$ in the initial guess is correct when the machine accepts. **Q.E.D.**

The reader should see that this proof breaks down if we attempt to simulate nondeterministic machines instead of deterministic ones.

Combining the two theorems yields the somewhat surprising result:

Corollary 32 *For $s(n) \geq \log n$, $ASPACE(s) = DTIME(O(1)^s)$.*

7.10 Final Remarks

This chapter introduced choice machines to provide a uniform framework for most of the choice modes of computation. It is clear that we can extend the basic framework

to other value sets S (analogous to the role of *INT*) provided S is equipped with a partial order $\sqsubseteq$ such that limits of $\sqsubseteq$-monotonic chains are in S and S has a $\sqsubseteq$-least element (such an S is called a *complete partial order*). The reader familiar with Scott's theory of semantics will see many similarities. For relevant material, see [22].

We have a precise relationship between alternating space and determistic time: for $s(n) \geq \log n$,

$$ASPACE(s) = DTIME(O(1)^s). \tag{7.8}$$

What is the precise relationship between alternating time and deterministic space? Although we have tried to emphasize that alternating time and deterministic space are intimately related, they are not identical. We know that

$$ATIME(s) \subseteq DSPACE(s) \subseteq NSPACE(s) \subseteq ATIME(s^2). \tag{7.9}$$

for $s(n) \geq \log n$. How 'tight' is this sequence? It is unlikely that that the first two inclusions could be improved in the near future.

From (7.8) and (7.9), we get find new characterizations of some classes in the canonical list:

$$\begin{aligned} P &= ASPACE(\log n) \\ PSPACE &= ATIME(n^{O(1)}) \\ DEXPT &= ASPACE(n) \\ EXPS &= ATIME(O(1)^n). \end{aligned}$$

What is the relationship of alternating reversal with deterministic complexity? Of course, for alternating machines, we must take care to simultaneously bound reversal with either time or space in order to get meaningful results. Other complexity measures for alternating machines have been studied. Ruzzo [20] studied the *size* (i.e., the number of nodes) of computation trees. In particular, he shows

$$A\text{-}SPACE\text{-}SIZE(s(n), z(n)) \subseteq ATIME(s(n) \log z(n)).$$

King [13] introduced other measures on computation trees: *branching* (i.e., the number of leaves), *width* (below), *visit* (the maximum number of nodes at any level). Width is not so easy to motivate but in the special case of binary trees (which is all we need for alternating machines), it is the minimum number of pebbles necessary to pebble the root of the tree. Among the results, he shows (for $s(n) \geq \log n$),

$$\begin{aligned} A\text{-}SPACE\text{-}WIDTH(s(n), w(n)) &= NSPACE(s(n)w(n)), \\ A\text{-}SPACE\text{-}VISIT(s(n), v(n)) &\subseteq ATIME(s^2(n)v(n)). \end{aligned}$$

Exercises

[7.1] Verify the identities in section 2 on interval algebra.

[7.2] Some additional properties of the lattice *INT*:
(a) Show that the two orderings $\leq$ and $\sqsubseteq$ are 'complementary' in the following sense: for all I and J, either I and J are $\leq$-comparable or they are $\sqsubseteq$-comparable.
(b) Show that I and J are both $\leq$-comparable and $\sqsubseteq$-comparable iff $I \approx J$ where we write $[x, y] \approx [x, v]$ if $x = u$ or $y = v$.
(c) Extend $\wedge$ and $\vee$ to arbitrary sets $S \subseteq INT$ of intervals: denote the meet and join of S by $\wedge S$ and $\vee S$. Show that *INT* forms a complete lattice with least element 0 and greatest element 1.

[7.3] Consider the 2-ary Boolean function *inequivalence* (also known as *exclusive-or*) denoted $\not\equiv$. We want to extend this to a function on intervals in *INT*. One way to do this is to use the equation

$$x \not\equiv y = (x \wedge \neg y) \vee (\neg x \wedge y)$$

valid for Boolean values, but now interpreting the $\wedge, \vee$ and $\neg$ as functions on intervals. For instance,

$$\begin{aligned} [0.2, 0.5] \not\equiv [0.6, 0.7] &= ([0.2, 0.5] \wedge [0.3, 0.4]) \vee ([0.5, 0.8] \wedge [0.6, 0.7]) \\ &= [0.2, 0.4] \vee [0.5, 0.7] = [0.5, 0.7]. \end{aligned}$$

Can you find other equations for $\not\equiv$ that are valid in the Boolean case such that the extension to intervals are not the same function?

[7.4] * (i) Consider generalized probabilistic machines in which we allow k-ary versions of the coin-tossing function, $\oplus_k$ for all $k \geq 2$. Show that these can be simulated by ordinary probabilistic machines with at most a constant factor slow-down in time.
(ii) Show the same result for stochastic machines where we now allow $\oplus_k, \oplus_k, \otimes_k$ for all k.

[7.5] (Hong) Consider basis sets B that are subsets of the 16 Boolean functions on 2 variables. As usual, we assume that the identity, 0 and 1 constant functions are not explicitly mentioned when we display B. For two bases B, B', say that B *linearly simulates* B' if for every B'-choice machine M′, there is a B-machine M accepting the same language such that if M′ accepts in time $t(n)$ then M accepts in time $O_{B,B'}(t)$. Say B and B' are *linearly equivalent* if they can linearly simulate each other.
(a) Prove that every such basis set B is linearly equivalent to one of the following 5 bases:

$$B_0 := \emptyset, B_1 := \{\vee\}, B_2 := \{\wedge\}, B_3 := \{\overline{\oplus}\}, B_4 := \{\vee, \wedge\}$$

where $\overline{\oplus}$ is the exclusive-or function.
(b) By simple set inclusions, it is clear that B_0 can be linearly simulated by the others and B_4 can linearly simulate B_1 and B_2. Show that B_4 can also linearly simulate B_3. *Hint:* Use the same idea as the proof for elimination of negation.
(c)** Show that these 5 classes are distinct up to linear equivalence. (Note: it is known that B_1 is distinct from B_0.)

[7.6] ** Let $B = \{\vee, \overline{\oplus}, \neg\}$. Can negation be eliminated from B-machines operating in polynomial time?

[7.7] Generalize choice machines by allowing values from any $\sqsubseteq$ -partially ordered set F that has a $\sqsubseteq$ -mimimal element $\perp \in F$ and such that any $\sqsubseteq$ -monotonic non-decreasing sequence has a least upper bound in F. In particular, let F be the Borel sets (obtain from *INT* under the operation of intersection, difference and countable unions).

[7.8] * Extend valuation theory for acceptors to transducers.

[7.9] (Paul, Praus, Reischuk) Construct a simple (i.e., one work-tape and no input tape) alternating machine that accepts palindromes in linear time and a constant number of alternation. *Hint:* Guess the positions (in binary) of about $\log n$ equally spaced-out input symbols, writing these directly under the corresponding symbols. The positions of all the other symbols can be determined relative to these guessed positions.

[7.10] (Paul, Praus, Reischuk) Show that if a language can be accepted by an alternating Turing machine in time $t(n)$ then it can be accepted by a simple alternating Turing machine in time $O(t(n))$. *Hint:* For a computation path $C_1 \vdash C_2 \vdash, \ldots, \vdash C_m$, let its *trace* be $\tau_1, \tau_2, \ldots, \tau_m$ where τ_i contains the state, the symbols scanned on each of the tapes (including the input tape) and the direction of each tape head in the transition $C_i \vdash C_{i+1}$. The head directions are undefined for τ_m. (Note: this definition of trace does not include head positions, in contrast to a similar definition in chapter 2.) Begin by guessing the trace of the paths in an accepting computation tree – you must use universal and existential guessing corresponding to the state in τ_i. To verify correctness of the guess, the technique for the previous question is useful.

[7.11] (Paterson, Paul, Praus, Reischuk) For all $t(n)$, if a language is accepted by a nondeterministic simple Turing machine in time $t(n)$ then it can be accepted by an alternating machine in time $n + t^{1/2}(n)$.

[7.12] Show the tight complexity relationships between the ordinary SAM's and the addressable-input version of SAM's. More precisely, give efficient time/space/reversal simulations of the addressable-input machines by ordinary machines.

[7.13] Rederive the various simulation of SAM's in this chapter in the case where the SAMs is the addressable-input model. In particular, show that $ATIME(t) \subseteq DSPACE(t)$.

[7.14] Obtain a lower bound on the space-time product of alternating Turing machines which accepts the palindrome language L_{pal}.

[7.15] * Obtain the tight bound of $\Theta(\log n)$ on the alternating time complexity for the multiplication problem define as follows:

$$L_{mult} = \{x\#y\#z\# : x, y, z \in \{0,1\}^*, x \cdot y = z\}.$$

Use the addressable-input model of machine.

[7.16] ** A very natural function that one would like to add to our basis sets is the cut-off function $\delta_{\frac{1}{2}}$ defined at the end of section 2. Note that it gives us a model of oracle calls in which the complexity of the oracle machine is taken into account. Of course, this is not continuous: extend the theory of valuations to allow monotonic, piece-wise continuous functions. (A functions is piece-wise continuous if it has a finite number of discontinuities.)

[7.17] Explore the power of choice machines: suppose the basis functions are rational, linear convex combinations of their arguments: more precisely, the valuation functions f have the form

$$f(x_1, \ldots, x_n) = \sum_{i=1}^{n} a_i x_i$$

where the a_i's are positive rational numbers depending on f and $\sum_i a_i = 1$. How do these machines compare to SAMs?

[7.18] Give a sufficient condition on a family F of complexity functions such that $ATIME(F) = PrA\text{-}TIME(F) = DSPACE(F)$. *Hint:* Consider the case $F = n^{O(1)}$.

[7.19] * Can the alternating version of I. Simon simulation (section 6) be improved? In particular, try to improve $DREVERSAL(r) \subseteq ATIME(r^3)$.

[7.20] In section 7, we saw a reduction of Turing machine simulation to graph pebbling, due to Hopcroft-Valiant-Paul. An alternative definition of the edge set E is this: "define (j, i) to be an edge of G if $j < i$ and there is some block b visited in the ith time interval and last visited in the jth time interval." Using this definition of G, what modifications are needed in the proof showing $DTIME(t) \subseteq DSPACE(t/\log t)$?

[7.21] What is the number of alternations in the proof of $DTIME(t) \subseteq ATIME(t/\log t)$? Can this be improved?

[7.22] Show that if $t(n)$ is time constructible then *co-ATIME*$(t(n)) \subseteq$ *ATIME*$(n+t(n))$. HINT: why do you need the "$n+$" term?
NOTE: For instance, if $t(n) \geq 2n$, *ATIME*$(n + t(n))$ = *ATIME*$(t(n))$, and so *ATIME*$(t(n))$ is closed under complementation. This is essentially the result of Paul and Reischuk.

[7.23] * An probabilistic-alternating finite automaton (f.a.) is simply a PAM with no work-tape. The restriction to purely alternating or to purely probabilistic f.a. is clear. We further restrict its input tape to be one-way.
(i) (Chandra-Kozen-Stockmeyer) Prove that the alternating f.a. accepts only regular languages.
(ii) (Starke) Show the following language (well-known to be non-regular) $L = \{0^m 1^n : m \geq n\}$ can be accepted by a probabilistic f.a..

[7.24] (Freivalds) Show a probabilistic finite automata to recognize the language $\{0^n 1^n : n \geq 1\}$ with bounded error. *Hint:* (B. Mishra) We are basically trying to check that the number of 1's and number of 0's are equal. Show that the following procedure works:
a. Choose a coin with probability $p << \frac{1}{2}$ of getting a *head*.
b. Toss coin for each 0 in input and each 1 in input. If we get all *heads* for 0's and at least one *tail* for the 1's then we say we have a 0-*win*. If we get at least *tail* for the 0's and all *heads* for the 1's, we have a 1-*win*. All other outcomes result in a *tie*.
c. Repeat this experiment until we have at least D 0-wins or D 1-wins. We accept if and only if there is at least one 1-win and at least one 0-win.
(For more information, see [10].)

[7.25] (Ruzzo, King) Show the following simulations among measures for alternating computation, as stated in the concluding section: for $s(n) \geq \log n$,

$$\begin{aligned} \textit{A-SPACE-SIZE}(s(n), z(n)) &\subseteq \textit{ATIME}(s(n) \log z(n)), \\ \textit{A-SPACE-WIDTH}(s(n), w(n)) &= \textit{NSPACE}(s(n)w(n)), \\ \textit{A-SPACE-VISIT}(s(n), v(n)) &\subseteq \textit{ATIME}(s^2(n)v(n)). \end{aligned}$$

[7.26] (King) Recall the definitions of branching and width resource for alternating machines in the concluding section. Show that the branching resource (simultaneously bounded with space) has the linear complexity reduction property: for $s(n) \geq \log n$,

$$\textit{A-SPACE-BRANCH}(s(n), b(n)) = \textit{A-SPACE-BRANCH}(s(n), b(n)/2).$$

Show the same result for width resource: for $s(n) \geq \log n$,

$$\textit{A-SPACE-WIDTH}(s(n), w(n)) = \textit{A-SPACE-WIDTH}(s(n), w(n)/2).$$

[7.27] Show that if a graph with goal node (G, i_0) has a pebbling tree with pebbling time t then it can be pebbled with t pebbles. Is the converse true?

[7.28] (Paul, Tarjan, Celoni, Cook)
(a) A *level graph* is a directed acyclic graph with bounded in-degree such that the vertices can be partitioned into 'levels' and edges only go from level i to level $i+1$. Show that every level graph on n vertices can be pebbled using $O(\sqrt{n})$ pebbles.
(b) Show an infinite family of graphs with indegree 2 that requires $\Omega(\sqrt{n})$ pebbles to pebble certain vertices.

Bibliography

[1] L. Adleman and M. Loui. Space-bounded simulation of multitape Turing machines. *Math. Systems Theory*, 14:215–222, 1981.

[2] László Babai and Shlomo Moran. Arthur-Merlin games: a randomized proof system, and a hierarchy of complexity classes. *Journal of Computers and Systems Sciences*, 36:254–276, 1988.

[3] A. Chandra, D. Kozen, and L. Stockmeyer. Alternation. *J. ACM*, 28:1:114–133, 1981.

[4] A. Condon and R. Ladner. Probabilistic game automata. *Journal of Computers and Systems Sciences*, 36:452–489, 1988.

[5] Peter Crawley and Robert Dilworth. *Algebraic theory of lattices.* Prentice-Hall, 1973.

[6] P. Dymond and M. Tompa. Speedups of deterministic machines by synchronous parallel machines. *Journal of Computers and Systems Sciences*, 30:149–161, 1985.

[7] J. T. Gill. Computational complexity of probabilistic Turing machines. *SIAM J. Comp.*, 6(4):675–695, 1977.

[8] S. Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proofs. *17th ACM Symposium STOC*, pages 291–304, 1985.

[9] Shafi Goldwasser and Michael Sipser. Private coins versus public coins in interactive proof systems. *18th ACM Symposium STOC*, pages 59–68, 1986.

[10] Albert G. Greenberg and Alan Weiss. A lower bound for probabilistic algorithms for finite state machines. *Journal of Computer and System Sciences*, 33:88–105, 1986.

[11] Jia-wei Hong. *Computation: Computability, Similarity and Duality.* Research notices in theoretical Computer Science. Pitman Publishing Ltd., London, 1986. (available from John Wiley & Sons, New York).

[12] J. E. Hopcroft, W. J. Paul, and L. G. Valiant. On time versus space. *Journal of Algorithms*, 24:332–337, 1977.

[13] Kimberley N. King. Measures of parallelism in alternating computation trees. *ACM Symp. on Theory of Computing*, 13:189–201, 1981.

[14] Burkhard Monien and Ivan Hal Sudborough. On eliminating nondeterminism from Turing machines which use less than logarithm worktape space. In *Lecture Notes in Computer Science*, volume 71, pages 431–445, Berlin, 1979. Springer-Verlag. Proc. Symposium on Automata, Languages and Programming.

[15] R. E. Moore. *Interval Analysis*. Prentice-Hall, Englewood Cliffs, N.J., 1966.

[16] Christos H. Papadimitriou. Games against nature. *Journal of Computers and Systems Sciences*, 31:288–301, 1985.

[17] Michael S. Paterson. Tape bounds for time-bounded Turing machines. *Journal of Computers and Systems Sciences*, 6:116–124, 1972.

[18] W. J. Paul, Ernst J. Praus, and Rüdiger Reischuk. On alternation. *Acta Informatica*, 14:243–255, 1980.

[19] W. J. Paul, R. E. Tarjan, and J. R. Celoni. Space bounds for a game on graphs. *Math. Systems Theory*, 11:239–251, 1977. (See corrections in Math. Systems Theory, 11(1977)85.).

[20] Walter L. Ruzzo. Tree-size bounded alternation. *ACM Symp. on Theory of Computing*, 11:352–359, 1979.

[21] Martin Tompa. An improvement on the extension of Savitch's theorem to small space bounds. Technical Report Technical Report No. 79-12-01, Department of Computer Sci., Univ. of Washington, 1979.

[22] Klaus Weihrauch. *Computability*. Springer-Verlag, Berlin, 1987.

[23] Chee K. Yap. On combining probabilistic and alternating machines. Technical report, Univ. of Southern California, Comp. Sci. Dept., January 1980. Technical Report.

Contents

Chapter 8

Stochastic Choices

September 29, 1998

8.1 Errors in Stochastic Computation

We continue investigating the choice-mode of computation. This chapter focuses on the stochastic choices, viz., coin-tossing $\oplus$, probabilistic-and $\otimes$ and probabilistic-or $\oplus$. For convenience, an appendix on the basic probabilistic vocabulary is included. Two interesting new phenomena arise with stochastic choices:

- Infinite loops in stochastic computation is an essential feature, rather than one we seek to eliminate (cf. alternation). This will be evident when we study space-bounded computations in section 3.
- An extremely rich theory arises from quantifying the forms of computational error. We think of error as a new computational resource.

  On forms of computational error. In this chapter, we revert to the use of intervals $I \in INT$ when discussing valuations. Let M be a choice machine and suppose $Val_{\mathrm{M}}(w) = [b, c]$ where w is an input word. If M accepts w (*i.e.*, $b > \frac{1}{2}$) then both $1-b$ and $1-c$ are useful measures of error. Since $\frac{1}{2} > 1-b \geq 1-c \geq 0$, we may call $1-b$ and $1-c$ (respectively) the **pessimistic acceptance error** and **optimistic acceptance error**. Similarly, if M rejects w, then b and c are (respectively) the **optimistic rejection error** and **pessimistic rejection error**. There are two basic paradigms for classifing errors: for optimistic errors, we say a choice machine has "zero error" if for all inputs, its optimistic error is 0. For pessimistic errors, we say a choice machine has "bounded-error" if its pessimistic errors are bounded away from $\frac{1}{2}$ by a positive constant.

Stochastic computation has been studied since the early days of automata theory [9]. One original motivation (and this line of work is still actively pursued) is the fundamental question of synthesizing reliable components from unreliable ones [21].

Stochastic computation in complexity theory started with the work of Gill [12]. Gill was in turn motivated by some surprising probabilistic algorithms for primality testing algorithms due to Rabin [22] and Solovay-Strassen [29].

Example 1 The said primality testing algorithms have this property: every computation path terminates and at each terminal configuration C,

1) if C answers YES (i.e., claims that the input is prime) then there is a "small probability" that it made an error;

2) if C answers NO (i.e., claims that the input is composite) then it is surely correct.

The algorithm does not have YO-answers. It is not so easy to quantify the above "small probability". But we may rephrase this property, taking the viewpoint of the inputs:

1′) if the input is prime then all local answers are YES;

2′) if the input is composite then the local answer is NO with "high probability".

The reader should verify that 1′) and 2′) (the global perspective) are just reformulations of 1) and 2) (the local perspective). We elaborate on this because it is easy to become confused by a mixture of these two perspectives. The global formulation is more useful because we can explicitly quantify the "high probability" in it: it means with probability more than 3/4. We see that these primality algorithms make no errors whenever they (globally) accept.

> To remember which way the primality algorithms may have errors, it is helpful to know this: the algorithms answers NO (*i.e.*, claims that an input is composite) only if it finds a 'witness' of compositeness. Hence NO-answers are never wrong by virtue of such witnesses. In contrast, the algorithm answers YES (*i.e.*, claims an input is prime) based on its failure to find a witness – such a YES answer could be wrong because an individual path does not exhaustively search for witnesses. In fact the algorithm looks for witnesses by randomly picking candidate witnesses and then checking each for the witness property. An example of a 'witness' of the compositeness of a number n is a factor m $(1 < m < n)$ of n. It is easy to check if any proposed witness m is a factor. Unfortunately, there may be as few as two witnesses for some composite numbers; in this case, the method of randomly selecting candidate witnesses for testing is unreliable. The algorithms of Rabin and of Solovay-Strassen solves this by using more sophisticated concepts of "witnesses" where it can be shown that each composite number has a positive fraction of witnesses among the candidates, and a prime number has no witness.

This example will be used again to illustrate error concepts. ■

We now classify Turing machines according to their error properties.

Definition 1
(i) A non-empty interval g containing the value $\frac{1}{2}$ is called an **error gap**. *Thus g has one of the forms*

$$[a,b], \quad (a,b], \quad [a,b), \quad (a,b)$$

where $0 \leq a \leq \frac{1}{2} \leq b \leq 1$. We call a and b (respectively) the **lower** *and* **upper bounds** *of g. (ii) A choice machine* M **accepts with error gap** *g if for all accepted inputs w,*

$$Val_{\mathrm{M}}(w) \cap g = \emptyset.$$

Nothing is assumed if w is rejected or undecided. Similarly, we define what it means to **reject with error gap** *g.*
(iii) M **has error gap** *g if for all inputs w, $Val_{\mathrm{M}}(w) \cap G(|w|) = \emptyset$. (Thus* M *is decisive.)*
(iv) We say M **accepts with bounded-error** *if there exists e ($0 < e \leq \frac{1}{2}$ such that for all accepted inputs w, the lower bound of $Val_{\mathrm{M}}(w)$ is $\geq \frac{1}{2} + e$. Similarly, we say* M **rejects with bounded-error** *if for all rejected inputs w, the upper bound of $Val_{\mathrm{M}}(w)$ is $\leq \frac{1}{2} - e$. Also,* M *has* **bounded-error** *if it is decisive, and it accepts and rejects with bounded-error. Say* M *has* **unbounded-error** *if it does not have have bounded-error.* ■

An **error gap function** G is a total function that assigns an error gap $G(n)$ to each natural number n. We can generalize the above definitions by replacing g by an error gap function $G(n)$.

Definition 2 *(Continued)*
(v) We say M **accepts a** *w* **with zero-error** *if it accepts w and the upper bound of $Val_{\mathrm{M}}(w)$ is 1. We say* M *has* **zero-error acceptance** *if every accepted word w is accepted with zero-error.*
Similarly, M **rejects a** *w* **with zero-error** *if it rejects w and the lower bound of $Val_{\mathrm{M}}(w)$ is 0. We say* M *has* **zero-error rejection** *if every rejected word w is rejected with zero-error.*[1]
(vi) We say M *has* **zero-error** *if it is decisive, and it has zero-error acceptance and zero-error rejection.*
(vii) We combine bounded-error and zero-error: M *has* **bounded zero-error rejection** *if it has bounded-error and zero-error rejection. Bounded zero-error rejection is also called* **one-sided error**. *By symmetry, we say* M *has* **bounded zero-error acceptance** *it has bounded-error and zero-error acceptance. Finally,* M *has* **bounded zero-error** *if it has bounded-error and zero-error.* ■

[1]We emphasize that 'zero-error' does not imply the absence of all errors: it only refers to the optimistic errors. Furthermore, it seems that "errorless" would be a preferable substitute for "zero-error" in this set of terminology, except that zero-error is quite well accepted.

We briefly discuss the significance of these forms of error and their motivations.[2]

Although our error concepts treat acceptance and rejection with an even hand, we still favor acceptance when it comes to defining languages and complexity classes: for any machine M, the notation

$$L(\mathrm{M})$$

continues to refer to the language **accepted** by M. So a word $w \notin L(\mathrm{M})$ is either rejected or undecided by M.

Bounded Errors. Note that undecided intervals in INT are error gaps. Clearly a stochastic machine has an error gap function if and only it has the minimal error gap $[\frac{1}{2}, \frac{1}{2}]$, if and only if it is decisive. In this sense, bounded-error is a strengthening of halting computation. To understand the significance of 'bounded-error', note that in general, acceptance and rejection errors can get arbitrarily close to $\frac{1}{2}$. This behavior is forbidden by bounded-error. The next section shows that with bounded-error we can modify a machine to yield error gaps of the form $[\epsilon, 1-\epsilon]$ for any desired constant $0 < \epsilon < \frac{1}{2}$, at a cost of increasing the computational time by a constant factor depending on ϵ. This yields a very important conclusion: *assuming we can tolerate constant factor slowdowns, bounded-error algorithms are practically as good as deterministic algorithms.*

Nondeterministic vs. probabilistic machines. Let N be a nondeterministic machine. The valuation function Val_{N} has values in $\{0, 1, \perp\}$. It follows that N has zero-error and accepts with no pessimistic error. Next, let us see what happens when N is converted into a probabilistic machine M, simply by declaring each state a toss-state. If N does not accept an input, M also does not accept. Hence we have

$$L(\mathrm{M}) \subseteq L(\mathrm{N}).$$

A further modification to M yields a probabilistic machine M$'$ that accepts the same language as N: let M$'$ begin by tossing a coin in the start state, and on one outcome it immediately answers YES, and with the other outcome it simulates N. So M$'$ is undecided on input $w \notin L(\mathrm{N})$, since the lower bound of $Val_{\mathrm{M}'}(w)$ is exactly $\frac{1}{2}$. This shows

$$NTIME(t) \subseteq PrTIME(t+1). \tag{8.1}$$

See the definition of $PrTIME(t)$ below. Also, M' has zero-error rejection. A trivial modification to M' will ensure that it has zero-error acceptance as well: simply

[2]The word 'error' in the terminology 'error gap' and 'bounded-error' refers to pessimistic errors. On the other hand, 'error' in 'zero-error' refers to optimistic errors. Thus the concepts of zero-error and bounded-error are independent. This double use of the term 'error' is regrettable, but it is a reasonably established convention. We feel justified to propagate this usage because each instance of such error terms will be accompanied by key words such as 'bounded', 'gap' or 'zero' which unambiguously indicate the form of error.

convert all terminating configurations into looping configurations. Notice that N, M and M′ are not decisive. If N were unequivocal in the sense of chapter 2 (§9) then both N and M would be decisive (but what could be done to make M′ decisive?).

Zero-error computation. The primality testing algorithms above accepts with zero-error. The concept of zero-error is best understood in terms of stochastic machines with no negation states: in this case acceptance with zero-error means that if an input is accepted then no computation path leads to a NO-configuration. Similarly, 'rejecting with zero-error' means that if an input is rejected, then no computation path leads to a YES-configuration. In either case, YO-configuration (or looping) is not precluded. Because of monotonicity properties, if a complete computation tree $T_{\mathrm{M}}(w)$ accepts with zero-error then any prefix T' of $T_{\mathrm{M}}(w)$ also accepts with zero-error, if T' accepts at all.

One-sided error. One-sided error is also motivated by the primality algorithms. These algorithms have no pessimistic errors on prime inputs, by virtue of property 1'), and have bounded error on composite inputs, by property 2'). Thus such an algorithm has bounded zero-error acceptance. Now with a trivial change, we can regard the same algorithm as a recognizer of composite numbers: it answers YES iff it finds a witness of compositeness. This new algorithm has bounded zero-error rejection, i.e., it has one-sided error.

This begs the question as to why we define one-sided error to favor "zero-error rejection" over "zero-error acceptance"? We suggest that the (non-mathematical) reason has to do with our bias towards nondeterministic machines: *a probabilistic machine* M *with one-sided error can be regarded as a nondeterministic machine* N. Let us clarify this remark. For, if M accepts w then some terminal configuration gives a YES-answer, and so N accepts w. Conversely, if M does not accept w, then it rejects w. Since M has zero-error rejection, this mean there is no YES-configuration. Hence, N does not accept w. We conclude that $L(\mathrm{M}) = L(\mathrm{N})$. This proves

$$PrTIME_1(t) \subseteq NTIME(t). \tag{8.2}$$

The subscript '1' in $PrTIME_1(t)$ refers to one-sided error (see notation below).

One final remark. The literature often discuss errors in the context of probabilistic machines that run in time $t(n)$ for some time-constructible t (e.g., t is a nice polynomial). Under such circumstances, we could simplify many of the concepts here. For instance, we need not deal with intervals: if a machine does not terminate within t steps, we simply answer NO. This avoids the value $\perp$ and more important, the pessimistic and optimistic errors coincide. Unfortunately this simplification is not so readily available in general (for instance, in space-bounded computations).

Complexity. Since error is viewed as another computational resource, we combine error with other resource bounds. The number of logical possibilities is large, but

happily, only a few forms are important. For instance, we exclusively use constant gap functions in complexity characteristics.

The following illustrates the combination of acceptance time with bounded-error or with zero-error:

Definition 3
Let t be a complexity function.
(i) A choice machine M **accepts in time** t **with bounded-error** *if there is an error gap $g = [a, b]$, $a < \frac{1}{2} < b$ such that for all accepted inputs w, there is an accepting computation tree T such that $Val_T(w) \cap g = \emptyset$ and each path in T has length at most $t(|w|)$.*
(ii) M **accepts in time** t **with zero-error** *if it accepts in time t and it has zero-error acceptance.* ■

In (i), the time bound and error gap are simultaneously achieved in a single computation tree T. In general, all complexity characteristics we specify for a single machine are assume to be simultaneous unless otherwise stated. But in some cases, simultaneity and non-simultaneity are the same. Such is the case with (ii) above: zero-error acceptance is a property of the complete computation tree, independent of acceptance in time t.

We can also combine all three: time bound, bounded-error and zero-error. The definition generalizes in the usual way to "rejection complexity" and to "running complexity". For rejection complexity, just replace acceptance by rejection. For running complexity, we combine accepting complexity, rejecting complexity and decisiveness.

There is another combination of errors that is important:

Definition 4 *We say* M *has* **runs in time** t **with one-sided error** *if it runs in time t with bounded-error and zero-error rejection.* ■

Note that the zero-error requirement is on the rejection side. A complement to this definition would be "running in time t with bounded-error and zero-error acceptance. However, this does not have a special name.

We naturally extend the above definitions to other resources such as space or reversals, including simultaneous resources.

Notation for error-restricted complexity classes. For the error-restricted classes, for simplicity, we will always the machine is decisive and running (time, space, etc) complexity is used. To refer to such classes, it is sufficient to augment our previous convention for complexity classes, simply by introducing new subscripts. Until now, we have only one kind of subscript, 'r', denoting running complexity. We now introduce three new subscript z,

$$z \in \{b, 0, 1\},$$

to indicate the following restrictions: **bounded-error** ($z = b$) or **one-sided error** ($z = 1$) or **bounded zero-error** ($z = 0$). Note a linear hierarchy in these subscripts: for instance,

$$PrTIME_0(t) \subseteq PrTIME_1(t) \subseteq PrTIME_b(t) \subseteq PrTIME_r(t) \subseteq PrTIME(t)$$

for any function t. We could replace $PrTIME$ here by any mode-resource pair. These notations are illustrated in the last column of the following table.

The table below lists some important time-feasible (i.e., polynomial time) complexity classes, under the various choice and error modes:

Some Polynomial Time Stochastic Classes

Error Mode	*Choice Mode*	*Common Symbol*	*Standard Notation*
Unbounded error	$\{\widetilde{\exists}\}$	PP	$PrTIME(n^{O(1)})$
Bounded error	$\{\widetilde{\exists}\}$	BPP	$PrTIME_b(n^{O(1)})$
One-sided error	$\{\widetilde{\exists}\}$	RP (also denoted VPP or R)	$PrTIME_1(n^{O(1)})$
Bounded zero-error	$\{\widetilde{\exists}\}$	ZPP	$PrTIME_0(n^{O(1)})$
Bounded error	$\{\widetilde{\exists}, \vee\}$	IP, AM	$IpTIME_b(n^{O(1)})$

We note some relationships among these classes and the canonical classes such P and NP. The following are trivial relationships.

$$P \subseteq ZPP \subseteq RP \subseteq BPP \subseteq PP,$$

$$BPP \subseteq IP \cap PP.$$

We also have

$$RP \subseteq_1 NP \subseteq_2 PP.$$

The inclusion ($\subseteq_2$) follows from equation (8.1) while inclusion ($\subseteq_1$) follows from equation (8.2).

From the previous chapter (corollary 25), we have:

$$PP \subseteq IP \subseteq PSPACE.$$

However, we shall see that $IP = PSPACE$. Recall that $Primes \in co\text{-}RP$. We now show the following, attributed to Rabin:

$$ZPP = RP \cap co\text{-}RP.$$

Proof. It is clear that $ZPP \subseteq RP \cap co\text{-}RP$. Conversely, suppose L is accepted by a probabilistic machine M (respectively, N) that runs in polynomial time with bounded zero-error acceptance (respectively, rejection). We may assume that M and N are halting. We then construct a probabilistic machine that dovetails the

computation of M and N in a step-for-step fashion until one of the following two events: (a) if N answers YES, we answer YES; (b) if M answers NO, we answer NO. If N answers NO or M answers YES, we simply continue simulation of the other machine. If both machines halt without events (a) or (b) occurring, we loop. This dovetail process essentially gives us a computation tree whose paths can be made to correspond in a bijective fashion with the set of all pairs of paths (p, p') where p comes from M and p' from N. Then it is not hard to see that the simulation runs in polynomial time with zero-error (Exercises).

Figure 8.1 summarizes the relationship of the main feasible stochastic-time classes with the canonical classes:

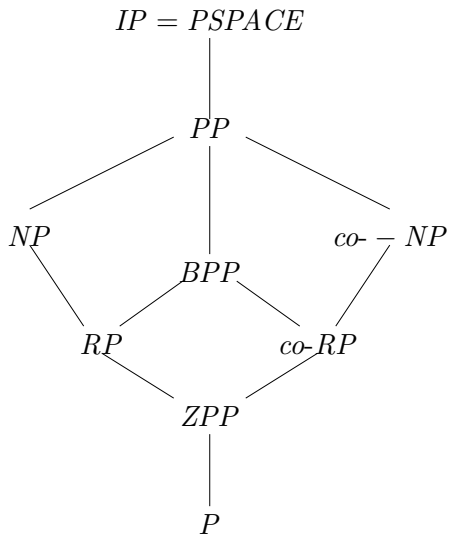


Figure 8.1: Feasible classes in the stochastic mode

8.2 How to Amplify Error Gaps

It is said that bounded-error probabilistic algorithms are practically as good as deterministic ones. In graphic terms, an algorithm that makes error with probability less than 2^{-1000} seems eminently more reliable than many other non-mathematical aspects[3] within the whole computational process. Basically, we seek techniques to convert a machine with error gap G into one with a strictly larger error gap G', $G \subset G'$. We now describe three such techniques, depending on the form of error and type of machine.

[3]Such as the reliability of hardware, not to speak of software.

(I) Zero-error Rejection Machines. Let M be a probabilistic machine zero-error rejection. Suppose M accepts[4] if with error gap $[0, b]$. We can boost the error gap very easily as follows. Fix some $k \geq 1$. Let N be the following machine:

> On any input w, simulate M on w for at most k times. If any of the simulation answers YES, N answers YES at once. If a simulation halts with the answers NO or YO, we go on to the next simulation. If the answer is NO or YO for k times, N answers NO.

If w is rejected by M, then N always answer NO. If w is accepted by M, the probability of a YES-computation path of M is at least b. Hence the probability that N accepts is at least $1 - (1 - b)^k$. So N accepts with error gap

$$[0, 1 - (1 - b)^k]$$

which strictly includes $[0, b]$. Thus, *zero-error rejection implies bounded-error acceptance.*

(II) Bounded-error Probabilistic Machines. The above technique will not work when there could be pessimistic errors in rejection. For bounded-error machines, we can use the 'majority voting' scheme: repeat for an odd number of times an experiment with binary outcome; we take the majority outcome (i.e., the outcome occurring more than half the time) as output. We justify this procedure with a lemma [26]:

Lemma 1 *(a) Consider an experiment in which an event E occurs with probability*

$$p \geq \frac{1}{2} + e$$

for some $0 < e < \frac{1}{2}$. Then in $2t + 1$ independent trials of the experiment, the probability that E is the majority outcome is greater than

$$1 - \frac{1}{2}(1 - 4e^2)^t.$$

(b) Similarly, if E occurs with probability

$$p \leq \frac{1}{2} - e$$

then the probability that E is the majority outcome is less than

$$\frac{1}{2}(1 - 4e^2)^t.$$

[4]Of course, $b \geq 1/2$. But in some sense, this technique works as long as $b > 0$.

Proof. (a) Let $q = 1 - p$ and $i = 0, \ldots, t$. Then the probability p_i that E occurs exactly i times out of $2t + 1$ is given by the binomial distribution,

$$\begin{aligned}
p_i &= \left(\begin{array}{c} 2t+1 \\ i \end{array} \right) p^i q^{2t+1-i} \\
&= \left(\begin{array}{c} 2t+1 \\ i \end{array} \right) (\frac{1}{2} + e)^i (\frac{1}{2} - e)^{2t+1-i} \left[\frac{p}{\frac{1}{2}+e} \right]^i \left[\frac{q}{\frac{1}{2}-e} \right]^{2t+1-i} \\
&= \left(\begin{array}{c} 2t+1 \\ i \end{array} \right) (\frac{1}{2} + e)^i (\frac{1}{2} - e)^{2t+1-i} \left[\frac{pq}{(\frac{1}{2}+e)(\frac{1}{2}-e)} \right]^i \left[\frac{q}{\frac{1}{2}-e} \right]^{2t+1-2i} \\
&\leq \left(\begin{array}{c} 2t+1 \\ i \end{array} \right) (\frac{1}{2} + e)^i (\frac{1}{2} - e)^{2t+1-i} \\
&\leq \left(\begin{array}{c} 2t+1 \\ i \end{array} \right) (\frac{1}{2} + e)^i (\frac{1}{2} - e)^{2t+1-i} \left[\frac{\frac{1}{2}+e}{\frac{1}{2}-e} \right]^{t-i} \\
&= \left(\begin{array}{c} 2t+1 \\ i \end{array} \right) (\frac{1}{2} + e)^t (\frac{1}{2} - e)^{t+1} \\
&< \left(\begin{array}{c} 2t+1 \\ i \end{array} \right) (\frac{1}{4} - e^2)^t \frac{1}{2}.
\end{aligned}$$

Therefore the probability that E occurs in more than t trials is at least

$$\begin{aligned}
1 - \sum_{i=0}^{t} p_i &> 1 - \sum_{i=0}^{t} \left(\begin{array}{c} 2t+1 \\ i \end{array} \right) (\frac{1}{4} - e^2)^t \frac{1}{2} \\
&= 1 - 2^{2t} (\frac{1}{4} - e^2)^t \frac{1}{2} \\
&= 1 - \frac{1}{2}(1 - 4e^2)^t.
\end{aligned}$$

(b) Similarly, if $p \leq \frac{1}{2} - e$ then for $i \geq t + 1$ we have

$$p_i \leq \left(\begin{array}{c} 2t+1 \\ i \end{array} \right) (\frac{1}{4} - e^2)^t \frac{1}{2}$$

and hence the probability that E occurs in more than t trials will be at most

$$\begin{aligned}
\sum_{i=t+1}^{2t+1} p_i &\leq 2^{2t} (\frac{1}{4} - e^2)^t \frac{1}{2} \qquad (8.3) \\
&= \frac{1}{2}(1 - 4e^2)^t. \qquad (8.4)
\end{aligned}$$

Q.E.D.

Using this, we can boost an error gap $G_e = [\frac{1}{2} - e, \frac{1}{2} + e]$ $(0 < \frac{1}{2} < e)$ to $[\frac{1}{2} - e', \frac{1}{2} + e']$ where

$$e' = \frac{1}{2}(1 - 4e^2)^t$$

if we do the majority vote for $2t + 1$ trials. For instance, with $e = 1/4$ and $t = 8$, we have $e' = \frac{1}{2}(3/4)^t < 0.051$.

Let G_0 be the error gap function given by

$$G_0(n) = [2^{-n}, 1 - 2^{-n}].$$

We have the following useful lemma:

Lemma 2 *Each language in BPP is accepted by a probabilistic acceptor that runs in polynomial time with error gap G_0.*

Proof. We may assume that the language is accepted by some M that runs in time n^d with error gap $G = [\frac{1}{2} - e, \frac{1}{2} + e]$ for some $d \geq 1$ and $0 < e < \frac{1}{2}$. Applying the lemma, we want to choose t satisfying

$$\begin{aligned} 2^{-n} &\geq \frac{(1 - 4e^2)^t}{2} \\ 2^{n-1} &\leq \frac{1}{(1 - 4e^2)^t} \\ n - 1 &\leq t \log\left(\frac{1}{1 - 4e^2}\right) \\ t &\geq \frac{n - 1}{\log(1/(1 - 4e^2))}. \end{aligned}$$

The desired machine N, on each computation path, simulates M for at most $2t+1 = O(n)$ times and outputs the majority outcome. Clearly N runs in time $O_e(n^{d+1})$ with error gap G_0. **Q.E.D.**

Let us give one immediate application of this lemma:

Theorem 3 (Ko, 1982) *If NP $\subseteq$ BPP then NP = RP.*

Proof. Since $RP \subseteq NP$, it suffices to show inclusion in the other direction. It is easy to see that RP is closed under polynomial-time many-one reducibility, and hence we only have to show that the NP-complete language SAT belongs to RP. Suppose we want to check if a given CNF formula $F = F(x_1, \ldots, x_n)$ on n variables is satisfiable. For any sequence of Boolean values $b_1, \ldots, b_k$ $(k \leq n)$, let $F_{b_1 b_2 \cdots b_k}$ denote the formula F with x_i replaced by b_i, for $i = 1, \ldots, k$. We show how to construct a sequence $b_1, \ldots, b_n$ such that if F is satisfiable then $F_{b_1 \cdots b_n}$ is true with

very high probability. By our assumption that $NP \subseteq BPP$, there is a bounded-error probabilistic machine M accepting SAT in polynomial time. Moreover, by the preceding lemma, we may assume that M has error gap function G_0 and that M halts on every path in polynomial time.

We shall operate in n stages. At the start of stage k ($k = 1, \ldots, n$), inductively assume that we have computed a sequence of Boolean values $b_1, \ldots, b_{k-1}$. It will be shown that $F_{b_1,\ldots,b_{k-1}}$ is probably satisfiable. In stage k, we compute b_k:

1.	Call M on input $F_{b_1 \cdots b_{k-1}0}$.
2.	If M answers YES, then set $b_k = 0$ and go to DONE.
3.	Else call M on input $F_{b_1 \cdots b_{k-1}1}$.
4.	If M answers NO again, we answer NO and return.
5.	Else set $b_k = 1$.
6.	DONE: If $k < n$ we go to stage $k + 1$.
7.	Else answer YES if $F_{b_1,\ldots,b_n} = 1$, otherwise answer NO.

Let us analyze this procedure. It is clearly polynomial time.

If $k < n$, we either terminate in stage k with a NO answer, or we proceed to stage $k + 1$. If $k = n$, we will surely terminate in stage k with answer YES or NO, and this answer is never in error. Thus our YES answers are never wrong. So if F is unsatisfiable, we answer NO on every path. Thus we have zero-error rejection.

Finally, let us prove that if F is satisfiable, then our procedure answer YES with probability $> 1/2$. Write F_k for $F_{b_1,\ldots,b_k}$, assuming that $b_1, \ldots, b_k$ are defined. Let the event A_k correspond to " no mistakes up to stage k", *i.e.*, F_k is defined and satisfiable. Similarly, let event E_k correspond to "first mistake at stage k", *i.e.*, $E_k = A_{k-1} \cap \overline{A_k}$.

CLAIM: $\Pr(E_k) \leq 2 \cdot 2^{-|F|+1})$.

Proof: Note that $\Pr(E_k) \leq \Pr(E_k|A_{k-1})$. We will bound $\Pr(E_k|A_{k-1})$. Assuming A_{k-1}, we consider 2 cases: (A) CASE $F_{b_1 \cdots b_{k-1}0}$ is not satisfiable. Then $F_{b_1 \cdots b_{k-1}1}$ is satisfiable. With probability $\geq (1 - 2^{-|F|})$, M will (correctly) answer NO the first time we invoke M. Then with probability $\geq (1 - 2^{-|F|})$, M will (correctly) answer YES the second time. So $\Pr(A_k|A_{k-1}) \geq (1 - 2^{-|F|})^2$ and

$$\Pr(E_k|A_{k-1}) \leq 1 - (1 - 2^{-|F|})^2 \leq 2^{-|F|+1}).$$

(B) CASE $F_{b_1 \cdots b_{k-1}0}$ is satisfiable. This case is even easier, and yields $\Pr(E_k|A_{k-1}) \leq 2^{-|F|}$. This proves the claim.

To conclude the theorem, the probability of making mistake at any stage is at most

$$\sum_{k=1}^{n} \Pr(E_k) \leq 2n \cdot 2^{-|F|} \leq 2n \cdot 2^{-2n}.$$

This is less than $1/2$ for n large enough. Hence F will be accepted. **Q.E.D.**

See Exercise for another proof.

(III) Stochastic machines. For stochastic machines, we introduce the following probability functions:

Definition 5
(i) $P(x) := x \otimes x = x^2$ *and* $Q(x) := x \oplus x = x(2-x)$. *(ii)* $A(x) := Q(P(x)) \,\overline{\oplus}\, P(Q(x))$.
■

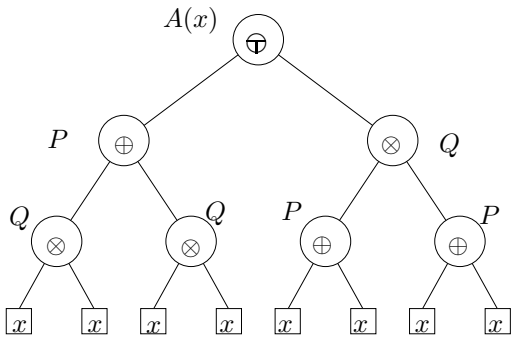


Figure 8.2: The operator $A(x)$

Thus,

$$A(x) = \frac{x^2(2-x^2) + x^2(2-x)^2}{2} = x^2(3-2x).$$

These operators are extended to INT in the usual way: thus, $A([u,v]) = [A(u), A(v)]$. The exercises show other properties of these functions. We show that $A(x)$ has the following 'amplification property':

Lemma 4 *If* $0 \leq e \leq \frac{1}{2}$, *then*

$$x \notin [\frac{1}{2} - e, \frac{1}{2} + e] \quad \Rightarrow \quad A(x) \notin [\frac{1}{2} - e', \frac{1}{2} + e']$$

where

$$e' = e(\frac{3}{2} - 2e^2).$$

Proof. Since $\frac{dA}{dx} = 6x(1-x)$, $A(x)$ is monotone increasing for $0 \leq x \leq 1$. The lemma then follows from a calculation,

$$\begin{aligned} A(\frac{1}{2} + e) &= \frac{1}{2}[Q(P(\frac{1}{2} + e)) + P(Q(\frac{1}{2} + e))] \\ &= \frac{1}{2} + e[\frac{3}{2} - 2e^2] \\ A(\frac{1}{2} - e) &= \frac{1}{2} - e[\frac{3}{2} - 2e^2]. \end{aligned}$$

Q.E.D.

Note that the error gap has an "amplification factor" $\alpha(e) = \frac{3}{2} - 2e^2$ which decreases monotonically from 3/2 to 1 as e increases from 0 to 1/2. Note that

$$\alpha(1/4) = \frac{11}{8}.$$

So if the error bound e for a value x is less than 1/4, then the error bound for $A(x)$ is at least $11e/8$. We conclude:

Theorem 5
Let $t(n)$ be a time-constructible function. Then

$$PrTIME(t) \subseteq StTIME_b(O(t)).$$

In particular,

$$PP \subseteq StTIME_b(n^{O(1)}).$$

Proof. Suppose M is a probabilistic machine M that accepts in time t. Since t is time-constructible, we can modify it always halt in time $O(t)$ and have error gap

$$G(n) = [\frac{1}{2} - 2^{-t}, \frac{1}{2} + 2^{-t}].$$

Let $s = \frac{t}{\log(11/8)}$. We construct a stochastic machine N that, on input w, operates by first 'computing the iterated function $A^s(x)$' (the meaning should be clear) and then simulating M on w. One checks that N has gap at least $[\frac{1}{4}, \frac{3}{4}]$. Note that N still runs in time $O(t)$. **Q.E.D.**

8.3 Average Time and Bounded Error

Notice that we have never discuss the concept of "average time" for probabilistic machines. We can systematically develop the concept as follows (the reader may review the appendix for probabilistic terms used here). Let M be a fixed probabilistic machine and $T_{\rm M}(w)$ denotes the usual complete computation tree on input w. We assume $T_{\rm M}(w)$ is a full binary tree, *i.e.*, a binary tree in which every internal node has two children. Let T any prefix of $T_{\rm M}(w)$. The **augmentation** T' of T is defined as the smallest prefix of $T_{\rm M}(w)$ that is full and contains T. Alternatively, T' consists of the nodes of T plus those nodes $u \in T_{\rm M}(w)$ whose sibling is in T. Call the nodes in $T' - T$ the **augmented nodes**.

We construct associate with T a probability space

$$(\Omega_T, \Sigma_T, \Pr_T)$$

in which the sample space Ω_T comprises all complete paths of T'. A subset of Ω_T is a **basic set** if it is the collection of complete paths of T' all sharing a common initial

prefix π; denote this set[5] by $B_T(\pi)$. In particular, Ω_T is the basic set $B_T(\epsilon)$ where $\pi = \epsilon$ is the empty path. Any singleton set consisting of a complete path is also a basic set. Let Σ_T^0 comprise all finite union and complement of basic sets: clearly Σ_T^0 forms a field. Let Σ_T be the Borel field generated by Σ_T^0. The probability measure $\Pr_T$ assigns to each basic set $B_T(\pi)$ the probability $\Pr_T(B_T(\pi)) = 2^{-|\pi|}$ where $|\pi|$ is the length of the path π. E.g., $\Pr_T(\Omega_T) = 2^0 = 1$, as expected. Notice that every element of Σ_T^0 is a finite union of disjoint basic sets. Extend the definition of $\Pr_T$ to sets in Σ_T^0 so as to preserve finite additivity of pairwise disjoint unions. One checks that the extension does not depend on how we partition we partition sets in Σ_T^0 into a countable union of basic sets. Finally, a theorem of Carathéodory (appendix) gives us a unique extension of $\Pr_T$ to all of Σ_T.

We often leave the probabilistic spaces Ω_T implicit – but the student should be able to formalize probabilistic arguments uing these spaces. We introduce three random variables for Ω_T:

Definition 6 (i) *The random variable* Time_T *for a computation tree* T *is defined as follows:*

$$\text{Time}_T(\pi) = \begin{cases} \infty & \text{if } \pi \text{ is infinite} \\ |\pi| & \text{if } \pi \text{ is a finite path terminating in } T \\ 0 & \text{if } \pi \text{ is a finite path terminating in } T' - T \end{cases}$$

where π *range over all complete paths in* Ω_T*.*

(ii) *The* **average time** *of* T *is the expected value of* Time_T*.*

(iii) *We introduce two indicator functions:* $\text{Accept}_T(\pi) = 1$ *iff* π *ends in a YES-node in* T*.* $\text{Halt}_T(\pi) = 1$ *iff* π *is a finite path.*

(iv) *If* T *is the complete computation tree for an input* w*, we write*

$$\text{Time}_w, \text{Accept}_w, \text{Halt}_w, \Omega_w, etc,$$

for Time_T*,* Accept_T*, etc.*

(v) *A machine* M **accepts/rejects in average time** $t(n)$ *if for all accepted/rejects inputs* w *of length* n*,*

$$\Pr\{\text{Time}_w \leq t(n), \text{Accept}_w = 1\} > \frac{1}{2}.$$

It **runs in averate time** $t(n)$ *if it is decisive and accepts and also rejects in average time* $t(n)$*.*

■

[5]The reason for introducint augmented trees is to ensure that $B_T(\pi) \neq B_T(\pi')$ for $\pi \neq \pi'$. Otherwise our probability space for T needs to be slightly modified.

An equivalent definition of average time is this: the average time of a computation tree T is equal to the sum over the weights of each edge in T where an edge from level $\ell - 1$ to level ℓ (the root is level 0) has weight $2^{-\ell}$. Naturally, **the probability that** M **halts on** w is the expected value of the random variable Halt_w. If M is halting (*i.e.*, there are no infinite computation paths) then it halts with probability 1; the converse is false. When we use running complexity for probabilistic machines, they are sometimes designated 'Monte Carlo algorithms'; when average complexity is used, they are designated 'Las Vegas algorithms'.

We first note an observation of Gill: *Every recursively enumerable language can be accepted by a probabilistic machine with constant average time.*

In proof, suppose M is a deterministic Turing machine. Without loss of generality assume that M accepts whenever it halts. We construct N to simulate M as follows:

```
repeat
    Simulate one step of M;
    if the simulated step answers YES, we answer YES;
until head = cointoss();
if head = cointoss() then answer YES else answer NO.
```

Here cointoss() is a random function that returns *head* or *tail* and there are two separate invocations of this function above. We note that if the input word is not accepted by M then N can only reject (since the probability of N saying YES is equal to the probability of saying NO). Hence N has zero-error rejection. If an input word is accepted by M, then we see that the probability of its acceptance by N is more than $\frac{1}{2}$ since each NO path can be uniquely paired with an YES path of the same length, but there is one YES path that is not paired with any NO path. The average time $\bar{t}$ spent in the repeat-loop satisfy the inequality

$$\bar{t} \leq 2 + \frac{\bar{t}}{2}$$

where the term '2+' comes from simulating a step of M and tossing a coin to decide on continuing inside the loop (it takes no time to decide to say YES if M says YES). Thus $\bar{t} \leq 4$. The average time of N is at most $1 + \bar{t} \leq 5$ (where the '1' for the final cointoss).

Gill notes that such pathological behaviour does not happen with bounded-error machines:

Lemma 6 *Let* M *be a probabilistic machine accepting/rejecting with bounded-error. There is a constant* $c > 0$ *such that if* M *accepts/rejects in average time* $\bar{t}(n)$ *and accepts/rejects in time* $t(n)$ *then*

$$\bar{t}(n) \geq \frac{t(n)}{c}.$$

Proof. Suppose M accepts with probability at least $\frac{1}{2} + e$ for some $0 < e < \frac{1}{2}$. (The proof if M rejects with probability at most $\frac{1}{2} - e$ is similar.) Choose $c = \frac{2}{e}$. Fix any input of length n and let $\bar{t} = \bar{t}(n)$. If $\bar{t} = \infty$ there is nothing to prove. Otherwise, let T be the complete computation tree. Since Time_T is non-negative, Markov's inequality yields

$$\Pr\{\text{Time}_T \geq c\bar{t}\} \leq \frac{1}{c}.$$

On the other hand,

$$\begin{aligned} \Pr\{\text{Time}_T < c\bar{t}, \text{Accept}_T = 1\} &\geq \Pr\{\text{Time}_T < c\bar{t}\} - \Pr\{\text{Accept}_T = 0\} \\ &\geq (1 - \frac{1}{c}) - (\frac{1}{2} - e) \\ &\geq \frac{1}{2} + \frac{e}{2}. \end{aligned}$$

This proves that T, truncated below $c\bar{t}$, accepts with bounded error. **Q.E.D.**

In view of the preceding, let

$$AvgTIME(t(n))$$

denote the class of languages accepted by probabilistic machines M where M has bounded-error and M runs in average time $t(n)$. Note that both properties here are independently defined for the entire computation tree. We have thus proved:

Corollary 7 *For any $t(n)$,*

$$AvgTIME(t) \subseteq PrTIME_b(O(t)). \tag{8.5}$$

As further example, we provide deterministic time upper bounds for languages accepted in average time $t(n)$ with bounded-error.

Corollary 8 *If a bounded-error probabilistic machine* M *accepts with average time $\bar{t}(n)$ then $L(\text{M}) \in DTIME(O(1)^{\bar{t}(n)})$.*

Proof. We can simulate M by computing the least fixed point of a computation tree of depth $O(\bar{t}(n))$. **Q.E.D.**

Lemma 9 *Let $s(n) \geq \log n$ be space constructible. Let* M *be any nondeterministic machine that accepts in space s. Then there is a probabilistic machine* N *with zero-error that accepts $L(\text{M})$ in space s.*

Proof. Choose $c > 0$ such that there are at most $c^{s(n)}$ configurations using space at most $s(n)$. Fix any input of length n and let $s = s(n)$. First we mark out exactly s cells. The probabilistic machine N proceeds as follows:

repeat forever
1. Initialize M to its initial configuration.
2. Simulate M for c^s steps. Nondeterministic choices of M become coin-tossing choices of N.
3. If M answers YES in this simulation then answer YES. (Hence, if M answers NO or does not halt in c^s steps, then we go to 1.)

end

Clearly N loops if M rejects. If M accepts then the probability that N answering YES is easily seen to be 1. **Q.E.D.**

This lemma implies that probabilistic space-bounds with zero-error is as powerful as nondeterministic space:

Theorem 10 *For any space-constructible $s(n) \geq \log n$,*

$$NSPACE(s) = PrSPACE_0(s) = PrSPACE_1(s).$$

Proof. The above lemma shows that $NSPACE(s) \subseteq PrSPACE_0(s)$. The converse is easy since for any probabilistic machine M with zero error, when viewed as a nondeterministic machine N, accepts the same language with the same space bound. We check that the same construction applied to probabilistic one-sided error machines in place of nondeterministic machines show $PrSPACE_1(s) \subseteq PrSPACE_0(s)$, and hence they are equal. **Q.E.D.**

This result can be generalized to log-space alternating machines, but we now have two-sided error [25].

The simulation in the above proofs can be modified so that the simulating machine N halts with probability 1. However, N is no longer zero-error. The technique will be introduced in section 6.

Probabilistic Simulating of Alternation. Consider how a probabilistic machine can simulate an alternating machine M. Moreover, we want our probabilistic machine to have bounded error. Suppose C is a configuration of M and $C \vdash (A, B)$. Let T_C, T_A, T_C denote the subtree at these nodes.

Inductively, assume that our recursive construction gives us probabilistic computation trees $T_{A'}$ and $T_{B'}$ (rooted at A' and B') which emulates T_A and T_B (respectively) with error gap

$$g_0 = [1/4, 3/4].$$

This means that if A accepts, then the value of A' is at least $3/4$ and if A rejects, the value of A' is at most $1/4$. Similarly for B and B'. Let us see how to carry the induction through.

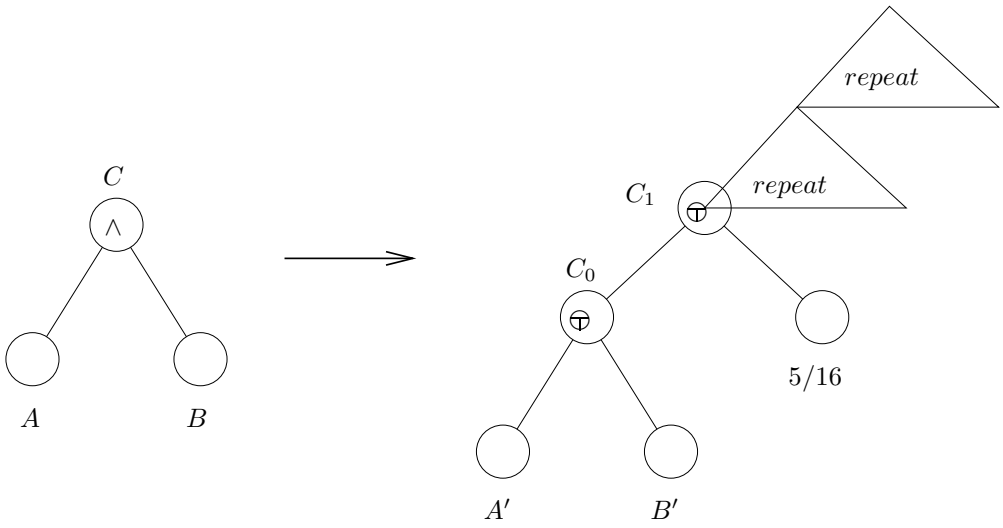


Figure 8.3: Simulating a $\wedge$-configuration C.

CASE: C is a $\wedge$-configuration. Let C_0 be a $\oplus$-configuration such that $C_0 \vdash (A', B')$ (see figure 8.3). Then the value at C_0 has an error gap of

$$[5/8, 3/4].$$

This is because, if at least one of A' or B' rejects, then value of C_0 is at most $(1/4) \oplus 1 = 5/8$. And if both A' and B' accepts, the value is at least $3/4$. Then, we 'shift' the gap so that it is centered, by averaging it with the value $5/16$. This gives us a new gap (of half the size!)

$$g_0 = [15/32, 17/32].$$

We now use majority voting to boost this gap back to at least $[1/4, 3/4]$. We need to take the majority of $2k + 1$ votes, where k is a constant that can be computed.

CASE: C is a $\vee$-configuration. In this case, C_0 has error gap of

$$[1/4, 3/8].$$

Now we shift this gap by averaging it with $11/16$, yielding the gap g_0 above. We again boost it back to $[1/4, 3/4]$.

Note that if the original tree has height h and this procedure produces a tree of height $\tau(h)$, then

$$\tau(h) = k(\tau(h-1) + 2) \leq (2k)^h.$$

Of course, we need to make this recursive transformation something that can be carried out by a suitable probabilistic machine. This is left as an exercise. We have thus shown:

Theorem 11

$$ATIME(t) \subseteq PrTIME_b(2^{O(t)}).$$

8.4 Interactive Proofs

This section introduces interactive proofs [13], Arthur-Merlin games [2].

Graph Non-Isomorphism Problem. A motivating example is the graph isomorphism problem: given a pair of undirected graphs $\langle G_0, G_1\rangle$, decide if they are isomorphic,

$$G_0 \sim G_1. \tag{8.6}$$

Let $\mathcal{G}_n$ denote the set of bigraphs on the vertex set $\{1, 2, \ldots, n\}$. Formally define the languages

$$\begin{aligned} \text{ISO} &= \{\langle G_0, G_1\rangle \in \mathcal{G}_n^2 : n \geq 1, G_0 \sim G_1\}, \\ \text{NONISO} &= \{\langle G_0, G_1\rangle \in \mathcal{G}^2 : n \geq 1, G_0 \not\sim G_1\}. \end{aligned}$$

These are basically complementary languages. If π is an n-**permutation** (*i.e.*, a permutation of $\{1, \ldots, n\}$), let G_0^π denote the graph G_0 when its vertices are renamed according to π: (i, j) is an edge of G_0 iff $(\pi(i), \pi(j))$ is an edge of G_0^π. Then (8.6) holds iff there exists π such that $G_0^\pi = G_1$. Call π a **certificate** for $\langle G_0, G_1\rangle$ in this case. Thus $\langle G_0, G_1\rangle \in$ ISO iff $\langle G_0, G_1\rangle$ has a certificate. Thus ISO has two useful properties:

- (Succinct Certificates) $x \in$ ISO iff x has a polynomial-size certificate $c = \pi$.
- (Verifiable Certificates) There is a deterministic polynomial-time algorithm V to decide if a given c is a certificate π for a given x.

These two properties characterize languages in *NP*. Hence, ISO $\in$ *NP*. It easily follows NONISO $\in$ *co-NP*. Unfortunately, *co-NP* does not have a characterization by certificates. Although certificates are not necessarily easy to find, they are easy to verify. In this sense, they have practical utility. We now introduce a generalizaation of certificates which shares this verifiability property. We then show that NONISO is verifiable in this more general sense.

Concept of Interactive Proofs. We generalize notion of verifiable certificates in two ways: first, we allows the verifying algorithm V to be probabilistic, and second, we allow interaction between the verifying algorithm with another algorithm called the "prover", denoted P. Thus there are two communicating processes (sometimes called **protocols**), an **interactive prover** P and an **interactive verifier** V which are Turing machines that send each other messages,

$$m_0, m_1, m_2, \ldots.$$

Message m_i is written by V if i is even, and by P if i is odd, and these are written on a common worktape. We assume some convention for each process to indicate that it is done writing its message (say, by entering a special state) and for some external agent to prompt the other process to continue. The computation ends when V answers YES or NO. The input is originally on V's input tape. We require V to be a probabilistic polynomial-time machine, but P has no computational bound (it does not even have to be computable). Intuitively, V is sceptical about what the process P is communicating to it, and needs to be "convinced" (with high probability). For any input w, let $\Pr(V, P, w)$ be the probability that V accept. We assume that V always halt, regardless of P, so that we avoid discussion of probability intervals. Languages will be defined with respect to V alone: writing

$$\Pr(V, w) := \sup_P \Pr(V, P, w),$$

then the language accepted by V is

$$L(V) := \{w : \Pr(V, w) > 1/2\}.$$

Say V has **bounded-error** if, to the preceding requirements, it satisfies

For all input w, $\Pr(V, w) \geq 2/3$ or $\Pr(V, w) \leq 1/3$.

The class *IP* comprises those languages that are accepted by bounded-error verifiers.

Interactive Verifier for Graph Non-Isomorphism. We want to describe an interactive verifer V such that $L(V) = \text{NONISO}$. Here is a well-known V_0 from the literature:

INPUT: STRING w
1. Reject unless $w = \langle G_0, G_1 \rangle \in \mathcal{G}_n^2$, $n \geq 1$.
2. Randomly generate an n-permutation π and a binary bit b.
3. Let $H \leftarrow G_b^\pi$ (so $H \sim G_b$).
4. Send message $m_0 = \langle H, G_0, G_1 \rangle$. This message asks P whether $H \sim G_0$ or $H \sim G_1$.
5. (Pause for P to reply with message m_1)
6. If $b = m_1$ answer YES, else answer NO.

Note that V_0 concludes in two message rounds (sends and receives a message). Assume $w = \langle G_0, G_1 \rangle$. There are two cases to consider.

- $w \in \text{NONISO}$: We claim $\Pr(V_0, w) = 1$. To see this, suppose P_0 is the prover who sends the message $m_1 = c$ such that $H \sim G_c$. Since c is unique, V_0 always answer YES, so $\Pr(V_0, P_0, w) = 1$.

- $w \notin$ NONISO: We wish to claim $\Pr(V_0, w) = 1/2$. Intuitively, an "honest prover" P_0 cannot distinguish whether the answer should be $H \sim G_0$ or $H \sim G_1$. It is reasonable for P_0 to flip a coin and answer $m_1 = 0$ and $m_1 = 1$ with equal probability. This will establish our claim. But what we have a "dishonest prover" P_1 whose goal is to mislead V_0 into accepting w? P_1 knows something about about π and b it may be able to mislead V_0. For instance, if P_1 knows the value of b, then it will always fool V_0. How can we be sure that such information has not leaked in our definition of message m_0? This justification is non-trivial (see [19, p. 175]) and may be based on the so-called Principle of Deferred Decisions.

This example points out that informal descriptions of interactive proofs (with suggestive language such as "V is convinced", "P knows", etc) can be tricky to formalize. For this reason, we prefer to use the view interactive proofs as choice computations. The idea is this: we can combine V and P into one choice machine denoted, loosely,

$$\mathrm{M} = \text{“}V + P\text{”},$$

where the states of M is the disjoint union of the states of V and P (so each state of M may be classified as a P-state or a V-state). We will let the choice function at each V-state q be $\gamma(q) = \oplus$. But what about P? We need to simulate all possible behavior for P (recall that we define $\Pr(V, w)$ as the maximization of $\Pr(V, P, w)$). Fortunately, this is not hard (we can essentially make all possible choices for the message). Furthermore, we let the choice function at each P-state q be $\gamma(q) = \vee$. Thus M is a $\{\oplus, \vee\}$-machine. Unfortunately, there are two issues.

One issue is that, although P is powerful, it seems that we do not want it to know about the coin tosses of V. Such a verifier is said to use "private coins". The formulation "M = $V + P$" apparently use "public coins". As noted, the use of public coins in V_0 above would be disastrous! Verifiers with private coins seems more powerful. It turns out, for polynomial-time computations, a verifier with private coins can be simulated by one with public coins, at the cost of of two extra rounds [14]:

$$IP[k] \subseteq AM[k+2]. \tag{8.7}$$

The parameters k and $k+2$ bound the number of message rounds; the full explanation for this notation is given below,

The second issue is this: the 0/1-message m_1 returned by the prover does not seem to be easily modeled by $\oplus$- and $\vee$-choices alone. The ability to pass a 0/1-message from P to V seems more powerful than simply allowing V to ask P question and receiving a YES/NO answer (Why?). For instance, V upon receipt of the Boolean message m_1, can trivially compute the negation of m_1. But a $\{\oplus, \vee\}$-computation cannot trivially negate the value at a node. Thus it seems we need a $\{\oplus, \vee, \neg\}$-machine (equivalently, a $\{\oplus, \vee, \wedge\}$-machine) to efficiently simulate an interactive proof. But this would make the interactive prover for NONISO uninteresting (NONISO is trivially accepted by a $\wedge$-machine in polynomial-time.) It turns

out $\neg$ can be avoided, but this is a non-trivial result. We will next avoid these complications of interactive proofs by using the Arthur-Merlin formulation of Babai and Moran. The advantage, besides its simplicity, is the direct connection to choice computation (which, as the previous chapter shows, has clean semantics).

Arthur-Merlin Games. Let M be an $\{\oplus, \vee\}$-machine, and $\pi = (C_0, C_1, \ldots, C_m)$ be a computation path of M. A subpath

$$\pi' = (C_i, C_{i+1}, \ldots, C_j), \quad (1 \leq i \leq j \leq m)$$

is called a $\oplus$**-round** (or **Arthur round**) if it contains at least one $\oplus$-configuration but no $\vee$-configurations. Notice that π' could contain deterministic configurations. Similarly, π' is called a $\vee$**-round** (or **Merlin round**) if we interchange $\oplus$ and $\vee$. We say π has k **rounds** if π can be divided into k subpaths,

$$\pi = \pi_1; \pi_2; \cdots; \pi_k \quad (k \geq 1)$$

where ";' denotes concatenation of subpaths such that π_i is an Arthur round iff π_{i+1} is a Merlin round. Note that k is uniquely determined by π. The definition generalizes in a natural way to $k = 0$ and $k = \infty$. We say M is a k**-round Arthur-Merlin game** if

- M has bounded error and runs in polynomial time
- every computation path of M has at most k rounds in which the first round (if any) is an Arthur round

A k**-round Merlin-Arthur game** is similarly defined, with the roles of Arthur and Merlin interchanged. Let

$$AM[k] = \{L(\mathrm{M}) : \mathrm{M} \text{ is an Arthur-Merlin game with at most } k \text{ rounds}\}$$

The class $MA[k]$ is similarly defined using Merlin-Arthur games instead. Of course, we can generalize this to $AM[t(n)]$ and $MA[t(n)]$ where $t(n)$ is a complexity function.

We can identify[6] Arthur with the verifier V, and Merlin with the prover P, of interactive proofs. We can similar define the classes $IP[k]$ and $IP[t(n)]$ accepted by interactive proofs in k or $t(n)$ rounds. This is the notation used in (8.7).

Arthur-Merlin Game for Graph Non-Isomorphism. The new idea for checking non-isomorphism is as follows: let $G_0, G_1 \in \mathcal{G}_n$. For simplicity, first assume that G_i $(i = 0, 1)$ has only the trivial automorphism (*i.e.*, $G_i^\pi = G_i$ iff π is the identity permutation). Consider the set

$$\mathrm{LIKE}'(G_0, G_1) = \{H : H \sim G_0 \text{ or } \ H \sim G_1\}.$$

[6]Here, as in the legend of King Arthur, the magician Merlin is more powerful than Arthur.

Then $|\text{LIKE}'(G_0, G_1)|$ is either $n!$ or $2n!$, depending on whether $G_0 \sim G_1$ or not. If we randomly pick $H \in \mathcal{G}_n$, then there is some constant c such that $\Pr[H \in \text{LIKE}'(G_0, G_1)]$ is c or $2c$, depending on whether $G_0 \sim G_1$ or not. This probabilistic may be the basis for recognizing NONISO. However, the constant c is exponentially small in n since $|\mathcal{G}_n| = 2^{\binom{n}{2}}\, n!$. We need to modify this gap so that the constant c does ot depend on n. The idea is to map $\mathcal{G}_n$ into a smaller set of size $\Theta(n!)$. The following technical lemma will be useful. To avoid interrupting the flow, we leave the proof to an exercise.

Lemma 12 (Boppana) *Let B be a $n \times m$ Boolean matrix, and let*

$$h_B : \{0,1\}^m \to \{0,1\}^n$$

be defined by $h_B(x) = B \cdot x$ where all arithmetic is modulo 2. A **random linear function** *h_B is obtained randomly and independently selecting the entries of B. Let $C \subseteq \{0,1\}^m$ and $h_B(C) = \{h_B(x) : x \in C\}$. If $c = |C|/2^m \leq 1$ and z is randomly chosen from $\{0,1\}^n$, then*

$$\Pr[z \in h(C)] \geq c - \frac{c^2}{2}.$$

The next idea is to get rid of the assumption that G_i has only the trivial automorphism. We now define $\text{LIKE}(G_0, G_1)$ by tagging each member H of $\text{LIKE}'(G_0, G_1)$ with an automorphism:

$$\text{LIKE}(G_0, G_1) = \{\langle H, \pi\rangle : \pi \text{ is an automorphism of } H, H \sim G_0 \text{ or } \ H \sim G_1\}.$$

Using some elementary facts of group theory, we may verify (Exercise) that $|\text{LIKE}(G_0, G_1)| = n!$ or $2n!$.

We are now ready for the proof of the following result:

Theorem 13 $\text{NONISO} \in AM[2]$.

Proof. Let us assume each element of $\mathcal{G}_n$ is given by a string in $\{0,1\}^{\binom{n}{2}}$. Let $p(n) = \lceil \lg(n!) \rceil + 2$, z be a random element of $\{0,1\}^{p(n)}$ and B be a random $\binom{n}{2} \times p(n)$ Boolean matrix. If $G_0 \not\sim G_1$, then by lemma 12,

$$\Pr[z \in h_B(\text{LIKE}(G_0, G_1))] \geq c(1 - \frac{1}{2})$$

where $c = 2n!/2^{p(n)}$. Since $\frac{1}{4} < c \leq \frac{1}{2}$, we conclude that

$$\Pr[z \in h_B(\text{LIKE}(G_0, G_1))] \geq 3c/4.$$

However, if $G_0 \sim G_1$, then $|h_B(\text{LIKE}(G_0, G_1))| \leq n!$ and

$$\Pr[z \in h_B(\text{LIKE}(G_0, G_1))] \leq \frac{n!}{2^{p(n)}} = c/2.$$

This gives rise to the following $AM[2]$-game:

INPUT: $\langle G_0, G_1 \rangle \in \mathcal{G}_n^2$.
1. Randomly choose B and z, as above.
2. Existentially choose a bit b, $H \in \mathcal{G}_n$ and two n-permutations π, σ.
3. Answer YES iff $H^\pi = H$ and $H^\sigma = G_b$.

This game has probability gap $[c/2, 3c/4]$. We have to modify the game so that this gap into one that is an error gap, *i.e.*, centered about $\frac{1}{2}$. Note that for each n, we can determine a suitable value $e = e(n)$ $(0 < e < 1)$ such that $[c/2 \oplus e, 3c/4 \oplus e]$ is an error gap. By majority vote, we can boost this error gap to $[1/3, 2/3]$. We leave the details to an exercise. **Q.E.D.**

Note that using (8.7) and the original interactive verifier V_0, we can infer that NONISO $\in AM[4]$. Hence this proof yields a stronger result. A survey on interactive proofs may be found in [23].

8.5 Markov Chains and Space-bounded Computation

We want to study computations by space-bounded probabilistic machines. The behavior of such computations can be analyzed in terms of finite[7] *Markov chains*. We develop the needed results on Markov chains (see also the appendix in this chapter). For further reference on Markov chains, see [16, 10].

The main result of this section is

Theorem 14 *For all $s(n) \geq \log n$,*

$$PrSPACE(s) \subseteq DSPACE(s^2)$$

Notice that this result is yet another strengthening of Savitch's theorem! We follow the proof of Borodin, Cook and Pippenger [5]; Jung [15] independently obtained the same result using different techniques[8]. This result improves earlier simulations by Gill [12] (which uses exponential space) and by J. Simon [27] (which uses $s(n)^6$ space).

A sequence of non-negative real numbers $(p_1, p_2, \ldots, p_i, \ldots)$ is *stochastic* if the sum $\sum_{i \geq 1} p_i = 1$; it is *substochastic* if $\sum_{i \geq 1} p_i \leq 1$. A matrix is stochastic (resp., substochastic) if each row is stochastic (substochatic). In general, stochastic sequences and stochastic matrices may be denumerably infinite although we will only consider finite matrices. An n-state **Markov process** (or *Markov chain*) is characterized by an $n \times n$ stochastic matrix $A = (p_{i,j})_{i,j=1}^n$. Call A the **transition matrix**. The states of A will be called **Markov states**, as distinguished from machine states. We interpret this chain as an n-state finite automaton where $p_{i,j}$ is the probability of

[7] *i.e.*, discrete time, homogeneous Markov processes, with finitely many states.

[8] Borodin, Cook and Pippenger uses redundant arithmetic techniques while Jung uses modular arithmetic techniques. Borodin, Cook and Pippenger states the result in a stronger form (in terms of circuit depth, see chapter 10), but it has essentially the proof to be presented here.

going from state i to state j. For any integer $k \geq 0$, the kth power $A^k = (p_{i,j}^{(k)})_{i,j}^n$ of A is defined inductively: A^0 is the identity matrix and $A^{k+1} = A \cdot A^k$. It is easy to check the product of stochastic matrices is stochastic; hence each A^k is stochastic. Clearly $p_{i,j}^{(k)}$ denotes the probability of a transition from state i to state j in exactly k steps.

Markov states admit a straight forward combinatorial classification. From the transition matrix $A = (p_{i,j})_{i,j=1}^n$ of the Markov chain, construct the Boolean matrix $B = (b_{i,j})_{i,j=1}^n$ where $b_{i,j} = 1$ iff $p_{i,j} > 0$. We view B as the adjacency matrix of a directed graph G_A, called the *underlying graph* of the matrix A. We may form the transitive closure $B^* = (b_{i,j}^*)_{i,j=1}^n$ of B (see chapter 2, section 6). As usual, define states i and j to be *strongly connected* if

$$b_{i,j}^* = b_{j,i}^* = 1.$$

This is easily seen to be an equivalence relationship and the equivalence classes form the *(strongly connected) components* of G_A. These strongly connected components in turn are related by the *reachability relation*: if C and C' are components, we say C *can reach* C' if there are states $i \in C$ and $j \in C'$ such that $b_{i,j}^* = 1$. It is easy to see that this definition does not depend on the choice of i and j. Furthermore, if C can reach C' and C' can reach C then $C = C'$. Thus the reachability relation induces an acyclic graph F on the components where F has an edge from C to C' iff C can reach C'. Those components C that cannot reach any other components are called *essential components* and the states in them known as *essential states*. The other components are called *inessential components* and their members known as *inessential states*.[9] We say state i is *absorbing* if $p_{i,i} = 1$. Such a state is clearly essential and forms a component by itself. A Markov chain is *absorbing* if all essential states are absorbing.

The above classification depends only on the underlying graph G_A. Let us now classify states by their stochastic properties. These notions properly belong to a subarea of probability theory called renewal theory. We introduce an important concept in renewal theory: let $f_{i,j}^{(n)}$ denote the probability that, starting from state i, we enter state j for the first time after n steps. We call these $f_{i,j}^{(n)}$ the *first entrance probabilities*. Write $f_i^{(n)}$ for $f_{i,i}^{(n)}$. It is not hard to see that for $n = 1, 2, \ldots$,

$$f_{i,j}^{(n)} = p_{i,j}^{(n)} - \sum_{k=1}^{n-1} f_{i,j}^{(k)} p_{j,j}^{(n-k)}$$

[9]The reader should be aware that the classification of Markov states are not all consistent in the literature. The essential/inessential distinction is due to Chung [7]. His terminology is justified in the sense that every chain has at least one essential component; but it also seems to reflect an attitude in probabilistic theory that the most interesting phenomena occur in the essential components. This is unfortunate because we will see that the inessential components are more interesting for us!

or,

$$p_{i,j}^{(n)} = \sum_{k=0}^{n} f_{i,j}^{(k)} p_{j,j}^{(n-k)} \tag{8.8}$$

where we conventionally take

$$f_{i,j}^{(0)} = 0, \quad p_{i,j}^{(0)} = \delta_{i,j}.$$

Here $\delta_{i,j}$ is Kronecker's delta function that assumes a value of 1 or 0 depending on whether $i = j$ or not. The sum

$$f_{i,j}^{*} = \sum_{n=1}^{\infty} f_{i,j}^{(n)}$$

clearly denotes the probability of ever reaching state j from i. Let f_i^* abbreviate $f_{i,i}^*$. We now define a state to be *recurrent* if $f_i^* = 1$ and *nonrecurrent* if $f_i^* < 1$.

Lemma 15 *An inessential state is nonrecurrent.*

Proof. By definition, if state i is inessential, there is a finite path from i to some state outside the component of i. Then $f_i^* \leq 1 - c$ where $c > 0$ is the probability of taking this path.
Q.E.D.

The converse does not hold in general (Appendix and Exercise). But in the case of finite Markov chains, essential states are recurrent. To show this result, we proceed as follows: let $g_{i,j}^{(n)}$ denote the probability of the event $G_{i,j}^{(n)}$ that starting from state i we will visit state j at least n times. Note that $G_{i,j}^{(n+1)} \subseteq G_{i,j}^{(n)}$ and so we may define the limit

$$g_{i,j} := \lim_{n\to\infty} g_{i,j}^{(n)} = \Pr(\bigcap_{n=0}^{\infty} G_{i,j}^{(n)}).$$

It is not hard to see that $g_{i,j}$ is the probability that starting from state i we visit state j infinitely often. Again, let $g_{i,i}$ be abbreviated to g_i.

Lemma 16
(i) $g_i = 1$ or 0 according as i is recurrent or not.
(ii) In a finite Markov chain, essential states are recurrent.

Proof. (i) Note that

$$g_i^{(n+1)} = f_i^* g_i^{(n)}.$$

Since $g_i^{(1)} = f_i^*$, we get inductively

$$g_i^{(n+1)} = (f_i^*)^n.$$

Taking limits as $n \to \infty$, we see that $g_i = 1$ if $f_i^* = 1$ and $g_i = 0$ if $f_i^* < 1$.
(ii) Let $E_i^{(n)}$ be the event that starting from state i, there are no returns to state i after n steps. Clearly

$$E_i^{(1)} \subseteq E_i^{(2)} \subseteq \cdots$$

and $E_i := \bigcup_{n \geq 0} E_i^{(n)}$ is the event that there are only finitely many returns. But

$$\Pr(E_i^{(n)}) \leq 1 - e$$

where $e > 0$ is the minimum probability that any state in the component of i can get to state i. (To see this,

$$\Pr(E_i^{(n)}) = \sum_j p_{i,j}^{(n)}(1 - f_{j,i}^*) \leq (1 - e) \sum_j p_{i,j}^{(n)}$$

which is at most $1 - e$.) Hence $\Pr(E_i) \leq 1 - e < 1$. But $g_i = 1 - \Pr(E_i)$. Hence $g_i > 0$ and by part (i), $g_i = 1$. This means state i is recurrent. **Q.E.D.**

We now see that for finite Markov chains, the combinatorial classification of essential/inessential states coincides with the stochastic classification of recurrent/nonrecurrent states. The appendix describe some refined classifications.

The *stochastic completion* of $A = (p_{i,j})_{i,j=1}^n$ is the matrix $A^* = (p_{i,j}^*)_{i,j=1}^n$ where

$$p_{i,j}^* = \sum_{k=0}^{\infty} p_{i,j}^{(k)}$$

with the understanding that the sum is ∞ when it diverges. The completion operation is defined even if A is not a stochastic matrix.[10]

The entries of A^* has this natural interpretation:

Lemma 17
(i) $p_{i,j}^*$ is the expected number of steps that the automaton spends in state j if it started out in state i.
(ii) Furthermore, if j cannot return to itself in one or more steps then $p_{i,j}^*$ is the probability that the automaton ever enters state j.

Proof. Interpretation (i) follows when we note $p_{i,j}^{(n)}$ is the expected fraction of time that the automaton spends in state j during nth unit time period, assuming that it started out in state i. For (ii), under the stated assumptions on state j, we see that $p_{i,j}^{(n)} = f_{i,j}^{(n)}$ and hence $p_{i,j}^* = f_{i,j}^*$. **Q.E.D.**

[10]The terminology is from in [5]. The notation $p_{i,j}^*$ is not to be confused with the limiting value of $p_{i,j}^{(k)}$ as $k \to \infty$. Unfortunately, a stochastic completion is no longer a stochastic matrix. This is obvious from the interpretation of $p_{i,j}^*$ as the expected number of steps in state j.

Let us introduce the following generating functions (see appendix) for state i:

$$\begin{aligned} F_i(s) &:= \sum_{n=0}^{\infty} f_i^{(n)} s^n \\ G_i(s) &:= \sum_{n=0}^{\infty} p_i^{(n)} s^n \end{aligned}$$

Using the relation (8.8), we see that

$$G_i(s) - 1 = F_i(s) G_i(s)$$

or,

$$G_i(s) = \frac{1}{1 - F_i(s)}$$

Now if we take the limit as $s \to 1^-$, the left hand side approaches $p_{j,j}^*$ and the right hand side approaches $\frac{1}{1-f_j^*}$. This proves

Lemma 18

$$p_{j,j}^* < \infty \iff f_j^* < 1.$$

To relate this to other values of $p_{i,j}^*$, we have

Lemma 19

$$p_{i,j}^* = \delta_{i,j} + f_{i,j}^* p_{j,j}^*.$$

Proof.

$$\begin{aligned} p_{i,j}^* &= \sum_{n=0}^{\infty} p_{i,j}^{(n)} \\ &= \delta_{i,j} + \sum_{n=1}^{\infty} \sum_{k=1}^{n} f_{i,j}^{(k)} p_{j,j}^{(n-k)} \\ &= \delta_{i,j} + \sum_{k=1}^{\infty} f_{i,j}^{(k)} \sum_{n=k}^{\infty} p_{j,j}^{(n-k)} \\ &= \delta_{i,j} + \sum_{k=1}^{\infty} f_{i,j}^{(k)} \sum_{n=0}^{\infty} p_{j,j}^{(n)} \\ &= \delta_{i,j} + f_{i,j}^* p_{j,j}^* \end{aligned}$$

Q.E.D.

Corollary 20 *For all i, j, if $f^*_{i,j} > 0$ then*

$$p^*_{i,j} < \infty \iff p^*_{j,j} < \infty.$$

We need one more basic fact [16].

Lemma 21 *Let A be a square matrix such that $A^n \to 0$ as $n \to 0$, i.e., each entry of the nth power of A approaches zero as n approaches infinity. Then the matrix $I - A$ is nonsingular where I is the square matrix with the same dimensions as A. Moreover the infinite sum*

$$\sum_{n=0}^{\infty} A^n$$

converges, and this sum is given by

$$(I - A)^{-1} = \sum_{n=0}^{\infty} A^n.$$

Proof. We begin with the identity

$$(I - A)(I + A + A^2 + \cdots + A^n) = I - A^{n+1}.$$

Now the right hand side approaches I for large n. So for sufficiently large n, $\det(I - A^{n+1}) \neq 0$. This means $\det(I - A)\det(I + A + \cdots + A^n) \neq 0$. Thus $I - A$ is nonsingular, as asserted. So we may multiply both sides of the identity on the left with $(I - A)^{-1}$, giving a new identity

$$(I + A + A^2 + \cdots + A^n) = (I - A)^{-1}(I - A^{n+1}).$$

Now as $n \to \infty$, the left hand side approaches the infinite sum of the lemma and the right hand side approaches $(I - A)^{-1}$. Since the right hand side approaches a definite limit, so the left hand side approaches the same limit. **Q.E.D.**

For any transition matrix A, let B be obtained by deleting the rows and columns of A corresponding to essential states. Following Kemeny and Snell, we call B the *fundamental part* of A. Note that B is a substochastic matrix. Then after permuting the rows and columns of A, we have

$$A = \begin{pmatrix} B & T \\ 0 & C \end{pmatrix}$$

where '0' denotes a matrix of zeroes of the appropriate dimensions. Moreover, the nth power of A is

$$A^n = \begin{pmatrix} B^n & T_n \\ 0 & C^n \end{pmatrix}$$

where B^n, C^n are the nth powers of B, C (respectively) and T_n is some matrix whose form need not concern us. Hence, the stochastic completion

$$A^* = \begin{pmatrix} B^* & T_* \\ 0 & C^* \end{pmatrix}$$

where B^*, C^* are the stochastic completions of B, C (respectively). In our applications, we only need B^*. Note that the entries in C^*, T_* are 0 or ∞, by what we have proved.

From the above development, the entries $p_{i,j}^{(n)}$ in B^n satisfy the property $\sum_{n=0}^{\infty} p_{i,j}^{(n)} < \infty$. This means $p_{i,j}^{(n)} \to 0$ as $n \to \infty$. Hence $B^n \to 0$ as $n \to \infty$ and the preceding lemma shows that B^* converges to $(B - I)^{-1}$. Hence computing B^* is reduced to the following:

Theorem 22 *Let A be the transition matrix of an absorbing chain. The stochastic completion of the fundamental part B of A can be computed in deterministic space $\log^2 n$ where B is n by n and each entry of B are rational numbers represented by a pair of n-bit binary number.*

The proof of this theorem requires several preparatory results that are interesting in their own right. Therefore we defer it to the next section. We are now ready to prove the main result (theorem 14) of this section. Although we only compute the fundamental part B^*, with a bit more work, one can compute all the remaining entries of the stochastic closure in the same complexity bounds (see [5]).

Proof of main result (theorem 14). Basically the proof amounts to reducing a space-bounded probabilistic computation to computing the stochastic closure of the fundamental part of an absorbing Markov chain.

Let M be a probabilistic machine accepting in space $s(n)$. We analyze the probability of M accepting an input w by considering the Markov chain whose (Markov) states correspond to those configurations of M on w using space at most $s(|w|)$. We introduce an extra Markov state. Number these Markov states from 1 to r, where we may assume that Markov state 1 is the initial configuration on input w, and r is the extra Markov state (viewed as a NO-configuration of M). The corresponding transition matrix is $A = (p_{i,j})_{i,j=1}^{r}$ where

$$p_{i,j} = \begin{cases} \frac{1}{2} & \text{if configuration } i \text{ non-uniquely derives configuration } j \\ & \quad \text{(i.e., } i \vdash (j,k) \text{ for some } k \neq j) \\ 1 & \text{if either configuration } i \text{ uniquely derives } j, \text{ i.e., } i \vdash (j,j) \\ & \quad \text{or if } i = j \text{ and } i \text{ is terminal} \\ 0 & \text{else.} \end{cases}$$

This is not quite all: how shall we treat state i if $i \vdash (j,k)$ where j and/or k uses more than $s(|w|)$ space? Now assign for such an i, $p_{i,r} = \frac{1}{2}$ or 1, depending on whether one or both of j,k use more than $s(|w|)$ space.

We derive from A an absorbing Markov chain with transition matrix B as follows: say a Markov state in A is **useful** if it is has a path to a YES-state. Clearly useful states are inessential, but some inessential states may not be useful. In B, we retain all the useful states of A and also their transition probabilities among themselves. We renumber the useful Markov states from 1 to some $m-1$ ($m < r$). In addition to the $m-1$ useful states inherited from A, B has two essential states, m and $m+1$. Basically, we collapse all essential YES-states into m and the remaining states in A (essential or not) are collapsed into $m+1$. States m and $m+1$ are both absorbing. More precisely, for each useful state $i = 1, \ldots, m-1$, if the sum of the transition probabilities into the YES-states is p then we set the $(i, m+1)$th entry $[B]_{i,m} := p$. If the sum of the transition probabilities from i to the non-YES and non-useful states is q then we make $[B]_{i,m+1} = q$. Also, we have

$$[B]_{m,m} = [B]_{m+1,m+1} = 1$$

We do one more small transformation: let now C be the matrix that is identical to B except that

$$[C]_{m,m} = 0, \quad [C]_{m,m+1} = 1.$$

So state m is now a transient state. For future reference, call C the *reduced transition matrix* (for input w). If D is the fundamental part of C (obtained by deleting the last row and last column of C) then by theorem 22, we can compute the stochastic completion D^* in $O(\log^2 m)$ space. Now m is $O(1)^{s(|w|)}$ and hence $O(\log^2 m) = O(s^2)$ space suffices.

Our 'interpretation' (lemma 17) of the entries of a stochastic completion suggests that the entry $[D^*]_{1,m}$ is the probability that starting out in state 1 we reach m (since state m cannot return to itself in 1 or more steps, by construction). It is instructive to carry out the proof that $[D^*]_{1,m}$ is indeed the least fixed point value $Val_\Delta(w)$ where Δ is the set $\{1, \ldots, r\}$ of configurations that uses space at most $s(|w|)$. A valuation V on Δ amounts to an r-vector $V = (v_1, \ldots, v_r)$ where v_i is the value of configuration i. (We may assume here that the values v_i are real numbers in $[0,1]$ rather than intervals.) The valuation operator τ_Δ is the linear transformation given by the transition matrix A, and $\tau_\Delta(V)$ is equal to $A \cdot V^T$ (V^T is the column vector obtained by transposing V). Let $V_0 = \tau_\Delta(V_\perp)$ be the row vector that assigns a 1 to the YES state and a zero to the NO state. (Note that V_0 is not stochastic in general.) The valuation $\tau^n_\Delta(V_0)$ is given by $V_n = A^n \cdot V_0^T$. We conclude: $Val_\Delta(w)$ *is the limiting value of the first component of* $A^n \cdot V_0^T$, *as* $n \to \infty$. Alternatively, if the set of YES-configurations are $S \subseteq \Delta$, then $Val_\Delta(w)$ is the limiting value of $\sum_{i \in S} [A^n]_{1,i}$.

It is not hard to see that our transformation of A to B does no harm and we have a slightly simpler picture: $Val_\Delta(w)$ is given by the limiting value of $[B^n]_{1,m}$

(Exercises).

The transformation from B to C is less obvious. Let us compare their nth powers, B^n and C^n, for each $n \geq 0$. The first $m-1$ columns of both powers are seen to be identical. We claim that the first $m-1$ entries in the mth column of B^n is equal to the corresponding entry in the sum $\sum_{i=1}^{n} C^i$: *for each* $j = 1, \ldots, m-1$, *and for all* $n \geq 0$, $[B^n]_{j,m} = \sum_{\ell=1}^{n} [C^\ell]_{j,m}$. In proof, this is clearly true for $n = 1$. For $n \geq 1$, we have

$$\begin{aligned}
[B^{n+1}]_{j,m} &= \sum_{k=1}^{m} [B^n]_{j,k}[B]_{k,m} \\
&= [B^n]_{j,m}[B]_{m,m} + \sum_{k=1}^{m-1} [B^n]_{j,k}[B]_{k,m} \\
&= [B^n]_{j,m} + \sum_{k=1}^{m-1} [C^n]_{j,k}[C]_{k,m} \\
&= [B^n]_{j,m} + [C^{n+1}]_{j,m} \\
&= \sum_{\ell=1}^{n+1} [C^\ell]_{j,m}
\end{aligned}$$

This essentially gives us the theorem.

There are several other details that we must defer to the next section: in particular we cannot afford to explicitly store the matrices A or C. Instead, they are represented 'procedurally' in the sense that each entry of such matrices can be obtained by invoking suitable subroutines. For instance, this means that our 'application' of theorem 22 is really not in the form as stated. We need to show that the techniques implicit in that theorem can be modified to accomodate the implicit representation of C. Another clarification is needed: to form matrix C, we need to determine the useful states in A (one efficient way to detect such states uses the the original Savitch's theorem technique). Modulo these details, we are done with the proof. **Q.E.D.**

8.6 Efficient Circuits for Ring Operations

By an *algebraic structure* we means a set A together with a finite set of partial functions f_i $(i = 1, \ldots, k)$ of the form

$$f_i : A^{\alpha(i)} \to A$$

where $\alpha(i) \geq 0$ are integers called the *arity* of f_i. We call f_i a *constant* if $\alpha(i) = 0$ and in this case, f_i is identified with an element of A. We write $(A; f_1, \ldots, f_k)$ for the algebraic structure. This is abbreviated to 'A' when the functions f_i are understood. In general, by an *operation* f *over an algebraic structure* A we mean a partial function $f : A^m \to A$ for some $m \geq 0$, where m is called the arity of f.

Example 2 a) Of course, for any set A, there is the trivial algebraic structure on A which no functions at all.
b) The integers $\mathbb{Z} = \{0, \pm 1, \pm 2, \ldots\}$ with the usual operations ($+$, $-$, $\times$ and $0, 1$) is an algebraic structure. It is, in fact, a unitary commutative ring (see below).
c) The rational numbers $\mathbb{Q}$, with the operations of $\mathbb{Z}$ but also including the division $\div$ operation, is an algebraic structure called a commutative field. Here division is a partial function since division by zero is not defined.
d) The set of all n-square matrices with entries from $\mathbb{Q}$ forms a matrix ring with the usual matrix operations of $+$, $-$ and $\times$.
e) Consider the Boolean algebra on two elements $(\{0, 1\}; \vee, \wedge, \neg, 0, 1)$ where $\vee, \wedge, \neg$ are interpreted as the usual Boolean operations.
f) A class of finite rings is $\mathbb{Z}_n = \{0, 1, \ldots, n-1\}$ with usual arithmetic operations modulo n. In case n is a prime, $\mathbb{Z}_n$ is a field, also called $GF(p)$. The case $GF(2)$ has special interest. ■

We are mainly interested in computing over various unitary commutative rings R, henceforth simply called 'rings'[11]. Our goal is to show how operations in common rings can be implemented efficiently. A computational model that is appropriate for algebraic structures is circuits.

The following definitions gather most of our notations and terminology related to circuits in one place. It will serve as reference beyond just the immediate concern of this section.

Definition 7 *Let $(A; f_1, \ldots, f_k)$ be an algebraic structure.*

(i) *Any set Ω of operations over A obtained by functional composition from $f_1, \ldots, f_k$ is called a* basis *of A. In general Ω may have infinite cardinality.*

(ii) *A circuit C for A over the basis Ω a finite directed acyclic graph (called the* underlying graph *of C) with the following properties: A node with indegree 0 is called an input node, and if there are n input nodes, we label each with a distinct integer between 1 and n. The remaining nodes are called* gates *and are each labeled by a basis function $f \in \Omega$. If a gate is labeled by f, we say its* type *is f or, equivalently, it is called an f-*gate*. Each f-gate u has indegree exactly equal to the arity $\alpha(f)$ of f. Furthermore the incoming edges to u are*

[11] Commutative rings are simply algebraic structures satisfying certain axioms. The student unfamiliar with rings simply need remember two main examples of such structures given above: the integers $\mathbb{Z}$ and the set of n-square matrices with rational number entries. So a ring comes with the five total operation $+, -, \times, 0, 1$ with the usual properties (inverse relation between plus and minus, associativity, commutativity, distributivity, and properties of 0 and 1) are satisfied. If one writes down these properties, they would constitute an axiomatization of unitary commutative rings (it is a good exercise to try this and compare your results with a standard algebra book). Here 'unitary' serves to to warn that, in general, rings are defined without assuming the existence of element 1.

labeled with distinct integers between 1 and $\alpha(f)$. So we may speak of the jth incoming edge *of u for $j = 1, \ldots, \alpha(f)$.*

(iii) *Each node u of a circuit C is said to* compute *the function*

$$\text{Result}_C(u) : A^n \to A$$

defined as follows: an input node u labeled by i computes the projection function $\text{Result}_C(u)(x_1, \ldots, x_n) = x_i$. *An f-gate u computes the function*

$$\text{Result}_C(u)(\overline{x}) = f(\text{Result}_C(u_1)(\overline{x}), \ldots, \text{Result}_C(u_m)(\overline{x}))$$

where $\overline{x} = (x_1, \ldots, x_n)$ and the ith incoming edge of u leads from node u_j ($j = 1, \ldots, m$), m is the arity of f.

(iv) *A* circuit family *(over the basis Ω) is an infinite sequence of circuits $\overline{C} = (C_n)_{n=0}^{\infty}$ such that each C_n is an n-input circuit over Ω.*

(v) *A* problem instance of size n *(over A) is a set P_n of functions $g : A^n \to A$ (so each $g \in P_n$ has arity n). An* aggregate problem *$P = (P_n)_{n=0}^{\infty}$ over A is an infinite sequence of problem instances P_n, each P_n of size n. When no confusion arises, we may omit the qualification 'aggregate'. Often, $P_n = \{f_n\}$ is a singleton set, in which case we simply write $P = (f_n)_{n \geq 0}$.*

(vi) *Let P_n be a problem instance of size n over A. A circuit C over A is said to* solve *or* realize *P_n if if C has n inputs and for each $g \in P_n$, there is a node $u \in C$ such that $\text{Result}_C(u) = g$. A circuit family $\overline{C} = (C_n)_{n=0}^{\infty}$ is said to* solve *a problem $P = (P_n)_{n=0}^{\infty}$ if C_n solves P_n for each n.*

(vii) *The size of a circuit C is the number of gates in the circuit*[12]*. The* size *of a circuit family $\overline{C} = (C_n)_{n=0}^{\infty}$ is the function $SIZE_{\overline{C}}$ where $SIZE_{\overline{C}}(n)$ is the size of C_n.*

(viii) *Two other complexity measures for circuits are as follows: the* depth *of C is the length of the longest path in C. The* width *of C is the maximum cardinality of an edge anti-chain*[13] *in C. As for the size-measure, we let $DEPTH_{\overline{C}}(n)$ and $WIDTH_{\overline{C}}(n)$ denote the depth and width of C_n, where $\overline{C} = (C_n)_{n \geq 0}$.*

(ix) *For any problem instance P_n, let $SIZE(P_n)$ denote the smallest sized circuit that realizes P_n. If $P = (P_n)_{n \geq 0}$ is an aggregate problem, the size function $SIZE_P$ is the function given by $SIZE_P(n) = SIZE(P_n)$. Similarly for*

$$DEPTH(P_n), WIDTH(P_n), DEPTH_P, WIDTH_P.$$

[12] It is unfortunate that we have to use the word 'size' for problem instances as well as for circuits, both of which appear in the same context. Since the usage is well accepted and there seems to be no better alternative, we will continue this usage. But for emphasis, we could say 'circuit size' or 'problem size'.

[13] An *edge anti-chain* in a directed acyclic graph is a set of edges such that no two edge in the set belongs to a common path. Of course, one can define *node anti-chain* as well.

(x) *For any complexity function $f(n)$, let $SIZE(f)$ denote the family of aggregate problems $\{P : \forall n, SIZE_P(n) \leq f(n)\}$. Similarly for $DEPTH(f), WIDTH(f)$. We can extend this to simultaneous measures, for instance $SIZE - DEPTH - width(f_1, f_2, f_3)$.*

(xi) *For any non-negative integer k, a circuit family $\overline{C}$ is said to be NC^k if $SIZE_{\overline{C}}(n) = n^{O(1)}$ and $DEPTH_{\overline{C}}(n) = O(\log^k n)$. An aggregate problem is said to be NC^k if it can be realized by an NC^k circuit family.*

■

Remark: We are often interested in problems for which there is really no problem instances of size n for certain values of n (e.g., multiplying square Boolean matrices only has interesting input sizes of the form $2n^2$). In these cases, we artificially create the trivial problem instance of size n, such as the identically zero function of arity n. Also, the above definition of circuits do not allow constant values as inputs. The definition can trivially be changed to accommodate this.

We are mainly interested in circuits for two types of algebraic structures: (a) where A is a ring and (b) where $A = \{0, 1\}$ is the Boolean algebra in the above example. We call a circuit for A an *arithmetic circuit* or a *Boolean circuit* in cases (a) or (b), respectively.

8.6.1 The Parallel Prefix Problem.

We begin with a basic but important technique from Ladner and Fischer [18] for the so-called *parallel prefix problem*. In this problem, we assume that we are given an algebraic structure $(A; \circ)$ where the only operation $\circ$ is a binary associative operation (which we will call 'multiplication' or 'product'). As is usual with multiplicative notations, when convenient, we may write xy and $\prod_{i=1}^{n} x_i$ (respectively) instead of $x \circ y$ and $x_1 \circ x_2 \circ \cdots \circ x_n$. We call a circuit over such an A a *product circuit*. The parallel prefix problem instance (of size $n \geq 0$) amounts to computing the set of n functions

$$f_i(x_1, \ldots, x_n) := x_1 \circ x_2 \circ \cdots \circ x_i, \quad \text{for each } i = 1, \ldots, n.$$

We may call these f_i's the set 'iterative-$\circ$ functions' (on n variables). We shall apply this in two cases: where $\circ$ is addition and where $\circ$ is multiplication in a ring. Then, we call parallel prefix the *iterative addition* and *iterative multiplication* problems, respectively.

Lemma 23 *There is a recursive construction of a family of product circuits $(C_n)_{n=0}^{\infty}$ of linear size and logarithmic depth that solves the parallel prefix problem.*

Proof. We may assume that n is a power of 2. C_1 is trivial, consisting of just an input node for x_1. So let $n > 1$. The following figures shows the recursive construction of C_n from $C_{n/2}$:

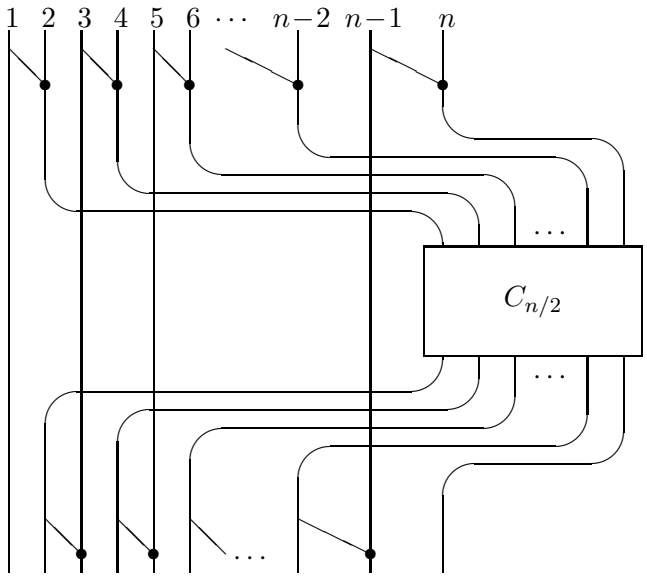


Figure 8.1 Construction of C_n ('•' indicates a gate)

In this figure, the edges of the circuit are implicitly directed downwards. It is easy to see that the size $s(n)$ of a circuit C_n satisfies the relation $s(1) = 1$ and $s(n) = s(n/2) + n - 1$. The solution is $s(n) = 2n - \log n - 2$. The depth of C_n is also easily seen to be $2 \log n$. **Q.E.D.**

8.6.2 Detecting Useless Markov States.

Recall in the previous section, in the context of a Markov chain whose nodes are machine configurations, we define a Markov state to be useless if it cannot reach a YES-configuration. To detect such useless states, we can formulate the following general problem: given the adjacency matrix of a digraph G, we want to determine its "useless nodes", defined to mean that those nodes that cannot reach a distinguished node in G.

Lemma 24 *There is an NC^2 Boolean circuit family which computes, on any input n by n matrix A_n which represents the adjacency matrix of directed graph, the set of 0/1-functions $\{f_i : i = 1, \ldots, n\}$ where $f_i(A_n) = 1$ iff i cannot reach node n.*

Proof. This is rather straight forward: we can compute the products of Boolean matrices in NC^1. The transitive closure of A_n is given by $(A_n)^m$ for any $m \geq n$. By the usual doubling method, the transitive closure is obtained by $\log n$ matrix multiplications, hence by NC^2 circuits. Finally a node i can reach node n if and only if $A^*(i, n) = 1$. **Q.E.D.**

8.6.3 Computing the characteristic polynomial.

The determinant of an $n \times n$ matrix $A = (a_{i,j})$ can be expanded by its ith row (for any i) in the standard fashion:

$$\det(A) = a_{i,1}D_{i,1} + a_{i,2}D_{i,2} + \cdots + a_{i,n}D_{i,n}$$

where $(-1)^{i+j}D_{i,j}$ is the determinant[14] of the $(n-1)$-square matrix obtained by deleting the ith row and the jth column of A. $D_{i,j}$ is called the (i,j)-*cofactor* (or (i,j)-*complement*) of A. Let the *adjoint* $\mathrm{adj}(A)$ of A be the $n \times n$ matrix whose (i,j)th element $[\mathrm{adj}(A)]_{i,j}$ is the (j,i)-cofactor of A (notice the transposition of subscripts i and j). It is not hard to see that[15] that the following is valid for all A:

$$A \cdot \mathrm{adj}(A) = \mathrm{adj}(A) \cdot A = \det(A) \cdot I$$

(We only have to see that the off-diagonal elements of $A \cdot \mathrm{adj}(A)$ are of the form

$$a_{i,1}D_{j,1} + a_{i,2}D_{j,2} + \cdots + a_{i,n}D_{j,n}$$

where $i \neq j$. But this sum is also seen to be zero since it is the determinant of the singular matrix obtained by replacing the jth row of A with the ith row. On the other hand, the diagonal entries are equal to $\det(A)$, which may be zero if A is singular.) The *characteristic polynomial* $P_A(x)$ of A is the determinant

$$\begin{aligned} P_A(x) &:= \det(xI_n - A) \\ &= x^n + \lambda_1 x^{n-1} + \cdots + \lambda_{n-1}x + \lambda_n \end{aligned}$$

where I_n is the $n \times n$ identity matrix. (We will omit the subscript in I_n when convenient.) A fundamental identity is the Cayley-Hamilton theorem (Exercises) that states that A is a root of the polynomial $P_A(x)$

$$P_A(A) = A^n + \lambda_1 A^{n-1} + \cdots + \lambda_{n-1}A + \lambda_n I = 0.$$

Our goal is to develop a space-efficient algorithm for computing the characteristic polynomial. To compute the characteristic polynomial of A means to determine the above coefficients $\lambda_1, \ldots, \lambda_n$. Note that this computation is a generalization of the problem of computing determinant since the constant term in P_A is (up to sign) equal to $\det(A)$. We present an efficient parallel implementation of Samuelson's method[16] by Berkowitz [3]. For this, we use the notation $red(A)$ ('reduction' of A) which is the $(n-1)$-square matrix obtained from A by deleting the first row and

[14] This determinant is called the (i,j)-*minor* of A

[15] See [11] for most basic results on matrices.

[16] The importance of this method (as opposed to an earlier method of Csanky that has comparable complexity) is that it uses no divisions, so it is applicable to any unitary commutative ring.

first column. Define the column $(n-1)$-vector $\mathrm{col}(A)$ and row $(n-1)$-vector $\mathrm{row}(A)$ so that

$$A = \begin{pmatrix} a_{1,1} & \mathrm{row}(A) \\ \mathrm{col}(A) & \mathrm{red}(A) \end{pmatrix}$$

Lemma 25 *The characteristic polynomial of A is related to the matrix $\mathrm{red}(A)$ as follows:*

$$P_A(x) = (x - a_{1,1}) \det(xI_{n-1} - \mathrm{red}(A)) - \mathrm{row}(A) \cdot \mathrm{adj}(xI_{n-1} - \mathrm{red}(A)) \cdot \mathrm{col}(A). \quad (8.9)$$

Proof.

$$\begin{aligned} P_A(x) &= \det(xI_n - A) \\ &= (x - a_{1,1}) \det(xI_{n-1} - A) + \sum_{j=2}^{n} a_{1,j} D_{1,j} \end{aligned}$$

where $D_{1,j}$ is the $(1,j)$-cofactor of $xI_n - A$. Write

$$D[i_1, i_2, \ldots; j_1, j_2, \ldots]$$

for the determinant of the square matrix obtained by deleting rows $i_1, i_2, \ldots$ and columns $j_1, j_2, \ldots$. The lemma follows from the following identity

$$\begin{aligned} \sum_{j=2}^{n} a_{1,j} D_{1,j} &= \sum_{j=2}^{n} a_{1,j} (-1)^{1+j} D[1; j] \\ &= \sum_{j=2}^{n} a_{1,j} (-1)^{1+j} \sum_{i=2}^{n} a_{i,1} (-1)^{i} D[1, i; 1, j] \\ &= -\sum_{j=2}^{n} \sum_{i=2}^{n} a_{1,j} a_{i,1} (-1)^{i+j} D[1, i; 1, j] \\ &= -\sum_{j=2}^{n} \sum_{i=2}^{n} a_{1,j} a_{i,1} D'_{i,j} \end{aligned}$$

where $D'_{i,j}$ is the $(i-1, j-1)$-cofactor of $\mathrm{red}(xI_n - A) = xI_{n-1} - \mathrm{red}(A)$. Since $D'_{i,j}$ is the $(j-1, i-1)$th entry of $\mathrm{adj}(xI_{n-1} - \mathrm{red}(A))$, the lemma follows. **Q.E.D.**

The adjoint of $xI - A$ can be expressed with the help of the the next lemma.

Lemma 26

$$\mathrm{adj}(xI - A) = \sum_{i=0}^{n-1} B_i x^{n-1-i} \quad (8.10)$$

where

$$B_i = A^i + \lambda_1 A^{i-1} + \cdots + \lambda_{i-1} A + \lambda_i I$$

and λ_i are the coefficients of the characteristic polynomial

$$P_A(x) = x^n + \lambda_1 x^{n-1} + \cdots + \lambda_{n-1}x + \lambda_n.$$

Proof. We observe that $B_0 = I_n$ and for $i = 1, \ldots, n$,

$$B_i = AB_{i-1} + \lambda_i I.$$

Hence

$$\begin{aligned}
&\det(xI - A) \cdot I \\
&= [x^n + \lambda_1 x^{n-1} + \cdots + \lambda_{n-1}x + \lambda_n] \cdot I \\
&= [x^{n-1}B_0](xI - A) + x^{n-1}AB_0 + [\lambda_1 x^{n-1} + \lambda_2 x^{n-2} + \cdots + \lambda_{n-1}x + \lambda_n] \cdot I \\
&= [x^{n-1}B_0 + x^{n-2}B_1](xI - A) + x^{n-2}AB_1 + [\lambda_2 x^{n-2} + \lambda_3 x^{n-3} + \cdots + \lambda_n] \cdot I \\
&= \cdots \\
&= \left[\sum_{i=0}^{n-1} x^{n-1-i}B_i\right](xI - A) + x^0 AB_{n-1} + \lambda_n I \\
&= \left[\sum_{i=0}^{n-1} x^{n-1-i}B_i\right](xI - A)
\end{aligned}$$

where the last equality follows from

$$x^0 AB_{n-1} + \lambda_n I = B_n = P_A(A) = 0$$

by the Cayley-Hamilton theorem. On the other hand $\det(xI - A) \cdot I$ can also be expressed as

$$\det(xI - A) \cdot I = \mathrm{adj}(xI - A) \cdot (xI - A).$$

Since $xI - A$ is nonsingular (x is a indeterminate), we can cancel $xI - A$ as a factor from the two expressions for $\det(xI - A) \cdot I$, giving the desired equality for the lemma. **Q.E.D.**

The last two lemmas show that the characteristic polynomial $P_A(x)$ of A can be computed from the characteristic polynomial $P_{\mathrm{red}(A)}(x)$ of $\mathrm{red}(A)$ and the matrix products $\mathrm{red}(A)^i$ (for $i = 1, \ldots, n-1$). Let us derive this precisely. Let the coefficients of $P_{\mathrm{red}(A)}(x)$ be given by μ_i $(i = 0, \ldots, n-1)$ where $P_{\mathrm{red}(A)}(x) = \sum_{i=0}^{n-1} \mu_{n-1-i}x^i$. Then we see that

$$\begin{aligned}
P_A(x) &= (x - a_{1,1})P_{\mathrm{red}(A)}(x) - \mathrm{row}(A) \cdot \left(\sum_{i=0}^{n-1} B_i x^{n-1-i}\right) \cdot \mathrm{col}(A) \\
&= (x - a_{1,1}) \sum_{i=0}^{n-1} \mu_i x^{n-1-i} + \sum_{i=0}^{n-1} x^{n-1-i}\mathrm{row}(A) \cdot \left(\sum_{j=0}^{i} (\mathrm{red}(A))^j \mu_{i-j}\right) \cdot \mathrm{col}(A)
\end{aligned}$$

$$\begin{aligned} &= \sum_{i=0}^{n-1} \mu_i x^{n-i} + \sum_{i=0}^{n-1} x^{n-1-i} \left(-a_{1,1}\mu_i + \sum_{j=0}^{i} \beta_j \mu_{i-j} \right) \\ &\qquad (\text{where } \beta_j = \text{row} \cdot (A)(\text{red}(A))^j \cdot \text{col}(A)) \\ &= \sum_{j=0}^{n} \lambda_{n-j} x^j \end{aligned}$$

where

$$\lambda_j = \begin{cases} \mu_0 & \text{if } j = 0 \\ \mu_j - a_{1,1}\mu_{j-1} + \sum_{k=0}^{j-1} \beta_k \mu_{j-1-k} & \text{if } j = 1, \ldots, n-1 \\ -a_{1,1}\mu_{n-1} + \sum_{k=0}^{n-1} \beta_k \mu_{n-1-k} & \text{if } j = n \end{cases}$$

We can write this in matrix form. Let C_0 be the following $(n+1) \times n$ matrix:

$$C_0 := \begin{pmatrix} 1 & & & & & & \\ \beta_0 - a_{1,1} & 1 & & & & & \\ \beta_1 & \beta_0 - a_{1,1} & 1 & & & & \\ \beta_2 & \beta_1 & \beta_0 - a_{1,1} & & & & \\ \vdots & & & \ddots & & & \\ \beta_{n-2} & \cdots & & & \beta_0 - a_{1,1} & 1 & \\ \beta_{n-1} & \beta_{n-2} & \cdots & & & \beta_0 - a_{1,1} & 1 \\ 0 & \beta_{n-1} & \beta_{n-2} & \cdots & & \beta_1 & \beta_0 - a_{1,1} \end{pmatrix}$$

Then we have

$$(\lambda_0, \ldots, \lambda_{n-1}, \lambda_n)^T = C_0 \cdot (\mu_0, \ldots, \mu_{n-1})^T$$

where $(\cdots)^T$ indicates matrix transpose. We repeat this procedure to express the μ_i's in terms of the characteristic polynomial of $\text{red}(\text{red}(A)) = \text{red}^2(A)$, etc. In general, let C_i be the $(n+1-i) \times (n-i)$ matrix that reduces the characteristic polynomial of $\text{red}^i(A)$ to that of $\text{red}^{i+1}(A)$. The entries[17] of C_i consists of 0, 1, the diagonal elements $a_{i+1,i+1}$ of the matrix A, and elements that can be constructed from

$$\text{row}(\text{red}^i(A)) \cdot \text{red}^{i+1}(A) \cdot \text{col}(\text{red}^i(A)).$$

Putting these together, we get

$$(\lambda_0, \ldots, \lambda_n)^T = C_0 C_1 \cdots C_{n-1}$$

Lemma 27 *Given an $n \times n$ matrix A we can construct in deterministic log-space an arithmetic circuit C of depth $O(\log^2 n)$ and size $n^{O(1)}$ such that C computes (the coefficients of) $P_A(x)$.*

[17] In fact C_i has the property that each (j,k)th entry is equal to the $(j+1,k+1)$th entry provided, of course, that the $(j+1,k+1)$th entry is defined. This property is the defining characteristic of *Toeplitz matrices* and it is known that the multiplication of Toeplitz matrices can be done more efficiently than we care to exploit here (Exercises).

Proof. We proceed as follows:

1. We first construct a circuit to compute the set of polynomials

$$\{(\text{red}^i(A))^j : i = 1, \ldots, n-1 \text{ and } j = 1, \ldots, i\}.$$

   Note that to compute a matrix means to compute each of its entries and to compute a polynomial means to compute each of its coefficients. It suffices to show that for each i, we can compute $\{(\text{red}^i(A))^j : j = 1, \ldots, i\}$ with a circuit of polynomial size and depth $O(\log^2 n)$. But this amounts to the parallel prefix computation on i copies of the matrix $\text{red}^i(A)$. Parallel prefix, we saw, uses an $O(\log n)$ depth product circuit. Each gate of the product circuit is replaced by an arithmetic circuit of depth $O(\log n)$ since we can multiply two $n \times n$ matrices in this depth (straightforward). Hence the overall depth is $O(\log^2 n)$.

2. Next, we compute the elements

$$\{\text{row}(\text{red}^{i-1}(A) \cdot (\text{red}^i(A))^j \cdot \text{col}(\text{red}^{i-1}(A) : i = 1, \ldots, n-1 \text{ and } j = 1, \ldots, i\}.$$

   This takes $O(\log n)$ depth. We have now essentially computed the entries of the matrices $C_0, \ldots, C_{n-1}$.

3. We can compute the product $\prod_{i=0}^{n-1} C_i$ using a balanced binary tree of depth $O(\log n)$ to organize the computation. Each level of this binary tree corresponds to the multiplication (in parallel) of pairs of matrices. Since each matrix can be multiplied in $O(\log n)$ depth, the overall circuit depth is $O(\log^2 n)$.

One can easily verify that the circuit is polynomial in size. **Q.E.D.**

8.6.4 Computing the Matrix Inverse

We want to compute the inverse of a matrix A. As before, let

$$P_A(x) = x^n + \lambda_1 x^{n-1} + \cdots + \lambda_{n-1} x + \lambda_n$$

be the characteristic polynomial. Since $P_A(x) = \det(xI - A)$ we see that

$$\lambda_n = P_A(0) = \det(-A) = (-1)^n \det(A).$$

Next, by lemma 26, we have that

$$\begin{aligned} \text{adj}(-A) &= B_{n-1} \\ &= A^{n-1} + \lambda_1 A^{n-2} + \cdots + \lambda_{n-2} A + \lambda_{n-1}. \end{aligned}$$

Note that $\text{adj}(-A) = (-1)^{n-1}\text{adj}(A)$. Putting these together, we get

$$\begin{aligned} A^{-1} &= \frac{1}{\det(A)}\mathrm{adj}(A) \\ &= -\frac{1}{\lambda_n}(A^{n-1} + \lambda_1 A^{n-2} + \cdots + \lambda_{n-2}A + \lambda_{n-1}). \end{aligned}$$

It should now be easy to deduce:

Lemma 28 *We can detect if an n-square matrix A is nonsingular using an NC^2 arithmetic circuit. Moreover, in case A is nonsingular, we can compute its inverse A^{-1} with another NC^2 arithmetic circuit.*

8.6.5 Balanced p-ary Notations

In applying the preceding results, we will use matrices A whose entries are rational numbers. Assuming the usual binary representation of integers, if the integers involved in the computation are k-bits long, then an $O(\log^2 n)$ depth arithmetic circuits translate into Boolean circuits of depth $\Omega(\log^2 n \log k)$, at least (and in our applications $k = \Omega(n)$). To obtain a depth of $O(\log^2 n)$ on Boolean circuits for computing characteristic polynomials, we need one new idea: by a suitable choice of representing integers, we can implement the operations of addition with constant depth (i.e., NC^0) circuits. This in turn yields an improved depth for several other problems. Circuits in this subsection shall mean Boolean circuits unless otherwise stated.

Definition 8
(i) A representation *r of a algebraic structure A is an onto function*

$$r : \{0,1\}^* \to A.$$

*We say $u \in \{0,1\}^*$ is an r-*representative *of $r(u)$.*
(ii) We say an operation $f : A^n \to A$ is in NC^k with respect to r if there is an NC^k Boolean *circuit family $\overline{C} = (C_m)_{m\geq 0}$ such that for each m, C_m computes f in the sense that for all $u_1, \ldots, u_n \in \{0,1\}^m$,*

$$r(C_m(u_1, \ldots, u_n)) = f(r(u_1), \ldots, r(u_n))$$

■

We usually like to ensure that each element can be represented by arbitrarily large binary strings. For instance, as in the binary representation of numbers, we may have the property $r(0u) = r(u)$ for all $u \in \{0,1\}^*$. In other words, we can left pad an representation by 0's. In practice, ring elements have some natural notion of "size" and we may insist that the representation r 'respect' this size function

within some bounds (e.g., $|r(x)| \geq size(x)$). We shall not concern ourselves with such considerations but the interested reader may consult [5].

Our goal is to find a representation of integers so that addition and negation is in NC^0, multiplication and iterated addition is in NC^1 and iterated multiplication is in NC^2. We resort to the *balanced p-ary representation* of integers of Avizienis [1]. Here $p \geq 2$ is an integer and a *balanced p-ary number* is a finite string

$$u_1 u_2 \cdots u_n$$

where u_i is an integer with $|u_i| \leq p-1$. This string represents the integer $(u_1, \ldots, u_n)_p$ given by

$$(u_1, \ldots, u_n)_p := \sum_{i=0}^{n-1} u_{i+1} p^i.$$

So u_1 is the least significant digit. Clearly, the usual p-ary representation of a number is a balanced p-ary representation of the same number. This representation is redundant in a rather strong sense. We will implicitly assume that strings such as $u_1, \ldots, u_n$ are ultimately encoded as binary strings, so that they fit our formal definition of representations.

Lemma 29 *With respect to the balanced p-ary representation of integers, for any $p \geq 3$:*
(i) Addition and negation of integers are in NC^0.
(ii) Iterated addition and multiplication of integers are in NC^1.
(iii) Iterated multiplication is in NC^2.

Proof. (i) Suppose we want to add $(u_1, \ldots, u_n)_p$ to $(v_1, \ldots, v_n)_p$. Note that for each $i = 1, \ldots, n$, we can express the sum $u_i + v_i$ as

$$u_i + v_i = px_i + y_i$$

with $|x_i| \leq 1$ and $|y_i| \leq p-2$. To see this, note that $|u_i + v_i| \leq 2p-2$ and if $|u_i + v_i| \leq p-2$ or $|u_i + v_i| \geq p$ then it is clear that the desired x_i, y_i can be found. The remaining possibility is $|u_i + v_i| = p-1$. In that case we could let $x_i = 1$ and $y_i = \pm 1$, but note that this is possible only because $p \geq 3$ (we would violate the constraint $|y_i| \leq p-2$ if $p = 2$). Now the sum of the two numbers is given by $(w_1, \ldots, w_n w_{n+1})_p$ where

$$w_i = x_{i-1} + y_i$$

for $i = 1, \ldots, n+1$ (taking $x_0 = y_{n+1} = 0$). Clearly this can be implemented by an NC^0 circuit.
(ii) To show iterated addition is in NC^1 we simply use part (i) and the technique for parallel prefix.

Now consider multiplication of integers. Suppose we want to form the product of the numbers $(u_1, \ldots, u_n)_p$ and $(v_1, \ldots, v_n)_p$. For each $i, j = 1, \ldots, n$, we form the product $u_i v_j$ and we can express this in the form

$$u_i v_j = p x_{i,j} + y_{i,j}$$

where $|x_{i,j}| \leq p-1$ and $|y_{i,j}| \leq p-1$ (since $|u_i v_j| \leq (p-1)^2 \leq p(p-1) + (p-1)$). For each $i = 1, \ldots, n$, form the number

$$X_i = (00 \cdots 00 x_{i,1} x_{i,2} \cdots x_{i,n-1} x_{i,n})_p$$

where X_i has a prefix of i zeroes. Similarly, form the number

$$Y_i = (00 \cdots 00 y_{i,1} y_{i,2} \cdots y_{i,n-1} y_{i,n})_p$$

where Y_i has a prefix of $i-1$ zeroes. It is then easy to see that the product is given by the sum

$$\sum_{i=1}^{n} (X_i + Y_i).$$

But each summand has at most $2n$ digits and there are $2n$ summands. We can form a balanced binary tree T on $2n$ leaves to organize this summation process: each leaf is labeled with one of these summands and each interior node is labeled with the sum of the labels at the leaves below. Clearly the root of T has the desired product. This tree converts into a Boolean circuit of depth $O(\log n)$. This shows multiplication is in NC^1.

(iii) We leave this as exercise. **Q.E.D.**

Next we extend the lemma to matrix rings. By the *balanced p-ary representation of matrices with integer entries* we mean that each entry is encoded by balanced p-ary notation, and matrices are stored (to be specific) in row-major order.

Lemma 30 *With respect to the balanced p-ary representation ($p \geq 3$) of integer matrices:*
(i) Addition and negation are in NC^0.
(ii) Multiplication is in NC^1.
(iii) Interated multiplication is in NC^2.

Proof. It is clear addition and negation of integer matrices can be implemented efficiently by the previous lemma. For multiplication, suppose we want to compute the product of two $n \times n$ matrices A and B, and each entry has at most n bits. We note that each entry of AB is the sum of at most n products of pairs of entries. These individual products can be viewed at the sum of at most $2n$ numbers of n bits, as revealed in the proof of the previous lemma. So for each entry, we need

to sum $O(n^2)$ numbers, each of n bits. Again, we can arrange these as a balanced binary tree of depth $O(\log n)$. This gives us the efficient NC^1 circuit we seek.

To get an NC^2 circuit family for iterated multiplication of integer matrices we simply apply parallel prefix to the previous part. **Q.E.D.**

Finally we consider matrices whose entries are rational numbers. A rational number is represented by a pair of balanced p-ary representation, extended to matrices as before. Unfortunately, we no longer know how to do addition of rational numbers in NC^0. Nevertheless, we have the following:

Lemma 31 *With respect to the balanced p-ary ($p \geq 3$) representation of matrices with rational number entries:*
(i) Iterated multiplication is in NC^2.
(ii) Characteristic polynomial computation is in NC^2.

Proof. (i) Suppose we want to compute the iterated product

$$A_1, A_1A_2, \ldots, A_1A_2 \cdots A_n$$

where each A_i is a $n \times n$ matrix with rational number entries, and each entry is represented by pairs of n-bit integers. We first convert each A_i to integer matrices B_i and compute an integer D_i such that $A_i = \frac{1}{D_i}B_i$. To do this, first form the product $D_{i,j}$ of the denominators in the jth row of A_i; then multiply each entry in the jth row of A_i by $D_{i,j}$. Doing this for all rows, we get B_i; of course D_i is just the product of all the $D_{i,j}$'s. It is clear that we can obtain each of the $D_{i,j}$ and D_i by iterated multiplication in NC^2. Notice that $D_{i,j}$ and D_i are $O(n^3)$-bit integers and so we can compute B_i from A_i and $D_{i,j}$'s in NC^1.

Next, we compute the iterated integer products $\{D_1, D_1D_2, \ldots, D_1D_2 \cdots D_n\}$ in NC^2. Similarly, we compute the iterated matrix product $\{B_1, B_1B_2, \ldots, B_1B_2 \cdots B_n\}$ in NC^2. It is clear that

$$A_1A_2 \cdots A_i = \frac{1}{D_1D_2 \cdots D_i}B_1B_2 \cdots B_i$$

for each $i = 1, \ldots, n$. This can be computed in NC^1 since each of the integer involved in polynomial in size.
(ii) We imitate the proof of lemma 27. Details are left as exercise. **Q.E.D.**

One more computational problem: we need to be able to check the sign of a balanced p-ary number. (In our application, we want to compare such a number with $\frac{1}{2}$, which is easily reduced to checking if a number is positive.) But after the preceding development, the reader should have no trouble devising an NC^1 solution (Exercises).

Putting it all together. We must tidy up the loose bits in the proof of the main theorem in the last section. In particular, we must address the issue of how to implicitly construct and represent the reduced transition matrices C described in the

last section. Let A be the transition matrix from which we derive C. Since we want to do all this using $O(s^2)$ space, we cannot afford to write A or C explicitly. Then we must see how the techniques given in this section can be adapted to some implicit representation. All this is tedious but it is crucial that the reader understands how this can be done. So let us assume a given probabilistic machine M accepting in space $s(n)$, and w is an input. Let us begin by constructing matrix C: the proof of lemma 24 shows a transitive closure circuit of depth $O(\log^2 n)$ applied to the underlying graph G_A of the transition matrix A to determine the inessential states. But note that the transitive closure circuit is relatively systematic that we can assume some numbering of its gates such that given any gate number g, we can determine the gates at the other end of incoming as well as outgoing edges at g, and given g, we also know the Boolean function labeling g. The gate numbers can be stored in $O(s)$ space and we can determine these information also in $O(s)$ space. It is now not hard to see that we can determine the output of any gate in $O(s)$ space, given that we know the input graph G_A. This is not hard to do (we basically store one Boolean value at each gate along the path from the output gate to the current position – a similar proof using this technique is shown in chapter 10.) In this way, in $O(s)$ space, we can determine if any given i is inessential.

The basis of the preceding argument is the observation that the transitive closure circuit is quite systematic and hence in space $O(s)$ we can answer basic questions such as connections between gates, etc. Similarly for all the other circuits in this section. The student should carefully work out some other cases. With this, we conclude.

Remark: In chapter 10, we will study the property stated above, that all the circuits in this section are 'systematically constructed' so that we can essentially determine gates and their interconnections in circuits efficiently. (These are called *uniformity* conditions.) For this reason we are contented with a somewhat sketchy outline here.

8.7 Complement of Probabilistic Space

This section proves that probabilistic space is closed under complementation. We begin with a useful result:

Lemma 32 *Let B be the fundamental part of the transition matrix of a Markov chain. If B is $m \times m$ and the entries of B are taken from $\{0, \frac{1}{2}, 1\}$, then the stochastic closure B^* has the property that dB^* is an integer matrix for some integer $d < m!2^m$.*

Proof. We know that $B^* = (I - B)^{-1}$. Thus $B^* = \frac{1}{\det(I-B)}\text{adj}(I - B)$. Clearly each entry of $2^{m-1}\text{adj}(I - B)$ is an integer. Also, $\frac{c}{\det(I-B)}$ is an integer for some $c \leq m!$. The result follows. **Q.E.D.**

Theorem 33 *Let* M *be any probabilistic machine that accepts in space* $s(n) \geq \log n$. *Then there is a* $c > 0$ *such that every accepted input of length* n *is accepted with probability more than*

$$\frac{1}{2} + 2^{-c^{s(n)}}.$$

Proof. At the end of section 3, we showed that the probability of accepting any input is given by a suitable entry of D^*, where D is the fundamental part of a reduced transition matrix C. D is $m \times m$ with $m = nO(1)^{s(n)} = O(1)^{s(n)}$. We then apply the previous lemma. **Q.E.D.**

Lemma 34 *Let* $s(n) \geq \log n$. *For any probabilistic machine that runs in space* $s(n)$, *there is a probabilistic machine* N *accepting* L(M) *and runs in space* s *with error gap* $(0, \frac{1}{2}]$. *Morever,* N *halts with probability* 1 *and has average time* $2^{2^{O(s)}}$.

Proof. Let $c = \max\{c_1, c_2\}$ where c_1 is chosen so that there are at most c_1^s configurations using space at most s, and c_2 is chosen so that (by the previous theorem) if M accepts an input w then M accepts with probability greater than $\frac{1}{2} + 2^{-c_2^{s(|w|)}}$. Now N is obtained by modifying the proof of lemma 9 in the last section:

repeat forever
1. Simulate M for another c^s steps. Nondeterministic choices of M become coin-tossing choices of N.
2. If M answers YES then we answer YES.
   If M answers NO, we rewind our tapes to prepare for a restarted simulation.
3. Toss $2c^s$ coins and answer NO if all tosses turn up *heads*.

end

(Notice that unlike in lemma 9, each iteration of the loop continues the simulation from where the last iteration left off, provided the last iteration did not halt.) Now we see that N halts with probability 1. The average time $\bar{t}$ is seen to satisfy the bound

$$\bar{t} \leq 3c^s + (1 - 2^{-2c^s})\bar{t}$$

which gives us $\bar{t} = 2^{2^{O(s)}}$ as desired.

If M rejects then clearly N rejects. So assume that M acccepts. Let us call a configuration C of M *live* if there is a computation path starting from C into a YES configuration. Similarly, a configuration C of N is *live* if there is a computation path from C into one in which the simulated M answers YES. If a configuration is not alive, we say it is *dead*.

For $k \geq 1$, define the following events for the probabilistic spaces Ω_w^{N} and Ω_w^{M}:

$$\begin{array}{lcl} R_k^{\mathrm{N}} & = & \{\text{N answers NO in the } k\text{th iteration}\} \\ D_k^{\mathrm{N}} & = & \{\text{N became dead during the } k\text{th iteration}\} \\ D_k^{\mathrm{M}} & = & \{\text{M became dead between the } (k-1)c^s\text{th and the } kc^s\text{th step}\} \\ A_k^{\mathrm{N}} & = & \{\text{N is alive at the end of the } k\text{th iteration}\} \\ A_k^{\mathrm{M}} & = & \{\text{M is alive at the end of the } kc^s \text{ step}\} \end{array}$$

Then the rejection event of N corresponds to

$$\begin{aligned} \bigcup_{k\geq 1}\{R_k^{\mathrm{N}}\} &= \bigcup_{k\geq 1}\left(\{R_k^{\mathrm{N}}, A_k^{\mathrm{N}}\} \cup \bigcup_{j=1}^{k}\{R_k^{\mathrm{N}}, D_j^{\mathrm{N}}\}\right) \\ &= (\bigcup_{k\geq 1}\{R_k^{\mathrm{N}}, A_k^{\mathrm{N}}\}) \cup (\bigcup_{j\geq 1}\bigcup_{k\geq j}\{R_k^{\mathrm{N}}, D_j^{\mathrm{N}}\}) \end{aligned}$$

Clearly the probability that M remains alive after c^s steps is at most $1-e$ where

$$e := 2^{-c^s}.$$

Since the probability of getting $2c^s$ heads in a row is e^2, the probability that N remains alive through one iteration is at most $(1-e)(1-e^2)$. We claim that

$$\Pr\{R_k^{\mathrm{N}}, A_k^{\mathrm{N}}\} = (1-e)^k(1-e^2)^{k-1}e^2.$$

(In this proof, it is instructive to set up the connection between the probabilistic spaces Ω_w^N and Ω_w^M for input w.) Hence

$$\begin{aligned} \Pr(\bigcup_{k\geq 1}\{R_k^{\mathrm{N}}, A_k^{\mathrm{N}}\}) &= \sum_{k\geq 1}\Pr\{R_k^{\mathrm{N}}, A_k^{\mathrm{N}}\} \\ &\leq e^2\sum_{k\geq 1}(1-e)^k(1-e^2)^{k-1} \\ &< e^2\sum_{k\geq 1}(1-e)^{k-1} \\ &= e. \end{aligned}$$

Next,

$$\begin{aligned} \Pr(\bigcup_{j\geq 1}\bigcup_{k\geq j}\{R_k^{\mathrm{N}}, D_j^{\mathrm{N}}\}) &\leq \Pr(\bigcup_{j\geq 1}\{D_j^{\mathrm{N}}\}) \\ &= \sum_{j\geq 1}\Pr\{D_j^{\mathrm{N}}\} \\ &\leq \sum_{j\geq 1}\Pr\{D_j^{\mathrm{M}}\} \\ &= \Pr\{\text{M rejects}\} \\ &\leq \frac{1}{2} - e \end{aligned}$$

by our choice of the constant $c \geq c_2$. Above we have used the inequality $\Pr\{D_j^{\mathrm{N}}\} \leq \Pr\{D_j^{\mathrm{M}}\}$ relating across two different probability spaces. Hence the probability that N rejects is less than the sum of $e + (\frac{1}{2} - e)$. We conclude that N accepts with probability greater than $\frac{1}{2}$. **Q.E.D.** reject??

The technique in this lemma can be viewed as using $m = c^s$ coin tosses to control a loop so that the expected number of iterations is 2^m. Since we need only $\log m$ space to control this loop, we are able to probabilistically achieve a number of iterations that is double exponential in the space used. This technique, due to Gill, demonstrates one of the fundamental capabilities of coin-tossing that distinguishes space-bounded probabilism from, say, space-bounded alternating computations. The expected number of iterations is achieved in the worst case: if M is a machine that does not halt, then N has expected time $2^{2^{\Omega(s(n))}}$.

We are ready to show that probabilistic space is closed under complementation.

Theorem 35 *If $s(n) \geq \log n$ is space-constructible then*

$$PrSPACE(s) = co\text{-}PrSPACE(s).$$

This result was shown by Simon [28]; the proof here is essentially from [25]. This result is almost an immediate consequence of lemma 34.

Proof. Given any probabilistic machine accepting in space $s(n)$, lemma 34 gives us another probabilistic machine accepting the same language in the same space bound with error gap $(0, \frac{1}{2}]$; moreover, M halts with probability 1. Then let N be the complement of M: i.e., N answers YES iff M answers NO. For any input w, the probability that N accepts w plus the probability that M accepts w is equal to the probability of halting, i.e., 1. Hence, N has the error gap $[\frac{1}{2}, 1)$. It follows that N accepts if and only if M rejects. **Q.E.D.**

8.8 Stochastic Time and Space

In this section, we give upper bounds on the complexity of time and space-bounded stochastic computations. Stochastic space is especially interesting in view of the tremendous computational power that seems inherent in it. Also, instead of Markov chains we now turn to the study of discrete time dynamical systems.

Theorem 36 *For all t, StA-TIME$(t) \subseteq$ ATIME(t^3).*

The basic idea for this result is the bit-counting technique that was used quite effectively for simulating probabilistic alternating machines (chapter 7, section 6). It turns out that several new technical problems arises.

Let M be a stochastic alternating machine that accepts in time $t(n)$. We construct an alternating machine N to simulate M. For the rest of this proof, we fix an input w that is accepted by M and let $t = t(|w|)$. We may assume that N has

guessed t correctly and let T_0 be the accepting computation tree of M on w obtained by truncating the complete computation tree $T_{\mathrm{M}}(w)$ at levels below t (as usual, root is level 0). To demonstrate the essential ideas, we assume that M has $\oplus$- and $\otimes$-states only. As in chapter 7 we 'normalize' the least fixed point values $Val_{T_0}(C)$ for each configuration in T_0:

$$VAL_0(C) := 2^{2^{t-\ell}}\, Val_{T_0}(C)$$

where $\ell = level(C)$. Thus T_0 is accepting if and only if $VAL_0(C_0) > 2^{2^t-1}$ where C_0 is the root of T_0. It is easy to see that $VAL_0(C)$ is an integer between 0 and 2^{2^t}. We shall think of $VAL_0(C)$ as a 2^t digit number in the balanced 4-ary notation. Although the balanced 4-ary notation is highly redundant, we will want to refer to the 'ith digit of $VAL_0(C)$' in an unambiguous manner. We will show how to uniquely choose a balanced 4-ary representation for each $VAL_0(C)$.

Let us note that in fact $VAL_0(C)$ can be explicitly defined as follows: if $\ell = level(C)$ and $C \vdash (C_L, C_R)$ (provided C is not a leaf) then

$$VAL_0(C) \quad = \quad \begin{cases} 0 & \text{if } C \text{ is a non-YES leaf} \\ 2^{2^{t-\ell}} & \text{if } C \text{ is a YES leaf} \\ 2^{2^{t-\ell-1}-1}(VAL_0(C_L) + VAL_0(C_R)) & \text{if } C \text{ is an } \oplus\text{-configuration} \\ VAL_0(C_L) \cdot VAL_0(C_R) & \text{if } C \text{ is an } \otimes\text{-configuration} \end{cases}$$

It follows that each $VAL_0(C)$ has at most $2^{t-\ell}$ digits of significance.

The alternating simulation of N effectively constructs a tree T_1 that is an 'expansion' of T_0. To describe T_1, we need to define a certain product of trees:

Definition 9 *Let T, T' be any two trees. For nodes $i, j \in T$, write $i \to j$ to mean that i is the parent of j. Their* product $T \times T'$ *consists of nodes (i, i') where $i \in T$ and $i' \in T'$ such that $(i, i') \to (j, j')$ if and only if either*
(a) $i = j$ and $i' \to j'$, or
(b) $i \to j$, i' is the root of T' and j' is a leaf of T'. ■

Clearly,

$$T \times T' = T' \times T \iff T = T',$$

$$SIZE(T \times T') = SIZE(T' \times T) = SIZE(T) \cdot SIZE(T'),$$

and

$$DEPTH(T \times T') = DEPTH(T' \times T) = DEPTH(T) + DEPTH(T').$$

Define tree T_1 to be $T_0 \times T^t$ where T^t is defined as the binary tree in which every internal node has 2 children and every path from the root to a leaf has length exactly $t + 1$. Hence T^t has exactly 2^{t+1} leaves. Let us assume the nodes in T^t are

strings $s \in \{L, R\}^*$ such that the root of T^t is the empty string ϵ, and each internal node $s \in T^t$ has two children, sL and sR.

We want an assignment function VAL_1 on T_1 in analogy to VAL_0. Write $VAL_1(C, s)$ (instead of $VAL_1((C, s))$, which is correct but pedantic) for the value assigned to the node $(C, s) \in T_1$, $C \in T_0, s \in T^t$. Instead of integers, $VAL_1(C, s)$ is a balanced 4-ary number.

First, we need one more definition: recall that in section 5, we compute the product of two balanced p-ary numbers $u = u_1 \cdots u_n$, $v = v_1 \cdots v_n$ as the sum of $2n$ balanced p-ary numbers X_i, Y_i $(i = 1, \ldots, n)$. Let us call X_i and Y_i the ith and $(n+i)$th *summand* of the product of u and v. Clearly the summands depend on the particular representation of the numbers $(u)_p$ and $(v)_p$. These $2n$ summands can be regarded as $2n$ digits numbers although they each have at most $2n-1$ digits of significance.

Suppose C is a leaf in T_0, $level(C) = \ell$. Then for any $s \in T^t$, $VAL_1(C, s)$ is defined to be the (ordinary) 4-ary representation of

$$2^{2^{t-\ell}} \text{ or } 0$$

depending on whether C is YES or not. Next assume $C \vdash (C_L, C_R)$. There are two cases:

(1) C is a $\otimes$-configuration.

(1.1) If s is a leaf of T^t. Then s is a string in $\{L, R\}^*$ of length $t+1$. Then $VAL_1(C, s)$ is the sth summand of the product of $VAL_1(C_L, \epsilon)$ and $VAL_1(C_R, \epsilon)$. Here s is interpreted as the 2-adic number (see chapter 1, section 4.2) where the symbols L, R in s are (respectively) interpreted as 1,2. Note that s ranges from 0 to 2^{t+1} and by definition the 0th summand is 0.

(1.2) If s is not a leaf of T^t then $VAL_1(C, s) = VAL_1(C, sL) + VAL_1(C, sR)$.

(2) C is a $\oplus$-configuration.

(2.1) If s is a leaf of T^t then let

$$VAL_1(C, s) = 2^{2^{t-\ell-1}-1}[VAL_1(C_L, \epsilon) + VAL_1(C_R, \epsilon)].$$

Note that we are multiplying by a power of 2 and this is relatively trivial in balanced 4-ary notation. Basically we must reexpress each balanced 4-ary digit as a pair of balanced 2-ary digit, shift these to the right by $2^{t-\ell-1} - 1$ positions. Then we recombine into balanced 4-ary digits.

(2.2) If s is not a leaf then $VAL_1(C, s) = VAL_1(C, sL) (= VAL_1(C, sR))$.

It is not hard to show that for all $C \in T_0$,

$$VAL_0(C) = VAL_1(C, \epsilon).$$

We note that VAL_1 is uniquely defined since the balanced p-ary representation at the leaves are uniquely specified, and this propagates to all other nodes using the above rules.

Now it should be easy for the reader to use the technique of chapter 7 to provide an alternating machine that guesses the tree T_1 in order to determine the predicate

$$DIGIT(C, s, i, b)$$

that is true if the ith digit of $VAL_1(C, s)$ is equal to b, for $|b| \leq 3$ and $i = 1, \ldots, 2^t$. To invoke this procedure, we may assume the work tapes of N contains the following information:

1. i in binary
2. C
3. s
4. $level(C)$ in unary
5. $t - level(C) + |s|$ in unary

Note that each of these uses $O(t)$ space. Moreover, from the above information, we can generate the arguments for the recursive calls in $O(t)$ steps. So the total work spend along any path in T_1 is $O(t^3)$ since T_1 has $O(t^2)$ levels. It is now clear that $DIGIT$ can be solved in $O(t^2)$ alternating time.

The final work to be done is to compare $VAL_1(C_0, \epsilon)$ to $\frac{1}{2}$ where C_0 is the root of T_0. Basically, our goal is to convert the balanced 4-ary number

$$VAL_1(C_0, \epsilon) = u_1 u_2 \cdots u_m \quad (m = 2^t)$$

to an ordinary 4-ary number

$$v_1 v_2 \cdots v_m.$$

We begin with a simple remark: since each of the digits v_i must be non-negative, if u_i is negative, we must borrow one unit from the next digit u_{i+1}. Let $b_i = 1$ or 0 depending on whether we need to borrow from u_{i+1} or not in order to make v_i non-negative. Of course, we must also take into account the borrow b_{i-1} from u_i. This gives us the equation

$$b_i = \begin{cases} 1 & \text{if } u_i - b_{i-1} < 0 \\ 0 & \text{if } u_i - b_{i-1} \geq 0 \end{cases}.$$

We set $b_0 = 0$. Note that this method is correct because we know that *apriori* that the number $(u_1 \cdots u_m)_4$ is non-negative: so the most significant non-zero digit is positive. It is not hard to reduce the above to: $b_i = 1$ *iff for some* j *(*$1 \leq j \leq i$*),* $u_j < 0$ *and for all* $k = j+1, \ldots, i$, $u_k = 0$. Hence we can in $O(t^2)$ alternating time check the value of any b_i for $i = 1, \ldots, 2^t$: we simply guess j and then universally check that $u_j < 0$ and $u_k = 0$ for $k = j+1, \ldots, i$. Of course, checking if $u_k = b$ is nothing but the subroutine $DIGIT(C_0, \epsilon, k, b)$ which can be determined in $O(t^2)$ time.

Since we can check for the value of borrow bits b_i, we can check the digits v_i in the 4-ary representation of $VAL_1(C_0, \epsilon)$ via $v_i = u_i - b_{i-1} + 4b_i$. Now it is easy to determine if $VAL_1(C_0, \epsilon)$ is greater than 2^{2^t-1}.

We must address one more detail. The above construction did not consider the other basis functions of a stochastic alternating machine: $\oplus, \wedge, \vee$. However, it should be evident that since we know how to add and to multiply, we can also compute the value

$$VAL_0(C) = 2^{2^{t-\ell-1}-1}(\,VAL_0(C_L) + \,VAL_0(C_R)) - \,VAL_0(C_L)\,VAL_0(C_R)$$

where $\ell = level(C)$, C is a $\oplus$-configuration and $C \vdash (C_L, C_R)$. The remaining MIN- and MAX-configurations are also easy. This completes our proof of theorem 36.

Space-bounded Stochastic computation. We consider an $s(n)$ space-bounded stochastic machine M. In analogy with Markov chains, we set up a finite number of *dynamical states*, each corresponding to a configuration of the machine M using space at most s. If the set of states is taken to be $\{1, \ldots, n\}$, a valuation V can be viewed as an n-vector

$$V = (v_1, \ldots, v_n) \in [0,1]^n.$$

Let V_0 denote the vector of zero elements. We have the usual operator $\tau = \tau_\Delta$ where $\Delta = \{1, \ldots, n\}$. Thus

$$\tau(V) = (\tau_1(V), \tau_2(V), \ldots, \tau_n(V))$$

where $\tau_i(V) = 0, 1$ or $v_j \circ v_k$ for some j, k depending on i, $\circ \in \{\oplus, \otimes, \oplus\}$. In chapter 7, we showed that the limit of the sequence

$$\tau(V_0), \tau^2(V_0), \tau^3(V_0), \ldots$$

is the least fixed point of our system. Let

$$V^*_\tau = (v^*_1, \ldots, v^*_n)$$

denote this limit. Clearly any fixed point $V = (v_1, \ldots, v_n)$ of τ satisfies the following set of equations: each $i = 1, \ldots, n$,

$$v_i = f_i(v_{j(i)}, v_{k(i)}) \tag{8.11}$$

where $f_i(x,y)$ is one of the stochastic functions $0, 1, x \fatsemi y, x \otimes y, x \oplus y$. Let $\Sigma(V)$ denote the set of equations (8.11). We can then characterize the least fixed point property V_τ^* as follows:

$$LFP(V_\tau^*) \equiv \Sigma(V_\tau^*) \wedge (\forall V)[\Sigma(V). \Rightarrow .V_\tau^* \leq V].$$

We are now ready to prove the following, by appeal to some recent results on the complexity of real closed fields (cf. Renegar [24]).

Theorem 37 *For all $s(n)$,*

$$StSPACE(s) \subseteq DTIME(2^{2^{O(s)}}).$$

Proof. Suppose M is a stochastic machine that accepts in space $s(n)$. We show how to decide if M accepts any input w in space $s(|w|)$. As usual, we can assume $s = s(|w|)$ is known, and let there be n configurations of M that uses space at most s. Without loss of generality, let these configurations be identified with the integers $1, 2, \ldots, n$ and 1 denotes the initial configuration on input w. Let τ be the operator corresponding of these configurations. Hence we want to accept iff the least fixed point V_τ^* of τ has the property that its first component $[V_\tau^*]_1$ greater than $\frac{1}{2}$. This amounts to checking the validity of the following sentence:

$$(\exists V_\tau^*)(\forall V)[\Sigma(V_\tau^*) \wedge [V_\tau^*]_1 > \frac{1}{2} \wedge (\Sigma(V). \Rightarrow .V_\tau^* \leq V)].$$

This sentence can be decided in time $2^{O(n^4)} = 2^{2^{O(s)}}$, using the results of Renegar [24]. **Q.E.D.**

EXERCISES

[8.1] Consider an alternative approach that distinguishes YO-answers from looping: assign the value $\frac{1}{2}$ to YO-configurations. (Thus, basis sets B for choice machines are required to the new constant function, $\frac{1}{2}$ in addition to the others. Also, the function $\frac{1}{2}$ adds nothing new if the toss-function $\textcircled{T}$ is already in B.) In the standard treatment of YO-configurations, it is not hard to see that a $\{\otimes, \oplus\}$-machine amounts to an alternating machine. In what sense does the alternative treatment of YO-configurations apparently increase the power of such machines? Is this apparent increase real?

[8.2] Suppose a stochastic machine does not have an error gap (*i.e.*, is not decisive). Prove that if it accepts in time bound $t(n)$ where t is time-constructible, then we can convert it into one accepting in time $O(t)$ with the minimal error gap $[\frac{1}{2}, \frac{1}{2}]$. Prove the same for space bounded acceptance.

[8.3] State all known inclusions among P, NP, RP, BPP, PP. Justify any inclusions claimed.

[8.4] Let M be a machine with error gap $G = (0, b]$. We construct the following machine N to boost the error gap G:

> 1. Simulate M on input from beginning until it halts.
> (If M loops then we loop)
> 2. If M answers YES then answer YES; else toss a coin.
> 3. If coin-toss is heads then answer NO; else go to 1.

(i) Show that N has error gap $G' = (0, b']$ where $b' = \frac{2b}{1+b}$.
(ii) For any $\epsilon > 0$, show how to modify step 3 so that $1 - b' \leq \epsilon$.

[8.5] Give another proof of Ko's theorem that $NP \subseteq BPP$ implies $NP = RP$ in §2. Recall the notation A_k in the first proof.
(i) Show that $\Pr(A_n) \geq (1 - 2^{-2n})^{2n}$.
(ii) Show that $(1 - 2^{-n})^n \geq \frac{1}{2}$ for large enough n. HINT: you may use the following facts:

- $e^t \geq 1 + t$ for all $t \in \mathbb{R}$ with equality iff $t = 0$.
- $\left(1 + \frac{t}{n}\right)^n \geq e^t \left(1 - \frac{t^2}{n}\right)$ for all $t, n \in \mathbb{R}$, $n \geq 1$ and $|t| \leq n$.

[8.6] Let $g = [1/3 - e, 1/3 + e]$ for some $0 < e < 1/3$. Give an analog of majority voting to amplify this gap.

[8.7] Let $t(n)$ be time-constructible. Determine the smallest function $t'(n)$ as a function of $t(n)$ such that

$$co\text{-}NTIME(t(n)) \subseteq PrTIME_b(t'(n)).$$

HINT: simulate each step of a universal machine, but at each step, ensure bounded-error by using majority votes.

[8.8] Let M be a probabilistic machine that runs in time $t(n)$ and which uses $\leq \log t(n)$ coin tosses along each computation path. Give an upper bound for the function $t'(n)$ such that $L(\mathrm{M}) \in PrTIME_b(t'(n))$.

[8.9] Let $P(x)$ and $Q(x)$ be as defined in section 2. For all $n = 0, 1, 2, \ldots$, let

$$P_n(x) := \begin{cases} x & \text{if } n = 0 \\ P(P_{n-1}(x)) & \text{if } n = odd \\ Q(P_{n-1}(x)) & \text{if } n = even \end{cases}$$

$$Q_n(x) := \begin{cases} x & \text{if } n = 0 \\ Q(Q_{n-1}(x)) & \text{if } n = odd \\ P(Q_{n-1}(x)) & \text{if } n = even \end{cases}$$

For example, $P_2(x) = Q(P(x)) = x^2(2-x^2)$, and $Q_2(x) = P(Q(x)) = x^2(2-x)^2$. The function $Q_2(x)$ was used by Valiant to give a non-constructive proof that the majority function for Boolean functions has a monotone formula of size $O(n \log n)$. The amplification function in the text is simply $A(x) = P_2(x) \oplus Q_2(x)$. Now write $p_n^+(e)$ for $P_n(\frac{1}{2} + e)$, $p_n^-(e)$ for $P_n(\frac{1}{2} - e)$, and similarly for $q_n^+(e), q_n^-(e)$ relative to Q_n. For example,

$$\begin{aligned}
p_1^+(e) &= \frac{1}{4} + e + e^2 \\
q_1^+(e) &= \frac{3}{4} + e - e^2 \\
p_1^-(e) &= \frac{1}{4} - e + e^2 \\
q_1^-(e) &= \frac{3}{4} - e - e^2 \\
p_2^+(e) &= \frac{7}{16} + \frac{3e}{2} + \frac{e^2}{2} - 2e^3 - e^4 \\
q_2^+(e) &= \frac{9}{16} + \frac{3e}{2} - \frac{e^2}{2} - 2e^3 + e^4 \\
p_2^-(e) &= \frac{7}{16} - \frac{3e}{2} + \frac{e^2}{2} + 2e^3 - e^4
\end{aligned}$$

$$\begin{aligned} q_2^-(e) &= \frac{9}{16} - \frac{3e}{2} - \frac{e^2}{2} + 2e^3 + e^4 \\ p_3^+(e) &= \frac{49}{256} + \frac{21e}{16} + \frac{43e^2}{16} - \frac{e^3}{4} - \frac{53e^4}{8} - 5e^5 + 3e^6 + 4e^7 + e^8 \\ q_3^+(e) &= \frac{207}{256} + \frac{21e}{16} - \frac{43e^2}{16} - \frac{e^3}{4} + \frac{53e^4}{8} - 5e^5 - 3e^6 + 4e^7 - e^8 \end{aligned}$$

Show
(i) $p_n^+(e) + q_n^-(e) = 1$
(ii) $p_n^-(e) + q_n^+(e) = 1$
(iii) $p_n^+(e) - p_n^-(e) = q_n^+(e) - q_n^-(e)$
(iv) $x = \frac{1}{2}$ is a fixed point of $A_n(x) = P_n(x) \oplus Q_n(x)$, *i.e.*, $A_n(\frac{1}{2}) = \frac{1}{2}$. Are there other fixed points?
(v) The exact relationships between the coefficients of $p_n^+(e), q_n^+(e), p_n^-(e)$ and $q_n^-(e)$.

[8.10] Are there reasons to believe that *co-NP* $\subseteq AM[2]$, based on the result about NONISO?

[8.11] Fill in some details in the proof that NONISO $\in AM[2]$.
(i) Prove Boppana's lemma on random linear functions h_B. HINT: By the inclusion-exclusion principle, infer

$$\Pr[z \in h_B(C)] \geq \sum_{x \in C} \Pr[z = h(x)] - \sum_{x,y \in C, x \neq y} \Pr[z = h(x) = h(y)].$$

(ii) Let aut(G) denote the automorphism group of $G \in \mathcal{G}_n$. Show that aut(G) is a subgroup of symmetric group S_n (comprising all n-permutations) (this is a basic fact about groups). Show that LIKE$(G_0, G_1) =$ aut$(G_0) \times$ LIKE$'(G_0, G_1)$ in case $G_0 \sim G_1$. What if $G_0 \not\sim G_1$?
(iii) Conclude that $|\,\text{LIKE}(G_0, G_1)| = n!$ or $2n!$ depending on whether $G_0 \sim G_1$ or not.
(iv) Carry out the details for converting $[c/2, 3c/4]$ into an error gap.

[8.12] If $L \in AM[k]$ then L is accepted by an Arthur-Merlin game in $k+1$ rounds with zero-error acceptance.

[8.13] Show how to amplify error gaps for languages in $AM[k]$.

[8.14] (1-dimensional random walk) Analyze the 1-dimensional random walk with parameter $0 < p < 1$.
(i) Show that the generating functions $G(s)$ for the probability $p_{0,0}^{(n)}$ ($n = 0, 1, \ldots$) of returning to the origin in n steps is given by $G(s) = (1 - 4pqs^2)^{1/2}$ where $q = 1 - p$.
(ii) Conclude that each Markov state is recurrent if and only if $p = \frac{1}{2}$.

(iii) In case $p = \frac{1}{2}$, show that the mean recurrence time is infinite. *Hint:* Use the relation that the generating function $F(s)$ for first re-entrance probability $f_{0,0}^{(n)}$ is related to $G(s)$ by $G(s)-1 = F(s)G(s)$ and the mean recurrence time is given by $\lim_{s\to 1^{-1}} \frac{dF(s)}{ds}$.

[8.15] (Erdös, Feller, Pollard) Let $(f_0, f_1, \ldots)$ be a stochastic vector with $f_0 = 0$ and period is equal to 1. (The period is the largest d such that $f_n > 0$ implies d divides n.) Now define $p_0 = 1$ and $p_n = \sum_{k=0}^{n} f_k p_{n-k}$. Prove that $\lim_{n\to\infty} p_n = \frac{1}{\mu}$ where $\mu = \sum_{n=0}^{\infty} n f_n$. *Hint:* Note that the relation between pair of sequences $\{f_n\}$ and $\{p_n\}$ is identical with the relation between the first entrance probabilities $f_i^{(n)}$ and n-step transition probabilities $p_{i,i}^{(n)}$ for a fixed state i in section 3.

[8.16] Define a transition matrix $A = (p_{i,j})_{i,j\geq 0}$ to be *doubly stochastic* if the row sums as well as column sums are equal to 1. Show that each space-bounded computation of a reversible probabilistic Turing machine gives rise to such a matrix. Study the properties of such machines.

[8.17] Show that $RP = NP$ iff some NP-complete language is in RP. Here RP is the class one-sided bounded-error probabilistic polynomial time languages.

[8.18] Let $t(n)$ be any complexity function and $K_1 = PrTIME_1(t(n))$. Also let K_2 be the class of languages accepted by probabilistic choice machines with bounded error and zero-error acceptance, running in time $t(n)$. Under what conditions on t would we obtain $K_1 = co\text{-}K_2$?

[8.19] Complete the proof in section 1 showing $ZPP = RP \cap co\text{-}RP$.

[8.20] (Gill)
(i) Show that PP and BPP are closed under complementation.
(ii) Show that BPP and RP are closed under union and intersection.

[8.21] (Gill) We consider probabilistic transducers. For any probabilistic transducer M and input w, we may talk of the probability that M on w produces x as output. Let this (least fixed point) probability be denoted $\Pr\{\mathrm{M}(w) = x\}$. Let t_{M} be the partial transformation such that for all w, $t_{\mathrm{M}}(w) = x$ if $\Pr\{\mathrm{M}(w) = x\} > 1/2$; and $t_{\mathrm{M}}(w) \uparrow$ if there is no such x. Clearly t_{M} is uniquely determined by M and is called the transformation computed by M. Show that if M is $s(n)$-space bounded then $x = t_{\mathrm{M}}(w)$ implies $|x| \leq f(s(|w|))$ where $f(n)$ is the number of configurations of M using space at most n.

[8.22]
(i) (Generalized Bezout's theorem) Let

$$F(x) = F_0 x^m + F_1 x^{m-1} + \cdots + F_m \quad (F_0 \neq 0)$$

be a matrix polynomial where each F_i is an $n \times n$ matrix. The *right value* of $F(x)$ at an $n \times n$ matrix A is given by

$$F(A) = F_0 A^m + F_1 A^{m-1} + \cdots + F_m.$$

(The *left value* $\hat{F}(A)$ is similarly obtained except that A is multiplied from the left.) Show that if $F(x)$ is divided by $xI - A$ from the left, the remainder is the right value $F(A)$. Hint: the proof is a direct long division of $F(x)$.
(ii) Let $B(x)$ be the adjoint of $xI - A$. Conclude that $(xI - A)B(x) = P_A(x)I$.
(iii) Infer the Cayley-Hamilton theorem from the Generalized Bezout's theorem.

[8.23] (Fisher-Ladner) Improve the $2 \log n$ depth for the parallel prefix circuit C_n in the text. *Hint:* consider a recursive construction of a circuit C'_n as illustrated in the following figure where, with the proper wires added, we have

$$DEPTH(C'_n) = 1 + \max\{DEPTH(C_{n/2}), DEPTH(C'_{n/2})\}.$$

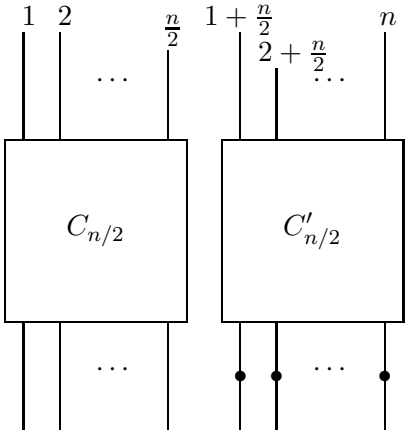


Figure 8.2 Construction of C'_n (the connections between $C'_{n/2}$ and $C_{n/2}$ not shown)

Note that the original C_n is used in this construction.
(a) Give exact expressions for the size and depth of C'_n. (Hint: the exact size of C'_n involves the Fibonacci numbers.)
(b) Compare the fan-out degree of C_n and C'_n.

[8.24] (Balanced p-ary representation)
(a) Suppose we have a fixed finite state automaton M that reads an input sequence of symbols $a_1 \cdots a_n$ in real time. Let q_i be the state of M after reading a_i, starting from the start state q_0. Show an NC^1 Boolean circuit

for the problem of computing $\{q_1, \ldots, q_n\}$ for any input sequence $a_1 \cdots a_n$. *Hint:* Apply parallel prefix.
(b) Apply the above to determine the sign of a balanced p-ary number in NC^1 ($p \geq 3$).
(c) Show how to convert a balanced p-ary number to a p-ary number in NC^1 (assume part (b)).

[8.25] (A general representation)
Let us fix $b \geq 1, p \geq 0, q \geq 1$. We say a word $u \in \{-p, -p+1, \ldots, q-1, q\}\star$ represents the integer

$$\sum_{i=0}^{n} u_i b^i \tag{8.12}$$

in the (b, p, q)-notation. Note that b-ary notations are $(b, 0, b-1)$-notations; b-adic notations are simply $(b, 1, b)$-notations; balanced p-ary numbers are $(p, -p, p)$-notations. Show that for $(3, 0, 3)$-notations, addition and multiplication of natural numbers (since we cannot represent negative numbers) is in NC^0 and NC^1, respectively. For what values of (b, p, q)-notations are these results true?

[8.26] (open) Does ZPP, RP or BPP have complete languages under reasonable reducibilities? (This seems unlikely)

[8.27] * Construct a complexity function $t(n)$ that the class $DTIME(t)$ does not have complete languages (under log-space reducibility).

[8.28] Instead of the usual acceptance rule for choice machines (*i.e.*, lower bound of $Val_{\mathrm{M}}(w)$ is greater than $\frac{1}{2}$), we could have generalized nondeterminism to the following computational mode:

> **Positive Acceptance Rule.** An input w is declared to be accepted by M if the lower bound of $Val_{\mathrm{M}}(w)$ is positive.

Show that if a language is accepted by the positive acceptance rule then it can be accepted by the usual acceptance rule with just one additional coin toss. Can a conversion in the other direction be effected?

[8.29] Recall the error gap $G_0(n) = [2^{-n}, 1 - 2^{-n}]$. Suppose we convert a machine with this error gap into one with bounded-error. What is the increase in time complexity?

[8.30] This simple exercise (from Emily Chapman) tests your basic understanding of probabilistic computation. To determine if a binary number n is odd, the obvious algorithm is, of course, to determine the predicate $LSB(n)$="the least significant bit of n is 1". Consider the following (unusual) algorithm:

LOOP:
Toss a coin. If head, then output $LSB(n)$. If tail, let $n \leftarrow n+2$ and go back to LOOP.

(i) Let T be the complete computation tree on input n. Determine the least fixed-point valuation Val_T.
(ii) Does this algorithm have bounded-error? zero-error?
(iii) Determine the running time $t(m)$ of this algorithm on inputs of length m. assuming $LSB(n)$ takes $\lg n$ steps to compute.
(iv) Determine the average running time $\overline{t}(m)$ of this algorithm.

[8.31]
(i) Under what conditions on $t(n)$ can we assert the following equality?

$$PrTIME_b(O(t)) = AvgTIME(O(t)).$$

NOTE: we cannot replace $O(t)$ by t here because our definition of probabilistic machines does not permit a linear speedup theorem even if we assume that $t(n)$ grows fast enough.
(ii) Show that

$$BPP = AvgTIME(n^{O(1)}).$$

[8.32] Assume the underlying ring R has nth roots of unity.
(i) Show that Toeplitz matrices can be multiplied with arithmetic circuits of size $O(n \log n)$ and depth $O(\log n)$. *Hint:* Use Fast Fourier Transform techniques.
(ii) (Open) Can one do better than $O(n^2)$ size circuits for general rings?

[8.33] (Berkowitz) Suppose the multiplication of two n by n matrices can be done in $O(n^\alpha)$ arithmetic operations, for some $\alpha \geq 2$. Currently, it is known that $\alpha < 2.496$ [8].
(i) Show that there is an arithmetic circuit of size $O(n^{1+\alpha})$ and depth $O(\log^2 n)$ to compute the characteristic polynomial over arbitrary rings.
(ii) (Conjecture) The size can be improved to $O(n^\alpha)$.

[8.34] Say a representation for a ring R is *efficient for isomorphism* if for all $n \geq 1$, there is an NC^1 circuit I_n to recognize isomorphic representations in $\{0,1\}^n$. Show that the balanced p-ary representation is efficient for isomorphism.

[8.35] Recall the construction of the probability space of a computation tree T. Carry out the proofs of the various properties asserted in the text. In particular, show that average time is well-defined and is equivalent to the alternative definition given.

[8.36] Is there a method to drop simultaneity requirement for probabilistic space with bounded error gaps?

[8.37] Imitate the proof that nondeterministic space is in probabilistic space with zero-error to show that we can ensure that the computation halts with probability 1 but with bounded error. *Hint:* Introduce a 'exit control' variable which is tossed many times and which allows the algorithm to halt (and answer NO immediately) only when all cointosses are heads. Thus there is a very small probability to exit.

[8.38] Carry out the proof of complement of probabilistic classes for bounded error and for 1-sided error.

Appendix A

Probabilistic Background

Basic Vocabulary. The original axiomatic treatment of probability theory due to Kolmogorov [17] is still an excellent rapid introduction to the subject. We refer to [4, 30] for additional reading of techniques useful for complexity applications. This appendix is a miscellany of quick reviews and useful facts.

A *Borel field* (or sigma-field) is a set system (Ω, Σ) where Ω is a set and Σ is a collection of subsets of Ω with three properties (i) $\Omega \in \Sigma$, (ii) $E \in \Sigma$ implies $\Omega - E \in \Sigma$ and (iii) $\{E_i\}_{i \geq 0}$ is a countable collection of sets in Σ then the countable union $\cup_{i \geq 0} E_i$ is in Σ. If (iii) is replaced by the weaker condition that $E_1, E_2 \in \Sigma$ imples $E_1 \cup E_2 \in \Sigma$ then we get a *field*. For any collection S of subsets of Ω, there is a unique smallest Borel field that contains S, called the Borel field *generated by* S. The most important example is the *Euclidean Borel field* (R^1, B^1) where R^1 is the real line and B^1 is the Borel field generated by the collection of intervals $(-\infty, c]$ for each real c, $-\infty < c < +\infty$. Members in B^1 are called *Euclidean Borel sets*.

A *probability measure on* (Ω, Σ) is a function $\Pr : \Sigma \to [0, 1]$ such that (a) $\Pr(\Omega) = 1$, (b) if $\{E_i\}$ is a countable collection of pairwise disjoint sets in Σ then $\Pr(\cup_{i \geq 0} E_i) = \sum_{i \geq 0} \Pr(E_i)$. A *probability space* is a triple $(\Omega, \Sigma, \Pr)$ where (Ω, Σ) is a Borel field and $\Pr$ is a probability measure on (Ω, Σ).

The elements in Ω are often called *elementary events* or *sample points*. Elements of Σ are called *events* or *measurable sets*. Thus Ω and Σ are called (respectively) the *event space* and *sample space*. $\Pr(E)$ is the *probability* or *measure* of the event E. A simple example of probabilistic space is the case $\Omega = \{H, T\}$ with two elements and Σ consists of all subsets of Ω (there are only 4 subsets), and $\Pr$ is defined by $\Pr(\{H\}) = p$ for some $0 \leq p \leq 1$.

A *random variable* (abbreviation: r.v.) X of a probability space $(\Omega, \Sigma, \Pr)$ is a real (possibly taking on the values $\pm\infty$) function with domain Ω such that for each real number c, the set

$$X^{-1}((-\infty, c]) = \{\omega \in \Omega : X(\omega) \leq c\}$$

belongs to Σ. We may simply write

$$\{X \leq c\}$$

for this event. In general, we write[1]

$$\{\ldots X \ldots\}$$

for the event $\{\omega : \ldots X(\omega) \ldots\}$. It is also convenient to write $\Pr\{X \in S\}$ instead of $\Pr(\{X \in S\})$. The intersection of several events is denoted by writing the defining conditions in any order, separated by commas: $\{X_1 \in S_1, X_2 \in S_2, \ldots\}$. If $f(x)$ is a "reasonably nice" real function, then $f(X)$ is a new random variable defined by $f(X)(\omega) := f(X(\omega))$. In particular, the following are random variables:

$$\max(X, Y), \quad \min(X, Y), \quad X + Y, \quad X - Y, \quad XY, \quad X^Y, \quad X/Y$$

where X, Y are random variables. (The last case assumes Y is non-vanishing.) Similarly, if X_i's are random variables, then so are

$$\inf_i X_i, \quad \sup_i X_i, \quad \liminf_i X_i, \quad \limsup_i X_i.$$

Each X induces a probability measure $\Pr_X$ on the Euclidean Borel field determined uniquely by the condition $\Pr_X((\infty, c]) = \Pr\{X \leq c\}$. We call $\Pr_X$ the *probability measure* of X. The *distribution function* of X is the real function given by $F_X(c) := \Pr_X((-\infty, c])$. Note that $F_X(-\infty) = 0, F_X(\infty) = 1$, F_X is non-decreasing and right continuous. Conversely given any such function F, we can define a random variable whose distribution function is F. A set of random variables is *identically distributed* if all members have a common distribution function. A finite set of random variables $\{X_i : i = 1, \ldots, n\}$ is *independent* if for any Euclidean Borel sets B_i $(i = 1, \ldots, n)$,

$$\Pr(\cap_{i=1}^n \{X_i \in B_i\}) = \prod_{i=1}^n \Pr\{X_i \in B_i\}.$$

An infinite set of random variables is independent if every finite subset is independent. A very important setting for probabilistic studies is a set of independent and identically distributed random variables, abbreviated as *i.i.d.*.

Let (Ω, Σ) be a field, and we are given $m : \Sigma \to [0, 1]$ such that for any countable collection of pairwise disjoint sets $\{E_i \in \Sigma : i \in I\}$,

$$E = \cup_{i \in I} E_i \in \Sigma \text{ implies } m(E) = \sum_{i \in I} m(E_i).$$

Then a standard theorem of Carathéodory says that m can be uniquely extended to a probability measure on (Ω, Σ^*), the Borel field generated Σ.

[1] This '$\{\cdots\}$' notation for events reflects the habit of probabilists to keep the event space implicit. Notice that while probability measures are defined on Σ, random variables are defined on Ω.

It is a basic construction to show that for any countable set of probability measures $\{m_i : i \geq 0\}$ on the Euclidean Borel field, we can construct a probability space $(\Omega, \Sigma, \Pr)$ and a collection of random variables $\{X_i : i \geq 0\}$ such that for each i, m_i is the probability measure of X_i. *Sketch*: We let Ω be the product of countably many copies of the real line R^1, so a sample point is $(w_0, w_1, \ldots)$ where $w_i \in R^1$. A *basic set* of Ω is the product of countably many Euclidean Borel sets $\prod_{i\geq 0} E_i$ where all but a finite number of these E_i are equal to R^1. Let Σ_0 consists of finite unions of basic sets and then our desired Borel field Σ is the smallest Borel field containing Σ_0. It remains to define Pr. For each basic set, define $\Pr(\prod_{i\geq 0} E_i) := \prod_{i\geq 0} \Pr(E_i)$ where only a finite number of the factors $\Pr(E_i)$ are not equal to 1. We then extend this measure to Σ_0 since each member of Σ_0 is a finite union of disjoint basic sets. This measure can be shown to be a probability measure on Σ_0. Then the said theorem of Carathéodory tells us that it can be uniquely extended to Σ. This concludes our sketch.

The random variable is *discrete* if it takes on a countable set of distinct values. In this case, we may define its *expectation* of X to be $E[X] := \sum_i a_i \Pr\{X = a_i\}$ where i range over all the distinct values a_i assumed by X. Note that $E[X]$ may not be finite. The *variance* of X is defined to be $Var[X] := E[(X - E[X])^2]$. This is seen to give $Var[X] = E[X^2] - (E[X])^2$.

A fundamental fact is that $E[X + Y] = E[X] + E[Y]$ where X, Y are *arbitrary* random variables. Using this fact, one often derive unexpected elementary consequences. In contrast, $Var[X + Y] = Var[X] + Var[Y]$ and $E[XY] = E[X]E[Y]$ are valid provided X, Y are independent random variables.

A random variable X that is 0/1-valued is called a *Bernoulli random variable*. The distribution function of such an X is denoted $B(1, p)$ if $\Pr\{X = 1\}$ is p. If $X_1, \ldots, X_n$ is a set of i.i.d. random variables with common distribution $B(1, p)$ then the random variable $X = X_1 + \cdots + X_n$ has the *binomial distribution* denoted by $B(n, p)$. It is straightforward to calculate that $E[X] = np$ and $Var[X] = np(1 - p)$ if X has distribution $B(n, p)$. Note that Bernoulli random variables is just another way of specifying events, and when used in this manner, we call the random variable the *indicator function* of the event in question. Furthermore, the probability of an event is just the expectation of its indicator function.

Estimations. Estimating probabilities is a fine art. There are some tools and inequalities that the student must become familiar with.
(a) One of these, Stirling's formula in the form due to Robbins (1955), should be committed to memory:

$$n! = \left(\frac{n}{e}\right)^n e^{\alpha_n} \sqrt{2\pi n}$$

where

$$\frac{1}{12n + 1} < \alpha_n < \frac{1}{12n}.$$

Sometimes, the alternative bound $\alpha_n > (12n)^{-1} - (360n^3)^{-1}$ is useful [10]. Using

these bounds, it is not hard to show [20] that for $0 < p < 1$ and $q = 1 - p$,

$$G(p,n)e^{-\frac{1}{12pn}-\frac{1}{12qn}} < \left(\begin{array}{c} n \\ pn \end{array} \right) < G(p,n) \tag{A.1}$$

where

$$G(p,n) = \frac{1}{\sqrt{2\pi pqn}} p^{-pn} q^{-qn}.$$

(b) The 'tail of the binomial distribution' is the following sum

$$\sum_{i=\lambda n}^{n} \left(\begin{array}{c} n \\ i \end{array} \right) P^i Q^{n-i}.$$

We have the following upper bound [10]:

$$\sum_{i=\lambda n}^{n} \left(\begin{array}{c} n \\ i \end{array} \right) P^i Q^{n-i} < \frac{\lambda Q}{\lambda - P} \left(\begin{array}{c} n \\ \lambda n \end{array} \right) P^{\lambda n} Q^{(1-\lambda)n}$$

where $\lambda > P$ and $Q = 1 - P$. This specializes to

$$\sum_{i=\lambda n}^{n} \left(\begin{array}{c} n \\ i \end{array} \right) < \frac{\lambda}{2\lambda - 1} \left(\begin{array}{c} n \\ \lambda n \end{array} \right) 2^{-n}$$

where $\lambda > P = Q = 1/2$.
(c) A useful fact is this: for all real x,

$$e^{-x} \geq 1 - x$$

with equality only if $x = 0$. If $x \geq 1$ then this is trivial. Otherwise, by the usual series for the exponential function, we have that for all real x

$$e^{-x} = \sum_{i=0}^{\infty} \frac{(-x)^i}{i!} = (1 - x) + \frac{x^2}{2!}(1 - \frac{x}{3}) + \frac{x^4}{4!}(1 - \frac{x}{5}) + \cdots.$$

The desired bound follows since $x < 1$.
(d) Jensen's inequality. Let $f(x)$ be a convex real function. Recall that 'convexity' means $f(\sum_i p_i x_i) \leq \sum_i p f(x_i)$ where $\sum_i p_i = 1, p_i \geq 0$ for all i where i range over a finite set. If X and $f(X)$ are random variables then $f(E(X) \leq E(f(X))$. We prove this for the case X has takes on finitely many values x_i with probability p_i. Then $E(X) = \sum_i p_i x_i$ and

$$f(E(X)) = f(\sum_i p_i x_i) \leq \sum_i p_i f(x_i) = E(f(X)).$$

For instance, if $r > 1$ then $E(|X|^r) \geq (E(|X|))^r$.

(e) Markov's inequality. Let X be a non-negative random variable, $e > 0$. Then we have the trivial inequality $H(X - e) \leq \frac{X}{e}$ where $H(x)$ (the Heaviside function) is the 0-1 function given by $H(x) = 1$ if and only if $x > 0$. Taking expections on both sides, we get

$$\Pr\{X > e\} \leq \frac{E(X)}{e}.$$

(f) Chebyshev's inequality. Let X be a discrete random variable, $\Pr\{X = a_i\} = p_i$ for all $i \geq 1$, with finite second moment and $e > 0$. Then

$$\Pr\{|X| \geq e\} \leq \frac{E(X^2)}{e^2}.$$

We say this gives an upper bound on tail probability of X. In proof,

$$\begin{aligned} E(X^2) &= \sum_{i \geq 1} p_i a_i^2 \\ &\geq e^2 \sum_{|a_i| \geq e} p_i \\ &= e^2 \Pr\{|X| \geq e\} \end{aligned}$$

Another form of this inequality is

$$\Pr\{|X - E(X)| > e\} \leq \frac{Var(X)}{e^2}$$

where $|X - E(X)|$ measures the deviation from the mean. We could prove this as for Markov's inequality, by taking expectations on both sides of the inequality

$$H(|X - E(X)| - e) \leq \left(\frac{X - E(X)}{e}\right)^2.$$

(g) Chernoff's bound [6] is concerned a set of i.i.d. random variables $X_1, \ldots, X_n$. Let $X = X_1 + \cdots + X_n$ and assume $E[X]$ is finite. Define

$$M(t) := E[e^{tX_1}]$$

and

$$m(a) = \inf_t E[e^{t(X_1 - a)}] = \inf_t e^{-at} M(t).$$

Then Chernoff showed that

$$E[X] \geq a \Rightarrow \Pr\{X \leq na\} \leq [m(a)]^n$$

and

$$E[X] \leq a \Rightarrow \Pr\{X \geq na\} \leq [m(a)]^n.$$

In particular, if X has distribution $B(n,p)$ then it is not hard to compute that

$$m(a) = \left(\frac{p}{a}\right)^a \left(\frac{1-p}{1-a}\right)^{1-a}.$$

Since it is well-known that $E[X] = np$, we obtain for $0 < e < 1$:

$$\Pr\{X \leq (1-e)np\} \leq \left(\frac{1}{1-e}\right)^{(1-e)np} \left(\frac{1-p}{1-(1-e)p}\right)^{n-(1+e)np}.$$

Markov Chains. We continue the discussion of Markov chains from section 3. The *period* of a state i in a chain A is defined in a combinatorial way: it is the largest positive integer d such that every cycle in the underlying graph G_A that contains i has length divisible by d. A state is *periodic* or *aperiodic* depending on whether its period is greater than 1 or not. It is left as an exercise to show that the period is a property of a component.

Recall that state i is recurrent if $f_i^* = 1$. In this case, there is certainty in returning to state i, and under this condition, we may speak of the *mean recurrence time for state i* μ_i, defined as follows:

$$\mu_i = \sum_{n=0}^{\infty} n f_i^{(n)}$$

Using the mean recurrence time, we may introduce new classification of states: state i is *null* if $\mu_i = \infty$, and *non-null* otherwise.

To illustrate the classification of states, we consider the (1-dimensional) random walk with parameter p_0 $(0 < p_0 < 1)$: this is the Markov chain whose states are the integers, and the transition probability is given by $p_{i,i+1} = p_0$ and $p_{i,i-1} = 1 - p_0$, for all i. It is clear that every state is essential. It can be shown that each Markov state if recurrent or transient depending on whether $p_0 = \frac{1}{2}$ or not (Exercises). So state 0 is recurrent iff $p = \frac{1}{2}$. Thus $p_0 \neq \frac{1}{2}$ provide examples of essential but transient states. In the recurrent situation, the mean recurrence time is infinite (Exercises). So this illustrates recurrent but null states.

Generating functions. A (real) formal power series is an infinite expression $G(s) = \sum_{n=0}^{\infty} a_n s^n$ in some indeterminate s, where $a_0, a_1, \ldots$ are given real numbers. We say that $G(s)$ is the (ordinary) generating function for the seqence $a_0, a_1, \ldots$. We can manipulate $G(s)$ algebraically: we may add, multiply or (formally) differentiate power series in the obvious way. One should think of $G(s)$ as a convenient way to simultanueously manipulate all the elements of a sequence; hence the terms s^n are just 'place-holders'. These operations reflect various combinatorial operations on the original series. Using well-known identities we can deduce many properties of such series rather transparently. Although we have emphasized that the manipulations are purely formal, we occasionally try to sum the series for actual values of s; then one must be more careful with the analytic properties of these series. Most

elementary identities involving infinite series reduces (via the above manipulations) to the following most fundamental identity $(1-x)^{-1} = \sum_{i \geq 1} x^i$. For example, the student observes that all the results from sections 3 and 4 involving limits is basically an exploitation of this identity.

Rate of Convergence of Substochastic matrices. In section 3, we showed that the entries of the nth power of the fundamental matrix of an absorbing chain approaches 0. We now give a more precise bound on the rate of convergence. For any matrix B, let $\delta_*(B)$ and $\delta^*(B)$ denote the smallest and largest entries in B. Let $\delta(B) = \delta^*(B) - \delta_*(B)$. We have the following simple lemma (cf. [16]):

Lemma 38 *Let $A = (a_{i,j})$ be a stochastic matrix each of whose entries are at least e for some $0 < e < 1$. For any $n \times m$ non-negative matrix $B = (b_{i,j})$, we have*

$$\delta(AB) \leq (1-2e)\delta(B).$$

Proof. Consider the (i,j)th entry $\sum_{k=1}^{n} a_{i,k}b_{k,j}$ of AB. Without loss of generality, assume that $a_{i,1} \leq a_{i,2}$.

To obtain a lower bound on the (i,j)th entry, assume wlog that $\delta^*(B) = \max\{b_{1,j}, b_{2,j}\}$. Then

$$\begin{aligned}
\sum_{k=1}^{n} a_{i,k}b_{k,j} &\geq a_{i,1}b_{1,j} + a_{i,2}b_{2,j} + (\sum_{k=3}^{n} a_{i,k})\delta_*(B) \\
&\geq a_{i,1}\delta^*(B) + a_{i,2}\delta_*(B) + (\sum_{k=3}^{n} a_{i,k})\delta_*(B)
\end{aligned}$$

where the last inequality must be justified in two separate cases (in one case, we use the simple fact that $a \geq b$ and $a' \geq b'$ implies $aa' + bb' \geq b' + a'b$). Thus

$$\begin{aligned}
\sum_{k=1}^{n} a_{i,k}b_{k,j} &\geq a_{i,1}\delta^*(B) + (\sum_{k=2}^{n} a_{i,k})\delta_*(B) \\
&= e\delta^*(B) + (\sum_{k=2}^{n} a_{i,k})\delta_*(B) + (a_{i,1} - e)\delta^*(B) \\
&\geq e\delta^*(B) + (1-e)\delta_*(B)
\end{aligned}$$

To obtain an upper bound on the (i,j)th entry, Assuming wlog that $\delta_*(B) = \min\{b_{1,j}, b_{2,j}\}$, we have

$$\begin{aligned}
\sum_{k=1}^{n} a_{i,k}b_{k,j} &\leq a_{i,1}b_{1,j} + a_{i,2}b_{2,j} + (\sum_{k=3}^{n} a_{i,k})\delta^*(B) \\
&\leq a_{i,1}\delta_*(B) + a_{i,2}\delta^*(B) + (\sum_{k=3}^{n} a_{i,k})\delta^*(B)
\end{aligned}$$

$$\begin{aligned} &\leq a_{i,1}\delta_*(B) + (\sum_{k=2}^{n} a_{i,k})\delta^*(B) \\ &= e\delta_*(B) + (\sum_{k=2}^{n} a_{i,k})\delta^*(B) + (a_{i,1} - e)\delta_*(B) \\ &\leq e\delta_*(B) + (1-e)\delta^*(B). \end{aligned}$$

The lemma follows since the difference between the largest and smallest entry of AB is at most

$$(e\delta_*(B) + (1-e)\delta^*(B)) - (e\delta^*(B) + (1-e)\delta_*(B)) \leq (1-2e)\delta(B).$$

Q.E.D.

In the exercises, we show how to extend this to substochastic matrix B, *i.e.*, each row sum in B is at most 1.

Exercises

[1.1] Show the following inequalities ((i)-(iv) from [Kazarinoff]):
(i) $(1+\frac{1}{n})^n < (1+\frac{1}{n+1})^{n+1}$.
(ii) $(1+\frac{1}{n})^n < \sum_{k=0}^{n} \frac{1}{k!} < (1+\frac{1}{n})^{n+1}$.
(iii) $n! < (\frac{n+1}{2})^n$ for $n = 2, 3, \ldots$.
(iv) $(\sum_{i=1}^{n} x_i)(\sum_{i=1}^{n} \frac{1}{x_i}) \geq n^2$ where x_i's are positive. Moreover equality holds only if the x_i's are all equal.
(v) $n! < \left(\frac{12n}{12n-1}\right)(2\pi n)^{1/2} e^{-n} n^n$. (Use Robbin's form of Stirling's formula.)

[1.2]
(i) (Hölder's Inequality) If X and Y are random variables, and $1 < p < \infty$, $\frac{1}{p} + \frac{1}{q} = 1$ then

$$E[XY] \leq E[|XY|] \leq E[|X|^p]^{1/p} E[|Y|^q]^{1/q}.$$

When $p = 2$, this is the Cauchy-Schwartz inequality. (In case $Y \equiv 1$ we have $E[|X|] \leq E[|X|^p]^{1/p}$, which implies the Liapounov inequality: $E[|X|^r]^{1/r} \leq E[|X|^s]^{1/s}$ for $1 < r < s < \infty$.)
(ii) (Minkowski's Inequality)

$$E[|X+Y|^p]^{1/p} \leq E[|X|^p]^{1/p} + E[|Y|^p]^{1/p}.$$

(iii) (Jensen's inequality) If f is a convex real function, and suppose X and $f(X)$ are integrable random variables. Then $f(E(X)) \leq E(f(X))$. (Note that convexity means that if $\sum_{i=1}^{n} c_i = 1$, $c_i > 0$, then $f(\sum i = 1^n c_i x_i) \leq \sum_{i=1}^{n} c_i f(x_i)$.)

[1.3] Construct the probability space implicitly associated with a Markov chain.

[1.4] For any positive integer k, construct a finite Markov chain with states $0, 1, \ldots, n$ such that states 0 has the value $k \leq p^*_{0,0} < k+1$. Try to minimize $n = n(k)$.

[1.5] In this exercise, we do not assume the Markov chain is finite. Show that the following are properties, though defined for individual states, are characteristics of components:
(i) Period of a Markov state.
(ii) Nullity of a Markov state.

[1.6] Show that $g_{i,j} = f^*_{i,j} g_{j,j}$. (From this we conclude that $g_{i,j} > 0$ if and only if $g_{i,j} = f^*_{i,j}$.) *Hint:* Write $g_{i,j} = \sum_{n=0}^{\infty} \Pr(A_n B_n C_n | D)$ where D is the event that the state at time 0 is i, A_n is the event that the states at times $1, \ldots, n-1$ are not equal to j, B_n is the event that the state at time n is equal to j, C_n is the event that the state at time s is equal to j for infinitely many $s > n$. Then $\Pr(A_n B_n C_n | D) = \Pr(C_n | A_n B_n D) \Pr(A_n B_n | D)$ But the Markov property implies $\Pr(C_n | A_n B_n D) = \Pr(C_n | D)$.

[1.7] In the appendix, we proved a bound on the rate of convergence of stochastic matrices. Extend it to substochastic matrices.

[1.8] (a) Prove equation (A.1) in the appendix. (b) Say $f(n) \sim g(n)$ if $f(n)/g(n)$ approaches 1 as $n \to \infty$. Conclude that for $k = 1, \ldots, n/2$, $p = k/n$ and $q = 1 - p$, then as $k \to \infty$ and $n - k \to \infty$:

$$\left(\begin{array}{c} n \\ k \end{array} \right) \sim \frac{1}{\sqrt{2\pi pqn}(p^p q^q)^n}$$

(c) Let $0 < p < 1$ and $q = 1 - p$. Show that the probability that a Bernoulli random variable with mean p attains k successes in n trials is

$$\left(\begin{array}{c} n \\ k \end{array} \right) p^k q^{n-k} \sim \frac{1}{\sqrt{2\pi pqn}}$$

[1.9] (J. Schmidt) Show
(a)

$$\left(\frac{1-p}{1-\delta p} \right)^{1-\delta p} \leq e^{\delta - 1}$$

for $0 < \delta < 2$.
(b)

$$\left(\frac{1}{1+e} \right)^{1+e} \leq e^{-e-(e^2/3)}$$

for $0 < e \leq 1$.
(c)

$$\left(\frac{1}{1-e}\right)^{1-e} \leq e^{e-(e^2/2)}$$

for $0 < e \leq 1$.
(d) Conclude that in the binomial case of Chernoff's inequality (see appendix),

$$\Pr\{X \geq (1+e)np\} \leq e^{-(e^2/3)np}$$

and

$$\Pr\{X \leq (1-e)np\} \leq e^{-(e^2/2)np}.$$

[1.10] Deduce from Chernoff's bound the following estimate on the tail of binomial distribution:

$$\sum_{i=\lfloor t/2 \rfloor}^{t} \left(\begin{array}{c} t \\ i \end{array} \right) p^i q^{t-i} \leq (4pq)^{t/2}.$$

Bibliography

[1] A. Avizienis. Signed-digit number representation for fast parallel arithmetic. *Inst. Radio Engr. Trans. Electron. Comput.*, 10:389–400, 1961.

[2] László Babai and Shlomo Moran. Arthur-Merlin games: a randomized proof system, and a hierarchy of complexity classes. *Journal of Computers and Systems Science*, 36:254–276, 1988.

[3] S. Berkowitz. On computing the determinant in small parallel time using a small number of processors. *Information Processing Letters*, 18:147–150, 1984.

[4] Béla Bollobás. *Random Graphs*. Academic Press, 1985.

[5] A. Borodin, S. Cook, and N. Pippenger. Parallel computation for well-endowed rings and space-bounded probabilistic machines. *Information and Computation*, 58:113–136, 1983.

[6] Herman Chernoff. A measure of asymptotic efficiency for tests of hypothesis based on sum of observations. *Ann. of Math. Stat.*, 23:493–507, 1952.

[7] Kai Lai Chung. *Markov Chains with stationary transition probabilities*. Springer-Verlag, Berlin, 1960.

[8] Don Coppersmith and Samuel Winograd. On the asymptotic complexity of matrix multiplication. *Proc. 22nd FOCS*, pages 82–90, 1981.

[9] K. de Leeuw, E. F. Moore, C. E. Shannon, and N. Shapiro. *Computability by probabilistic machines*. Automata Studies. Princeton, New Jersey, 1956.

[10] William Feller. *An introduction to Probability Theory and its Applications*. Wiley, New York, 2nd edition edition, 1957. (Volumes 1 and 2).

[11] F. R. Gantmacher. *The Theory of Matrices*. Chelsea Pub. Co., New York, 1959. Volumes 1 and 2.

[12] J. T. Gill. Computational complexity of probabilistic Turing machines. *SIAM J. Comp.*, 6(4):675–695, 1977.

[13] S. Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proofs. *17th ACM Symposium STOC*, pages 291–304, 1985.

[14] Shafi Goldwasser and Michael Sipser. Private coins versus public coins in interactive proof systems. *18th ACM Symposium STOC*, pages 59–68, 1986.

[15] H. Jung. Relationships between probabilistic and deterministic tape complexity. *10th Sympos. om Mathematical Foundations of Comp. Sci.*, pages 339–346, 1981.

[16] J. G. Kemeny and J. L. Snell. *Finite Markov chains*. D. Van Nostrand, Princeton, N.J., 1960.

[17] A. N. Kolmogorov. *Foundations of the theory of probability*. Chelsea Publishing Co., New York, 1956. Second English Edition.

[18] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, 1980.

[19] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[20] W. Wesley Peterson and Jr. E. J. Weldon. *Error-Correcting Codes*. MIT Press, 1975. 2nd Edition.

[21] Nicholas Pippenger. Developments in "The synthesis of reliable organisms from unreliable components". manuscript, University of British Columbia, 1988.

[22] Michael O. Rabin. Probabilistic algorithms. In J. F. Traub, editor, *Algorithms and Complexity: New Directions and Recent Results*, pages 21–39. Academic Press, New York, 1976.

[23] Jaikumar Radhakrishnan and Sanjeev Saluja. Lecture notes: Interactive proof systems. Research Report MPI-I-95-1-007, Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany, March 1995. This is a full TECHREPORT entry.

[24] James Renegar. On the computational complexity and geometry of the first-order theory of the reals: Parts I, II, III. Technical Report 853, 854, 856, School of Operations Research and Industrial Engineering, Cornell University, 1989.

[25] W. L. Ruzzo, J. Simon, and M. Tompa. Space-bounded hierarchies and probabilistic computations. *Journal of Computers and Systems Science*, 28:216–230, 1984.

[26] Uwe Schöning. *Complexity and Structure*, volume 211 of *Lecture Notes in Computer Science*. Springer-Verlag, 1986.

[27] J. Simon. On tape bounded probabilistic Turing machine acceptors. *Theoretical Computer Science*, 16:75–91, 1981.

[28] Janos Simon. Space-bounded probabilistic turing machine complexity are closed under complement. *13th Proc. ACM Symp. Theory of Comp. Sci.*, pages 158–167, 1981.

[29] R. Solovay and V. Strassen. A fast monte-carlo test for primality. *SIAM J. Computing*, 6:84–85, 1977.

[30] Joel Spencer. *Ten Lectures on the Probalistic Method*. CBMS-NSF Regional Conference Series in Applied Mathematics. Society for Industrial and Applied Mathematics, Philadelphia, 1987.

Contents