Birthday party

Author(s) of the problem: **Michal Forišek**
Contest-related materials by: **Michal Forišek, Jana Gajdošíková**

Introduction

From our point of view, the names of John's friends will be boolean variables. If a variable is true, it means that John should invite the corresponding person and vice versa. But then the requests John got are nothing else than logical formulas! Our task is to assign logical values to the variables so that each of the formulas will be true.

This is an important problem in theoretical computer science. It is so important, that in has even got a name – SAT. (This is just an abbreviation of "satisfiability".) In general, this problem is known to be NP-complete. Between other things this means, that there is no known algorithm, that solves SAT in polynomial time.

On the other hand, some of the input files were pretty large, and obviously no exponential-time algorithm had a chance to solve them in mere 5 hours. But then, this was an open data problem. If the backtracking algorithm doesn't do the trick, we will have to find something that may help us. Keep in mind that you may use any means necessary to produce the correct output. This especially means that sometimes it is much easier to edit something by hand than to code another 100 lines into your program.

Some words about logic

In the following paragraphs, letters `A`, `B`, ... will denote arbitrary logical formulas, not only variables. We will use `-` to denote the negation of any formula. This means, that we will be able to denote also some logical formulas that weren't allowed in the problem statement (for example `-(A | B)`). You have probably realized that the operator `&` was logical *and* (we will call it a *conjunction* of the variables), `|` was logical *or* (a *disjunction*) and `=>` was an *implication*.

As a most basic fact note that the formulas `(A => B)`, `(-A | B)` and `(-B => -A)` are equivalent. As a consequence, the formula `(A => -A)` is true iff `A` is false. The formula `(A => (B => C))` is equivalent to `((A & B) => C)`. Therefore `(A => (-A => B))` is always true. We will also need the *de Morgan's rules*:

- `-(A1 & ... & Am)` is equivalent to `(-A1 | ... | -Am)`
- `-(A1 | ... | Am)` is equivalent to `(-A1 & ... & -Am)`.

From the facts mentioned above follows that the following formulas are equivalent:

- `(A1 => (A2 => (... (Am => (B1 | ... | Bn))...)))`
- `((A1 & ... & Am) => (B1 | ... | Bn))`
- `((-(A1 & ... & Am)) | (B1 | ... | Bn))`
- `(-A1 | ... | -Am | B1 | ... | Bn)`

We will call all variables and their negations by the common name *literal*. We say, that a formula is in the *conjunctive normal form* (CNF), if it is a conjunction of some logical formulas and each of these formulas is a disjunction of some literals. For example, the formula `((A | B) & (B | -C | C | C) & -D & (A | C))` is in CNF. It is not hard to prove, that each formula has an equivalent one, that is in CNF. The observations we made will help us later to rewrite some input files into equivalent ones, that are in CNF.

Inputs 1-4

Just parsing the input file and reading it correctly is quite a lot of work. But when we take a look at the input files, we may see that with almost no work we can make reading the input a lot easier.

First of all, note the names of the variables in inputs 2..10. They are: `b, c, d, ..., i, j, ba, bb, ...` Does it remind you of something? And when you see the sequence: `1, 2, 3, ..., 8, 9, 10, 11, ...`? After we replace the letters `a-j` by the numbers `0-9`, the names are just numbers from 1 to N. (Under Linux, this can be done by one command: "`tr a-j 0-9`".) And how convenient, negation is denoted by the minus sign, so negations of the variables will be the numbers from -1 to $-N$. Input 1 differs, and the most efficient way to get rid of this difference is to solve it completely by hand.

Almost all formulas in the first four inputs are of the form `(lit1 | lit2 | ... | litK)`, where each `litX` is a literal. This type of input is quite convenient, because it simply means that at least one of the literals in the formula has to be true. We simply rewrite the remaining few formulas into equivalent ones, having this form. As we don't need the characters `(,),|` anymore, we may delete them. If we regard the whole input file as one big conjunction of its lines, we see, that after rewriting the bad lines the input is in CNF.

Input files 1 to 4 were quite small, any (for input 4: any not completely brute-force) backtracking algorithm could find a solution in a reasonable amount of time. Loading the input and checking whether all the formulas are satisfied for some particular values of the variables becomes easy when the input file is in CNF.

Input 10

This was the biggest and ugliest of the input files, but definitely not the hardest one. When we take a closer look at the input file, we discover that its last lines are of the form `(A => -A)` and `(-B => B)`. From the first one we know that `A` has to be false, from the second one `B` is true. In this way we know the values of all but the first three variables. The remaining three variables can be determined by looking at the first three lines of the input.

From the problem statement we know that a solution exists. What we found is the only possible solution, therefore it is the solution we seek. We **don't** have to verify, whether also the other formulas are true. (In fact they are, the input file was correct. How would you generate such an input file?)

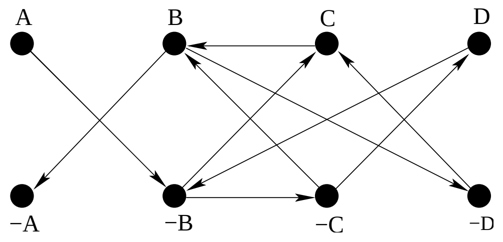
Inputs 5-9

These inputs are way too big for an exponential-time algorithm to work in reasonable time. So let's take a closer look at the input files. We will find out that each (input

9: almost each) of the formulas contains only two literals forming an implication or a disjunction. How may this help us?

We may rewrite each formula into the form of an implication. For example `(A | B)` becomes `(-A => B)`. Now we will build a directed graph. The vertices of our graph will be the literals, also the variables and their negations. The implications will form directed edges in our graph. The meaning of an edge is following: if its source vertex is true, then also its destination vertex has to be true.

From the formula above we would get the edge from `-A` to `B`. Note that the formula is also equivalent to `(-B => A)`, and so we get also the edge from `-B` to `A`. In a similar way each formula in the input file creates two edges in our graph. Note that the graph is symmetric in the following way: if we swap variables and their negations and rotate the direction of the edges, we get the same graph.



Our graph for the following formulas:
`(A => -B)`, `(B | -C)`, `(B | C)`, `(D => -B)` and `(C | D)`.

Now we want to label each of the vertices *true* or *false*, so that for each variable `A` exactly one of the vertices corresponding to `A`, `-A` is *true*. Also if some vertex v is *true*, then all vertices u such that there is an oriented $v - u$ path have to be *true*.

Clearly if for some variable `A` the vertices corresponding to `A` and `-A` lie in the same strongly connected component, such labeling does not exist. (One of them has to be true, and if they are in the same component, this means that the other one has to be true too – a contradiction.) We will show that in all other cases a solution does exist.

Take some topologically maximal strongly connected component C. In other words, divide the graph into strongly connected components and take any component C such that no edge enters C. (Is it possible that there would be no such component? Why not?) We will label the vertices in C *false*. By the symmetry of the graph, the vertices corresponding to the negations of literals in C form a topologically minimal (e.g. such that no edge leaves it) strongly connected component C' in our graph. We label all the vertices in C' *true*. Clearly the labeling of vertices in C and C' is correct and it does not restrict the labeling of the rest of the graph in any way. Thus we may remove the components C, C' from the graph and label the rest of it recursively.

The program is a straigthforward implementation of the idea above. The size of the graph is linear in the size of the input. There is a well-known algorithm (based on depth-first search) to find the strongly connected components of a graph in time linear in its size. Then we apply topological sort to the resulting component graph and label its vertices in the way described above. Thus the solution is linear in the size of the input.