CEOI 2002 tasks & correct solutions
(c) Michal Forišek et al.
Preface

Dear reader,

As you maybe know, in the year 2002 the Central European Olympiad in Informatics took place in Košice, Slovakia on 30 June – 6 July. We did our best to prepare the best CEOI ever :-)) and as a beginning of a new tradition, we compiled all the prepared tasks and also their solutions into this booklet. You may find here also the backup tasks that weren’t used during the real contest. We hope that this booklet will help many contestants to prepare for their future programming contests.

Let me tell you a few words about the people who prepared this contest. In Slovakia almost all local programming contests are organized by a group of young enthusiastic students of Comenius University, Bratislava. You may already have heard about our Correspondence Seminar in Programming or the annual Internet Problem Solving Contest (which was in our opinion the first open-data contest). We are also responsible for our national Informatics Olympiad.

However, this CEOI was by far the most important programming contest we organized so far and all of us worked very hard to prepare everything for it. I have to mention that none of us helps to organize these contests for profit, as we hardly ever get any. I think that the least I can do now is to say a big ‘Thank you!’ to everyone that helped to make this CEOI a little bit better.

Last but not least, none of us is a native English speaker, so please be tolerant in case you should encounter any spelling mistakes.

Michal Forišek
CENTRAL EUROPEAN OLYMPIAD IN INFORMATICS
Košice, Slovak Republic
30 June – 6 July 2002

Contents

Preface ..................................................... iii

Practice task ............................................. 1
   Give it a try! ......................................... 2
   Solution ............................................... 3

Tasks ....................................................... 4
   Bugs Integrated, Inc. ............................... 5
   Conqueror’s battalion .............................. 7
   A decorative fence .................................. 10
   A highway and the seven dwarfs .................. 12
   Royal guards ........................................ 14
   Birthday party ...................................... 16
   Building roads ..................................... 18
   Travel cheap! ....................................... 20
   Turtle breeding ..................................... 22

Solutions .................................................. 24
   Bugs Integrated, Inc. ............................... 25
   Conqueror’s battalion .............................. 28
   A decorative fence .................................. 30
   A highway and the seven dwarfs .................. 32
   Royal guards ........................................ 33
   Birthday party ...................................... 34
   Building roads ..................................... 37
   Travel cheap! ....................................... 38
   Turtle breeding ..................................... 40
Practice task
Give it a try

Input file: try.in                        100 points
Output file: try.out                    Time limit: 0.1 s
Source Code: try.pas/.c/.cpp           Memory limit: 1 MB

This problem is easy. Its main purpose is to let you test the environment, web interface and all the other things you will encounter tomorrow during the real contest session. Good luck!

Task
You will be given several integers \( N_i \). For each of them you have to compute the sum of all integers between 1 and \( N_i \) (inclusive).

Input
The first line of the input file consists of a single integer \( D \) (1 \( \leq \) \( D \) \( \leq \) 5 000), denoting the number of input cases. \( D \) lines follow, the \( i \)-th of them will contain a single integer \( N_i \) (|\( N_i \)| \( \leq \) 30 000).

Output
For each number \( N_i \) output one line containing the corresponding sum.

Example

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>55</td>
</tr>
</tbody>
</table>

Note
The example here is different from the one used to test your solutions when you submit them. During the real contest session your solution will be tested on the example input when you submit it.
Give it a try

It is a well-known fact that $\sum_{i=1}^{N} i = \frac{N(N+1)}{2}$. The only catch with this problem was that $N$ could be zero and even negative. For example for $N = -5$ the set of all integers between 1 and $N$ (inclusive) is $\{-5, -4, -3, -2, -1, 0, 1\}$. Also for $N \leq 0$ the corresponding sum is $-\frac{N(N+1)}{2} + 1$.

If your program didn’t work correctly for negative $N$, you should’ve get the Wrong Answer message when you submitted it. The trivial solution (using a cycle to compute the sum) caused a Time Limit Exceeded when you submitted it. However, it was possible to pre-compute all the sums using the trivial solution and hardwire them into the source code you submitted. The resulting source code would be smaller than 1 MB, also this was another way to solve this problem in the given time limit.
Tasks
Bugs Integrated, Inc. is a major manufacturer of advanced memory chips. They are launching production of a new six terabyte Q-RAM chip. Each chip consists of six unit squares arranged in a form of a $2 \times 3$ rectangle. The way Q-RAM chips are made is such that one takes a rectangular plate of silicon divided into $N \times M$ unit squares. Then all squares are tested carefully and the bad ones are marked with a black marker.

Finally, the plate of silicon is cut into memory chips. Each chip consists of $2 \times 3$ (or $3 \times 2$) unit squares. Of course, no chip can contain any bad (marked) squares. It might not be possible to cut the plate so that every good unit square is a part of some memory chip. The corporation wants to waste as little good squares as possible. Therefore they would like to know how to cut the plate to make the maximum number of chips possible.

**Task**

You are given the dimensions of several silicon plates and a list of all bad unit squares for each plate. Your task is to write a program that computes for each plate the maximum number of chips that can be cut out of the plate.

**Input**

The first line of the input file consists of a single integer $D$ ($1 \leq D \leq 5$), denoting the number of silicon plates. $D$ blocks follow, each describing one silicon plate. The first line of each block contains three integers $N$ ($1 \leq N \leq 150$), $M$ ($1 \leq M \leq 10$), $K$ ($0 \leq K \leq MN$) separated by single spaces. $N$ is the length of the plate, $M$ is its height and $K$ is the number of bad squares in the plate. The following $K$ lines contain a list
of bad squares. Each line consists of two integers $x$ and $y$ ($1 \leq x \leq N$, $1 \leq y \leq M$) – coordinates of one bad square (the upper left square has coordinates $[1, 1]$, the bottom right is $[N, M]$).

**Output**

For each plate in the input file output a single line containing the maximum number of memory chips that can be cut out of the plate.

**Example**

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>6 6 5</td>
<td>4</td>
</tr>
<tr>
<td>1 4</td>
<td></td>
</tr>
<tr>
<td>4 6</td>
<td></td>
</tr>
<tr>
<td>2 2</td>
<td></td>
</tr>
<tr>
<td>3 6</td>
<td></td>
</tr>
<tr>
<td>6 4</td>
<td></td>
</tr>
<tr>
<td>6 5 4</td>
<td></td>
</tr>
<tr>
<td>3 3</td>
<td></td>
</tr>
<tr>
<td>6 1</td>
<td></td>
</tr>
<tr>
<td>6 2</td>
<td></td>
</tr>
<tr>
<td>6 4</td>
<td></td>
</tr>
</tbody>
</table>

![Diagram of memory chips]

![Diagram of memory chips]
In the whole history of mankind one can find several curious battles, like the following one in France, in 1747...

There was a fortress in Bassignac-le-Haut, a small village lying on the left bank of river Dordogne, just over the Chastang dam. From the dam up to the fortress there was a wide staircase made out of red marble. One day in the morning, the guard spotted a large battalion approaching the fortress, with a dreaded leader – The Conqueror.

When The Conqueror reached the fortress, he was already awaited by its commandant. As the commandant had only a small part of his soldiery available, he proposed to The Conqueror: "I see that you have many soldiers behind you, standing on the stairs. We can play a small 'game': In each round, you will divide your soldiers into two groups in an arbitrary way. Then I will decide which one of them stays and which one goes home. Each soldier that stays will then move up one stair. If at least one of your soldiers reaches the uppermost stair, you will be the winner, in the other case, you will be the loser. And your destination will be the dam down there...", added the commandant, pointing to the Chastang dam by his hand.

The Conqueror immediately liked this game, so he agreed and started to 'conquer'.

**Task**

Your role is The Conqueror’s now. There are \( N \) stairs to the fortress (\( 2 \leq N \leq 2000 \)) and you have at most \( 1000000000 \) soldiers. For each stair, you are given the number of soldiers standing on it, with number 1 being the uppermost stair and \( N \) the bottom one. None of your soldiers stands on stair 1 at the beginning.

For each starting position given to your program, if the position is winning (i.e. there is a strategy that enables you to win the game regardless of your opponent’s moves), your program should win. Otherwise it should just play the game (and lose) correctly.

This is an interactive problem; you will play against a library as specified below. In each round, your program will describe a group of soldiers to our library. The library returns 1 or 2 specifying which group of soldiers should stay (1 means the group you described, 2 means the rest of the soldiers). In case the game ends (either because you won or there are no more soldiers in the game), the library will terminate your program correctly. Your program may not terminate in any other way.

**Library interface**

The library `libconq` provides two routines:

- `start` – returns the number \( N \) and fills an array `stairs` with numbers of soldiers standing on the stairs (i.e. there are `stairs[i]` soldiers standing on stair \( i \))
• step – takes an array subset (with at least $N$ elements\(^1\)), describing the group of soldiers you chose, and returns 1 or 2 as described above; the group of soldiers is specified by numbers of soldiers on each stair, as in the start function.

If you fail to specify a valid group of soldiers, the game will be terminated and your program will score zero points for the particular test case. **Please note that also in C/C++ the stairs are numbered starting from 1.**

Following are the declarations of these routines in FreePascal and C/C++:

```plaintext
FreePascal example:
uses libconq;
var stairs: array[1..2000] of longint;
subset: array[1..2000] of longint;
i,N,result: longint;
...
start(N,stairs);
...
for i:=1 to N do
  if random(2)=0 then subset[i]:=0
  else subset[i]:=stairs[i];
result:=step(subset);
...

C/C++ example:
#include "libconq.h"
int stairs[2001];
int subset[2001];
int i,N,result;
...
start(&N, stairs);
...
for (i=1;i<=N;i++)
  if (rand()%2==0) subset[i]=0;
  else subset[i]=stairs[i];
result=step(subset);
...
```

You have to link the library to your program – by uses libconq; in FreePascal and by #include "libconq.h" in C/C++, where you have to compile your program by adding libconq.c to the compiler arguments.

\(^1\) $N + 1$ elements in C/C++, see below
An example of the game

<table>
<thead>
<tr>
<th>You:</th>
<th>Library:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>start(N,stairs)</code></td>
<td>N=8, stairs=(0,1,0,3,3,4,0)</td>
</tr>
<tr>
<td><code>step((0,1,0,0,1,0,1,0))</code></td>
<td>returns 2</td>
</tr>
<tr>
<td><code>step((0,1,0,0,1,0,1,0))</code></td>
<td>returns 2</td>
</tr>
<tr>
<td><code>step((0,0,0,3,2,0,0,0))</code></td>
<td>returns 1</td>
</tr>
<tr>
<td><code>step((0,0,2,0,0,0,0,0))</code></td>
<td>returns 2</td>
</tr>
<tr>
<td><code>step((0,1,0,0,0,0,0,0))</code></td>
<td>returns 2</td>
</tr>
<tr>
<td><code>step((0,1,0,0,0,0,0,0))</code></td>
<td>no return: you won</td>
</tr>
</tbody>
</table>

Resources

On the web page you may find example libraries for both C/C++ and FreePascal. These libraries are different from those that will be used during testing. You may use them to make sure your library calls are correct. The example library reads the input from the file `libconq.dat`, containing two lines. On the first line is the number $N$ of stairs, the second line contains $N$ integers – the numbers of soldiers on each of the stairs $1..N$.

The file `libconq.dat` for the example above would look like this:

```
8
0 1 1 0 3 3 4 0
```
A decorative fence

Input file: fence.in
Output file: fence.out
Source Code: fence.pas/.c/.cpp

100 points
Time limit: 1 s
Memory limit: 1 MB

Richard just finished building his new house. Now the only thing the house misses is a cute little wooden fence. He had no idea how to make a wooden fence, so he decided to order one. Somehow he got his hands on the ACME\textsuperscript{2} Fence Catalogue 2002, the ultimate resource on cute little wooden fences. After reading its preface he already knew, what makes a little wooden fence cute.

A wooden fence consists of $N$ wooden planks, placed vertically in a row next to each other. A fence looks cute if and only if the following conditions are met:

- The planks have different lengths, namely $1, 2, \ldots, N$ plank length units.
- Each plank with two neighbors is either larger than each of its neighbors or smaller than each of them. (Note that this makes the top of the fence alternately rise and fall.)

It follows, that we may uniquely describe each cute fence with $N$ planks as a permutation $a_1, \ldots, a_N$ of the numbers $1, \ldots, N$ such that $(\forall i; 1 < i < N) (a_i - a_{i-1}) \ast (a_i - a_{i+1}) > 0$ and vice versa, each such permutation describes a cute fence.

It is obvious, that there are many different cute wooden fences made of $N$ planks. To bring some order into their catalogue, the sales manager of ACME decided to order them in the following way: Fence $A$ (represented by the permutation $a_1, \ldots, a_N$) is in the catalogue before fence $B$ (represented by $b_1, \ldots, b_N$) if and only if there exists such $i$, that $(\forall j < i) a_j = b_j$ and $(a_i < b_i)$. (Also to decide, which of the two fences is earlier in the catalogue, take their corresponding permutations, find the first place on which they differ and compare the values on this place.) All the cute fences with $N$ planks are numbered (starting from 1) in the order they appear in the catalogue. This number is called their catalogue number.

After carefully examining all the cute little wooden fences, Richard decided to order some of them. For each of them he noted the number of its planks and its catalogue number. Later, as he met his friends, he wanted to show them the fences he ordered, but

\textsuperscript{2}A Company Making Everything
he lost the catalogue somewhere. The only thing he has got are his notes. Please help him find out, how will his fences look like.

**Input**

The first line of the input file contains the number $K$ ($1 \leq K \leq 100$) of input data sets. $K$ lines follow, each of them describes one input data set.

Each of the following $K$ lines contains two integers $N$ and $C$ ($1 \leq N \leq 20$), separated by a space. $N$ is the number of planks in the fence, $C$ is the catalogue number of the fence.

You may assume, that the total number of cute little wooden fences with 20 planks fits into a 64-bit signed integer variable (`long long` in C/C++, `int64` in FreePascal). You may also assume that the input is correct, in particular that $C$ is at least 1 and it doesn’t exceed the number of cute fences with $N$ planks.

**Output**

For each input data set output one line, describing the $C$-th fence with $N$ planks in the catalogue. More precisely, if the fence is described by the permutation $a_1, \ldots, a_N$, then the corresponding line of the output file should contain the numbers $a_i$ (in the correct order), separated by single spaces.

**Example**

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1 2</td>
</tr>
<tr>
<td>2 1</td>
<td>2 3 1</td>
</tr>
<tr>
<td>3 3</td>
<td></td>
</tr>
</tbody>
</table>
A highway and the seven dwarfs

Input file: standard input 100 points
Output file: standard output Time limit: 5 s
Source Code: dwarfs.pas/.c/.cpp Memory limit: 16 MB

Once upon a time, there was a land where several families of dwarfs were living. This land was called Dwarfland. Each family lived in one house. Dwarfs were often visiting their friends from the other families. Because the Dwarfland was free of evil, it happened that each dwarf visited each other during some short period of time.

Once, the humans living in countries around Dwarfland decided to build several straight highways. As the humans weren’t aware of the dwarfs, some of the planned highways passed through Dwarfland. The dwarfs discovered this and were quite unhappy about it. The dwarfs are very little, and also very slow, so they are unable to cross the highway safely.

The dwarfs managed to get the plans for the highways somehow, and now they need your help. They would like to keep on visiting each other, so they don’t like those highways which divide their houses into two non-empty parts. After they find out which highways they don’t like, they will magically prevent the humans from building them.

The dwarfs are very little, and cannot reach the keyboard. So they asked for your help.

Task

Given is a number N of points (houses) in the plane and several straight lines (highways). For each given line, your task is to determine whether all N points lie on the same side of the line or not. Your program has to output the answer for the currently processed line before reading the description of the next one. You may assume that no highway passes through any of the houses.

Input and output description

Your program is supposed to read the input from the standard input (stdin in C/C++, input in FreePascal) and write its output to the standard output (stdout in C/C++, output in FreePascal). The first line of the input contains one integer N (0 ≤ N ≤ 100 000). N lines follow, the i-th of them contains two real numbers xi, yi (−109 ≤ xi, yi ≤ 109) separated by a single space – the coordinates of the i-th house.

Each of the following lines contains four real numbers X1, Y1, X2, Y2 (−109 ≤ X1, Y1, X2, Y2 ≤ 109) separated by a single space. These numbers are the coordinates of two different points [X1, Y1] and [X2, Y2], lying on the highway. For each line of input, your program is supposed to output a line containing the string “GOOD” if all of the given points are on the same side of the given line, or “BAD” if the given line divides the points. After writing out each line of the output, your program should flush the output buffer. In the following sections you may find an example on how to do this.

We will terminate your program after it gives the answer for the last highway. Your program is not supposed to terminate by itself. You may assume that there will be no more than 100 000 highways.
### Input and output routines in C/C++

Reading one line (note that there is no space after the last %lf):

```c
double X_1, Y_1, X_2, Y_2;
scanf(" %lf %lf %lf %lf", &X_1, &Y_1, &X_2, &Y_2);
```

Writing the output for one line:

```c
printf("GOOD\n"); fflush(stdout);
```

### Input and output routines in FreePascal

Reading one line:

```pascal
var X_1, Y_1, X_2, Y_2 : double;
read(X_1,Y_1,X_2,Y_2);
```

Writing the output for one line:

```pascal
writeln('GOOD'); flush(output);
```

### Warning

You are advised to use the `double` data type (in both C/C++ and FreePascal) to store real numbers. Note that when using real number arithmetics, rounding errors may occur. If you want to test, whether two real numbers $x$, $y$ are equal, don’t test whether $x = y$ but whether $|x - y| < \varepsilon$ (where $\varepsilon$ is a small constant, $10^{-4}$ will suffice).

### Example

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>GOOD</td>
</tr>
<tr>
<td>0.0 0</td>
<td>BAD</td>
</tr>
<tr>
<td>6.00 -0.001</td>
<td>BAD</td>
</tr>
<tr>
<td>3.125 4.747</td>
<td></td>
</tr>
<tr>
<td>4.747 0.47</td>
<td></td>
</tr>
<tr>
<td>5 3 7 0</td>
<td></td>
</tr>
<tr>
<td>4 -4.7 7 4.7</td>
<td></td>
</tr>
<tr>
<td>4 47 4 94</td>
<td></td>
</tr>
</tbody>
</table>
Input file: guards.in  
Output file: guards.out  
Source Code: guards.pas/.c/.cpp

Once upon a time, there was a kingdom. It had everything a kingdom needs, namely a king and his castle. The ground-plan of the castle was a rectangle that was divided into $M \times N$ unit squares. Some of the squares are walls, some of them are free. We will call each of the free squares a room. The king of our kingdom was extremely paranoid, so one day he decided to make hidden pits (with alligators at the bottom) in some of the rooms.

But this was still not enough. One week later, he decided to place as many guards as possible inside his castle. However, this won’t be so simple. The guards are trained so that immediately after they see someone, they shoot at him. And so the king has to place the guards carefully, because if two guards would see each other, they would shoot at themselves! Also evidently the king can’t place a guard into a room with a pit.

Two guards in a room see each other, so each room may contain at most one guard. Two guards in different rooms see each other if and only if the squares corresponding to their rooms are in the same row or in the same column of the plan of the castle and there is no wall between them. (The guard can see only in four directions, much like a rook in chess.)

Task

Your task is to find out, how many guards can the king place inside his castle (according to the rules above) and to find one possible assignment of that many guards into the rooms.

Input

The first line of the input file contains two numbers $M$, $N$ (1 ≤ $M$, $N$ ≤ 200) – the dimensions of the ground-plan of the castle. The $i$-th of the following $M$ lines contains $N$ numbers $a_{i,1}, \ldots, a_{i,N}$, separated by single spaces, where:

- $a_{i,j} = 0$ means that the square $[i,j]$ is free (a room without a pit)
- $a_{i,j} = 1$ means that the square $[i,j]$ contains a pit
- $a_{i,j} = 2$ means that the square $[i,j]$ is a wall

Note that the first coordinate of a square is the row and the second one is the column.

Output

The first line of the output file should contain the maximum number $K$ of guards the king may place inside his castle. The following $K$ lines should contain one possible assignment of $K$ guards into the free rooms of the castle so that no two guards would see each other.

More precisely, the $i$-th of these lines should contain two integers $r_i$, $c_i$ separated by a single space – the coordinates of the room where $i$-th guard will be placed ($r_i$ is the row and $c_i$ is the column).
Castle from the example input and one possible correct output.

**Example**

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 4</td>
<td>2</td>
</tr>
<tr>
<td>2 0 0 0</td>
<td>1 2</td>
</tr>
<tr>
<td>2 2 2 1</td>
<td>3 3</td>
</tr>
<tr>
<td>0 1 0 2</td>
<td></td>
</tr>
</tbody>
</table>
Birthday party

Input Files: party*.in 100 points
Output Files: party*.out

John’s birthday is approaching slowly, and as every year, John is going to organize a great garden party. He wants all his friends to come, but (sadly) he knows, that it is almost impossible. For example Susie left Steve last week, and it will be almost impossible to make both of them come. John spent most of the last week visiting his friends and asking them to come. He got some promises, but even more requests. (‘If you invite me, you just have to invite my boyfriend!’ exclaimed Veronica. ‘If you invite the Burdiliak twins, don’t expect me or Joseph to come!’ stated Peter.) Suddenly, John realized, that it will be quite hard just to satisfy all the requests he got.

Task
You are given a description of the requests John got from his friends. Your task is to find a group of people such that if John invites the people in this group (and nobody else) to his party, all the requests he got will be satisfied. The requests are described in the following way:

- name is a request. This request is satisfied if and only if John invites name.
- -name is a request. This request is satisfied if and only if John doesn’t invite name.
  (In both cases, name is a string of at most 20 lowercase letters without spaces.)
- If R1, . . . , Rk are requests, then (R1 & . . . & Rk) is a request. This request is satisfied if and only if all requests R1, . . . , Rk are satisfied.
- If R1, . . . , Rk are requests, then (R1 | . . . | Rk) is a request. This request is satisfied if and only if at least one of the requests R1, . . . , Rk is satisfied.
- If R1, R2 are requests, then (R1 => R2) is a request. This request is not satisfied if and only if R1 is satisfied and R2 is not satisfied.

Input
You can find ten input files, called party1.in to party10.in on the web page. Each of the inputs is worth 10 points.

On the first line of the input file is the number of John’s friends F, next F lines contain their names, one per line. On the next line is the number of requests N. Each of the following N lines contains one request.

Output
For each file partyX.in you have to produce the corresponding output file partyX.out, containing one correct solution. The first line of the output file will be the number K of people John should invite. The following K lines should contain their names, one per
line. You may assume that each of the input files has a (not necessarily unique) solution. If there are more possible solutions, you may output any of them.

**Submits**

Submit the files `party*.out` using the web interface in the same way you submit your programs for the other tasks.

**Example**

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>veronica</td>
<td>steve</td>
</tr>
<tr>
<td>steve</td>
<td>dick</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>(veronica =&gt; dick)</td>
<td></td>
</tr>
<tr>
<td>(steve =&gt; -veronica)</td>
<td></td>
</tr>
<tr>
<td>(steve &amp; dick)</td>
<td></td>
</tr>
</tbody>
</table>
You surely know the country of Absurdistan. It is large, flat and not very civilized. In fact, there are no roads, just one railroad going straight through the entire country. Along the railroad there are some villages. All the other inhabitants live in tents scattered all over the country. If we look at the country from above, Absurdistan would look just like an infinite plane, with the railroad being the x-axis, villages being points on the x-axis and tents being points in the plane.

Last year the government decided to build a road along the railroad. Now, as the road is just finished, the government wants to make the next step in civilizing the country. They decided to build bus stations in some of the villages and in addition to build straight roads from every tent in the country to some bus station. But building the (infinitely long) road has almost drained their budget, so now they have to plan carefully. And that’s where you come in.

**Task**

There are $N$ villages along the road, numbered from 1 to $N$ (not necessarily in the order they lie on the road). The $x$ coordinate of village $i$ is $v_i$. Building a bus station in the village $i$ costs $c_i$ absurdollars. In the country there are $M$ tents. Tent $i$ is standing on the coordinates $x_i$, $y_i$. All coordinates are given in meters. Building one meter of road costs 1 absurdollar.

Your task is to find the cheapest way to satisfy the government’s plans.

**Input**

The first line of the input file contains two integers $N$, $M$ ($1 \leq N \leq 1\,000$, $1 \leq M \leq 1\,000$) – the number of villages and tents. $N$ lines follow, the $i$-th of them contains an integer $v_i$ and a real number $c_i$ ($-10^9 \leq v_i \leq 10^9$, $0 \leq c_i \leq 10^9$) – the coordinate of the $i$-th village and the cost of building a bus station there. Following $M$ lines describe the tents, the $i$-th of them contains two integers $x_i$, $y_i$ ($-10^9 \leq x_i, y_i \leq 10^9$) – the coordinates of the $i$-th tent.

**Output**

The output file should contain one real number giving the minimum cost (in absurdollars) of building the bus stations and straight roads from each tent to one of the stations. Output the number rounded to two decimal places by standard output routines.
Example

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 6</td>
<td>29.89</td>
</tr>
<tr>
<td>0 10.0</td>
<td></td>
</tr>
<tr>
<td>1 20.0</td>
<td></td>
</tr>
<tr>
<td>4 20.0</td>
<td></td>
</tr>
<tr>
<td>8 10.0</td>
<td></td>
</tr>
<tr>
<td>0 1</td>
<td></td>
</tr>
<tr>
<td>1 1</td>
<td></td>
</tr>
<tr>
<td>2 -1</td>
<td></td>
</tr>
<tr>
<td>7 -1</td>
<td></td>
</tr>
<tr>
<td>8 -1</td>
<td></td>
</tr>
<tr>
<td>6 2</td>
<td></td>
</tr>
<tr>
<td>1.0</td>
<td></td>
</tr>
</tbody>
</table>

Explanation

The best solution is to build the bus stations in villages 1 and 4, and to connect tents 1-3 to the station in the village number 1, tents 4-6 to the station in the village number 4. This solution is displayed also in the image.
Travel cheap!

Input file: travel.in 100 points
Output file: travel.out Time limit: 1.5 s
Source Code: travel.pas/.c/.cpp Memory limit: 8 MB

Your mother-in-law is going to visit you tomorrow. You are not too happy about this, but then, who would? And to make matters worse, she is probably going to stay for a longer time. You spent the last night thinking how to avoid this, and finally, at 6am, the saving idea came: buy her a ticket for a trip around the local country! She cannot refuse such a wonderful gift, and you will get rid of her. You fell asleep with a smile on your face. But as you woke up in the morning, you realized, that such ticket will probably be quite expensive.

Task

Suppose you know the number $K$ of days your mother-in-law is going to stay. You are given a description of local railroad network. The network consists of $N$ stations and $M$ trains. Each train connects exactly two stations. Each train goes in both directions every day. For each train, the price of the ticket is given. This price is the same for both directions. Your mother-in-law is already an elderly woman, so travelling is quite exhausting for her. Thus each day she can travel by only one train. You want her to travel every day, otherwise she would get bored and come to visit you instead. Your task is to find the minimum amount of money you need to buy her tickets for the following $K$ days.

Input

The first line of the input file contains the three integers $N$, $M$, $K$ ($2 \leq N \leq 500$, $1 \leq M \leq 10000$, $1 \leq K \leq 1000000$) separated by single spaces. The stations are numbered from 1 to $N$, in the beginning your mother-in-law is at the station 1. The trains are numbered from 1 to $M$.

$M$ lines follow, the $i$-th of them describes the $i$-th train. This line contains three integers $a_i$, $b_i$, $p_i$ ($1 \leq a_i, b_i \leq N$, $0 \leq p_i \leq 1000$) separated by single spaces. The $i$-th train connects the stations $a_i$ and $b_i$, the price of the ticket (for each of the directions) is $p_i$. You may suppose that for at least one train $a_i = 1$ or $b_i = 1$.

Output

The output file should contain a single line with one integer – the minimum amount of money you need to buy the tickets.
Example

Input
7 9 4
1 2 11
1 3 9
2 4 6
3 4 10
3 5 9
4 5 1
4 6 1
5 6 7
5 7 1

Output
19

Explanation
You could for example buy the following tickets: Day 1: 1-2 (price 11), day 2: 2-4 (price 6), day 3: 4-5 (price 1), day 4: 5-7 (price 1).
Once upon a time, professor Andrew was researching a special species of turtles and their manner of life. These turtles had very interesting graphics on their carapaces. The lines on a turtle’s carapaces formed together some positive integer. At the beginning of the research, the scientist had only two turtles. One turtle had number $A$ on its carapace and the other turtle number $B$. Every now and then, some turtle (with number $X$) could meet another, not necessarily different, turtle (with number $Y$).\footnote{Also it could even meet itself.} And then, they had a baby turtle. (There were no differences between male and female turtles.) Of course, the baby was born with a number on its carapace. The baby’s number was $X \cdot Y - X - Y + 2$.

These turtles are extremely rare, and thus extremely expensive. Some customers even want a turtle with their favourite number on its carapace. But sometimes, it may be impossible to breed such a turtle. The professor is good at turtle breeding, but maths and programming have never been his favourite subjects. He now needs you to help him decide, which numbers are possible and which are not.

**Task**

Definition: A number $C$ is a descendant of numbers $A$ and $B$ if and only if one of the following conditions is met:

- $C = A$
- $C = B$
- $C = X \cdot Y - X - Y + 2$, where both $X$ and $Y$ are descendants of $A$ and $B$

You are given two integers $A$, $B$ – the numbers of professor’s first two turtles. For each of the customers’ favourite numbers you have to determine whether it is a descendant of $A$ and $B$ or not.

**Input**

On the first line of the input file, there are two integers $A$ and $B$ ($1 \leq A, B \leq 10\,000$) separated by one space. On the second line, there is one integer $N$ ($1 \leq N \leq 1000$) – the number of favourite numbers. $N$ lines follow, on the $i$-th of them is one integer $C_i$ ($1 \leq C_i \leq 10^{1000}$).

**Output**

The output file will have exactly $N$ lines. On the $i$-th line, there will be one string determining whether the number $C_i$ is a descendant of numbers $A$ and $B$. If $C_i$ is a descendant of $A$ and $B$, the corresponding line should contain the string “YES”, otherwise it should contain the string “NO”.

---

---

---

---
Example

Input          Output
10  15          YES
6            NO
10           YES
1            YES
82           NO
15877         YES
23
1135

Explanation

10  =  10
15  =  15
82  =  10*10 - 10 - 10 + 2
127 =  10*15 - 10 - 15 + 2
1135 = 127*10 - 127 - 10 + 2
15877 = 127*127 - 127 - 127 + 2
Solutions
We present a solution based on dynamic programming with the time complexity $O(MN3^M)$. We note that all the coordinates $[x, y]$ of squares are decreased by one, that is $0 \leq x < N$, $0 \leq y < M$, as is common for C programmers.

**Definition 1:** Let $B = (b_0, b_1, \ldots, b_{M-1})$ be a vector of numbers (called the *border*). We define the *borderset* $S(B)$ as the set of all squares $[x, y]$ of the plate satisfying $x \leq b_y$. Informally, it is the set of those squares that lie to the left of the border $B$. We usually won’t distinguish between the border and its set.

The set of the border $B = (1, 5, 0, -7, 3)$ ($N = 9$, $M = 5$).

**Definition 2:** For each $[i, j]$ ($0 \leq i < N$, $0 \leq j < M$) we define an $[i, j]$-*border* as $B[i, j] = (b_0, b_1, \ldots, b_{M-1})$, where $b_0 = b_1 = \cdots = b_j = i$ and $b_{j+1} = \cdots = b_{M-1} = i - 1$. We will call the set $S(B[i, j])$ an $[i, j]$-*baseline*. In other words $S(B[i, j])$ is the set of the squares of the plate, lying to the left of the square $[x, y]$ or in the same column and above it (including the square $[x, y]$).

**Definition 3:** Let’s have a baseline with the border \( B[i, j] \). Let \( P = (p_0, p_1, \ldots, p_{M-1}) \), \( p_i \in \{0, 1, 2\} \) be a vector. We will denote the vector \((b_0 - p_0, \ldots, b_{M-1} - p_{M-1})\) as \( B - P \). We define a \( P \)-profile for this baseline as the set \( S(B[i, j] - P) \). The symbol 0 will denote the all-components-zero profile and \( e_j \) will denote a profile that has all components zero except for \( p_j = 1 \).

![Diagram of a baseline](image)

The profile \( P = (0, 2, 1, 0, 2) \) of a \([4, 2]\)-baseline.

We can think of a profile as of a number between 0 and \( 3^M - 1 \), written in base 3. We will sometimes use \( P \) as an index of an array in this sample solution. In the implementation, we first convert \( P \) from base 3 into base 10 and then use it as an index.

We can view the plate as the set of its good squares \( G \). The original problem was to determine what is the maximum number of the chips that can be cut out of the plate \( G \).

The basic idea how to solve the problem is to use dynamic programming in the following manner: For each baseline \( B[i, j] \) and for each profile \( P \) we compute \( A[i, j, P] \) – the maximum number of chips that can be cut out of the set (part of the plate) \( G \cap S(B[i, j] - P) \). Note that \( G \cap S(B[N - 1][M - 1] - 0) = G \), also the number \( A[N - 1, M - 1, 0] \) gives us the answer to problem.

The part of the plate \( G \cap S(B[0, j]) \) is too thin to cut any chip out of it, so for the beginning of the table we have \( A[0, j, P] = 0 \), for any \( j \) and any \( P \).

We will process all the other baselines \( B[i, j] \) in the order from left to right (i.e. \( i \) increases) and the baselines with the same \( i \) from top to bottom (i.e. \( j \) increases). For each baseline we consider each profile \( P \).

Let’s have a fixed baseline \( B[i, j] \) and a fixed profile \( P \). There are two possibilities: either \( p_j > 0 \) or \( p_j = 0 \).

If \( p_j > 0 \), then \( G \cap S(B[i, j] - P) = G \cap S(B' - P') \), where \( B' = S(B[i', j']) \) is the previous baseline in the order we process them and \( P' = P - e_j \). Because these two sets are equal, the maximum number of chips that can be cut out of them must also be equal. Therefore \( A[i, j, P] = A[i', j', P'] \).

If \( p_j = 0 \), there are three possibilities how to obtain the desired maximum of chips, that can be cut out of \( G \cap S(B - P) \).

- We cut no chip having the lower right corner at the position \([i, j]\).
We cut a horizontal \((3 \times 2)\) chip having the lower right corner at the position \([i, j]\).

We cut a vertical \((2 \times 3)\) chip having the lower right corner at the position \([i, j]\).

The maximum number of the chips corresponding to the first case is \(A[i', j', P]\), where \(S(B[i', j'])\) is the previous baseline in the order we process them.

The maximum number of the chips corresponding to the second case is \(A[i'', j'', P + 2e_j + 2e_{j-1}]\), where \(S(B[i'', j''])\) is the second previous baseline in the order we process them.

We are processing the baseline \(S(B[4, 3])\) with the profile \(P = (0, 1, 0, 0, 2)\).

It is possible to cut a horizontal chip here.

The second previous baseline is \(S(B[4, 1])\), the new profile will be \(P'' = (0, 1, 2, 2, 2)\).

Note that \(S(B[4, 1] - P'')\) is exactly \(S(B[4, 3] - P)\) without the horizontal chip.

The maximum number of the chips corresponding to the third case is \(A[i'', j'', P + e_j + e_{j-1} + e_{j-2}]\), where \(S(B[i'', j''])\) is the third previous baseline in the order we process them.

Clearly \(A[i, j, P]\) will be the maximum of these three numbers. (We consider the second and third case only if the corresponding chip can be cut out at this position.)

The test whether a horizontal or a vertical chip can be cut out at some position is easy. A horizontal chip can be cut out iff there are no bad squares at those positions and \(i \geq 2, j \geq 1\) and \(p_j = p_{j-1} = 0\). Similarly a vertical chip can be cut out iff there are no bad squares at those positions and \(i \geq 1, j \geq 2\) and \(p_j = p_{j-1} = p_{j-2} = 0\). Thus this test takes only a constant amount of time.

It can be easily shown that we have to remember the values \(A[i, j, P]\) only for the last four baselines (the one being computed and the three previous ones). There is only \(3^M \leq 3^{10} < 60000\) profiles for each baseline, so this will easily fit into memory.

The time complexity of the algorithm presented here is as promised \(O(MN3^M)\), because there are exactly \(3^M\) profiles and \(O(MN)\) baselines.
Conqueror’s battalion

Author(s) of the problem: unknown
Contest-related materials by: Vladimír Koutný, Michal Forišek

Let’s try to tell somehow how good a position is. If we have two soldiers on the same stair (and no other soldiers), in the next round we will have at most only one soldier but one stair higher. In this way, one soldier on the stair $S$ is equivalent to two soldiers on the stair $S + 1$. From now on a soldier on the stair $S$ will have the value $2^{N-S}$. The value of a position will be the sum of the values of all the soldiers. Note that all positions, where the Conqueror has won, have the value at least $2^N - 1$ because there is a soldier on the stair number 1.

Losing positions

If the value of the position is less than $2^N - 1$ and the commandant plays optimally, the Conqueror will lose.

Proof: If the value is less than $2^N - 1$, the Conqueror didn’t win yet. Let’s take a look at one round of the game. The Conqueror divides his soldiers in some way. The group with the smaller (or equal) value has to have the value less than $2^{N-2}$. The commandant will choose this group to stay and the other to leave. When any soldier moves up one stair, his value doubles. Therefore the value of the new position will be less than $2^{N-2} = 2^{N-1}$ and the number of soldiers will decrease. There is a finite number of soldiers, and so the game ends in a finite number of moves. The value of a position will always be less than $2^N - 1$, also at the end there will be no soldiers and the Conqueror will lose, q.e.d.

Winning positions

If the value of the position is at least $2^N - 1$ and the Conqueror plays optimally, he will win.

Proof: The Conqueror has to divide the soldiers into two groups so that the value of each group will be at least $2^{N-2}$. Then regardless of the commandant’s choice the new position will again have the value at least $2^{N-1}$ and there will be less soldiers than before. There is a finite number of soldiers, and so the game will end in a finite number of moves. At the end of the game the value of the position will be at least $2^{N-1}$. This means that some of the soldiers has to stand on the stair 1 and the Conqueror won.

We only need to show, that the Conqueror can always divide the soldiers in the way described above.

Lemma: Let $a_1, \ldots, a_M$ ($2^{N-2} \geq a_1 \geq \ldots \geq a_M$, $M \geq 2$) be powers of two with $\sum_{i=1}^{N} a_i \geq 2^{N-2}$. Then for some $k$ holds $\sum_{i=1}^{k} a_i = 2^{N-2}$.

Proof of the lemma: Induction on $M$. For $M = 2$ the lemma holds. Now let $M > 2$ and $\sum_{i=1}^{M} a_i > 2^{N-2}$. Since the sum and $2^{N-2}$ are multiples of $a_M$, the sum is at least $2^{N-2} + a_M$. This means that $\sum_{i=1}^{M-1} a_i \geq 2^{N-2}$ and we may apply the induction assumption.

From the lemma we get that if the sum of the soldiers’ values is at least $2^{N-1}$, we may select the first (e.g. closest to the top of the staircase) few of them so that the sum of
their values will be exactly $2^{N-2}$. The Conqueror may also divide his soldiers into these
two groups, q.e.d.

The solution is a straightforward implementation of the ideas above. We will have to implement arithmetics with big numbers to count the value of a position. If we are in a losing position, we choose an empty set and loose instantly. If we are in a winning position, we find the place where to split the soldiers by adding the values of soldiers on stairs 1, 2, \ldots until the value reaches $2^{N-2}$. Each operation with the big numbers we need can be implemented to run in $O(N)$ time, also the time required for one move is $O(N^2)$.

We could pre-compute some data at the beginning of the game to have a faster algorithm. Let $s_i$ be the number of soldiers on the stair $i$. We will pre-compute the sums $T_k = \sum_{i=1}^{k} s_i 2^{N-i}$. Let’s number the soldiers from 1 to some $M$ ordered by the stair, where they stand at the beginning of the game. If we play the game as described above, we may always split the soldiers so that both groups have a consecutive set of numbers. We will keep track of the current numbers of soldiers on each stair and we will keep two integers pointing to the uppermost and lowermost soldier. In fact, we won’t move the soldiers upwards, because we know, that after $r$ rounds they are exactly $r$ stairs higher.

From the partial sums we have we are able to compute the value of any consecutive set of soldiers in $O(N)$ time. Using this information we may binary search the place where to split the soldiers. This solution also requires $O(N^2)$ time for pre-computing the partial sums and then $O(N \log N)$ time for each move.

The library your programs played against used the commandant’s strategy described above, with the difference that if both possibilities led to a winning position for your program, the library chose the one of them in which the uppermost soldier was on a lower stair.
A decorative fence

Author(s) of the problem: Michal Forišek
Contest-related materials by: Richard Kráľovič, Jana Gajdošíková, Marián Dvorský

This problem could be solved using dynamic programming. Let $T_{N,i}^{up}$ be the number of permutations that describe some cute fence of length $N$ (let’s call them fence permutations) starting with the element $i$ and having the second element greater than the first one. Analogically, $T_{N,i}^{down}$ will be the number of fence permutations of length $N$ starting with the element $i$ and having the second element less than $i$. If the second element of a permutation is greater than the first one, we will say that the permutation is going up, otherwise it is going down.

We can compute the values $T_i$ for all $N$ from 1 to 20, for all appropriate $i$ and for both $up$ and $down$ in advance from the following facts:

- $T_{N,1}^{down} = 0$
- $T_{N,j+1}^{down} = \sum_{k=1}^{j} T_{N-1,k}^{up} = T_{N,j}^{down} + T_{N-1,j}^{up}$
- $T_{N,i}^{up} = T_{N,N+1-i}^{down}$

We get the second equation as follows: Take all the fence permutations of length $N$, starting with $j + 1$ and going down. The second element $k$ of this permutation is a number between 1 and $j$. If we take our permutation without its first element ($j + 1$) and decrease all elements greater than $j$ by 1, we get some fence permutation of length $N - 1$, starting with $k$ and going up. It is easy to verify, that this is a bijection between the set of fence permutations of length $N$, starting with $j + 1$ and going down, and the set of fence permutations of length $N - 1$, starting with an element less than $j + 1$ and going up. Thus the two sets have the same cardinality.

Values of $T$ could be also hardwired as constants into the solution, but it does not make much sense for $N \leq 20$ as computing them doesn’t take much time and we do it only once. These values will help us find the fence permutation with the catalogue number $C$.

There are $T_{N,i}^{up} + T_{N,i}^{down}$ fence permutations starting with $i$. Now we can easily determine the first element $a_1$ in the $C$-th fence permutation. Fence permutations starting with 1 have the catalogue numbers from 1 to $T_{N,1}^{up} + T_{N,1}^{down}$; those starting with 2 have the catalogue numbers from $T_{N,1}^{up} + T_{N,1}^{down} + 1$ to $T_{N,1}^{up} + T_{N,2}^{down} + T_{N,2}^{up} + T_{N,2}^{down}$, and so on. And so $a_1$ is the smallest number, for which $\sum_{i=1}^{a_1} \left( T_{N,i}^{down} + T_{N,i}^{up} \right) \geq C$.

Next, we will find the second element $a_2$. Suppose we decrease all elements of the permutation we seek, that are greater than $a_1$, by 1. Then the rest of the permutation will be a fence permutation of length $N - 1$. We know that if its first element is less than $a_1$, it has to go up, otherwise it has to go down. We also know that the fence permutation we seek is the $C_1 = \left( C - \sum_{i=1}^{a_1-1} \left( T_{N,i}^{down} + T_{N,i}^{up} \right) \right)$-th of all such fence permutations.

For $x$ from 1 to $a_1 - 1$ there are $T_{N-1,x}^{up}$ such permutations, starting with $x$, for greater $x$ there are $T_{N-1,x}^{down}$ such permutations. So we find the first element $y$ of this permutation in the same way as above. If $y < a_1$ then $a_2 = y$ else $a_2 = y + 1$. 

30
Now we already know whether the permutation we seek goes up or down and we know its first two elements. There probably are more such permutations and we know the catalogue number $C_2$ of our permutation between all possible ones. From now on determining the next element will be done in the same way.

Suppose we know the first $k$ ($k \geq 2$) elements of our permutation and its catalogue number $C_k$ between all such permutations. How to determine the $(k + 1)$-th element?

We will view the rest of the permutation as a permutation of a smaller length (with appropriately decreased elements). Last two known elements determine whether this permutation is going up or down. We then find the first element of the modified permutation in a similar way than above (we count ‘good’ permutations starting with $x$ until we reach $C_k$) and then appropriately increase it to get the next element of our permutation. Finally we compute the new value $C_{k+1}$ by subtracting the count of permutations starting with a lower element from $C_k$.

In our implementation, the program always works with permutations of numbers $0..m-1$ for some $m$, instead of permutations of some $m$ numbers from range $1..N$. It is also necessary to ‘decode’ these permutations. Number $i$ will be decoded to $i + 1$-th smallest number from $1..N$, that was not used before.

Both determining the elements of the permutation and decoding the permutation takes $O(N^2)$ time. Precomputing the values of $T$ takes the same time. Memory needed for the values of $T$ is also $O(N^2)$. 
A highway and the seven dwarfs

Author(s) of the problem: Michal Forišek
Contest-related materials by: Ján Oravec, Richard Kráfolvič

First of all note that a highway is good iff it doesn’t intersect the convex hull of all dwarfs’ houses (if it does intersect the convex hull, it intersects its edge and the vertices of this edge are two houses lying on different sides of the highway). After reading the coordinates of the $N$ houses we may pre-compute their convex hull, e.g. using some well-known algorithm running in $O(N \log N)$ time.

Now we will be given many lines and for each of them we have to decide, whether it intersects a given convex polygon. How to do this in a reasonable time? Suppose we have a line with some fixed direction. According to this direction, some of the vertices of the polygon is the leftmost one and some of them is the rightmost one. Clearly the line intersects the polygon iff these two points lie in different half-planes. Also it would suffice if we could find the leftmost and the rightmost point for a given direction.

There are a few different methods how to find them in $O(\log N)$ time, we will present one that involves some more pre-computation, but is in our opinion easiest to implement.

Suppose we have a fixed direction and we know the leftmost and the rightmost point for this direction. If we now rotate the direction by a small angle, it is very likely that the leftmost and the rightmost point won’t change. Let’s take a closer look on when does a change occur. We’ll draw two lines with our direction, one of them passing through the leftmost point and one through the rightmost one.

When we rotate the direction, the leftmost and rightmost point won’t change until one of the sides of the convex hull will lie on one of the two lines. In this moment the other end of this side will become the new left/rightmost point. Also the two points we seek change only when the direction of the line is the direction of one of the convex polygon’s sides.

There polygon has $O(N)$ sides and their directions divide the set of all directions (e.g. angles of the line with the $x$-axis) into $O(N)$ intervals. For each of them we compute the leftmost and the rightmost point using the algorithm described above. This phase can be implemented in $O(N)$ time.

Now if we get a line, we simply compute its direction and use binary search to find out its interval (and the corresponding two points) in $O(\log N)$. If the number of lines is $M$, the total time complexity of this algorithm is $O((M+N) \log N)$. 
Royal guards

Author(s) of the problem: Michal Forišek, Martin Pál
Contest-related materials by: Marián Dvorský, Michal Forišek

For easier implementation, let the castle be surrounded by walls. A row segment will be each continuous part of some row that doesn’t contain a wall and cannot be extended in any direction (e.g. immediately to the left and to the right of it are walls). Similarly we define a column segment. Clearly each segment may contain at most one guard.

Let’s build a bipartite graph where vertices of one partition correspond to the row segments, vertices of the other partition correspond to the column segments and two vertices are connected by an edge iff the corresponding row segment and column segment intersect and their intersection doesn’t contain a pit. In other words the edges correspond to the squares where a guard may stand.

Now consider any valid placement of guards. Then the edges corresponding to places where guards stand form a matching in our graph – each row segment and each column segment contains at most one guard, and thus each vertex incides with at most one edge. On the other hand each matching in our graph corresponds to some valid placement of the guards.

In other words, to place the maximum number of guards into the castle, it suffices to find a maximum matching in the graph we constructed above. There is a well-known algorithm to do this in $O(|V|(|V| + |E|))$ time, based on repeated construction of an augmenting path and exchanging matched vs. unmatched edges along it. As $|V|, |E| = O(MN)$, this algorithm has the time complexity $O(M^2N^2)$. 
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 $A, B, \ldots$ will denote arbitrary logical formulas, not only variables. We will use $\neg$ 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 $\neg(A \lor B)$). You have probably realized that the operator $\&$ was logical and (we will call it a conjunction of the variables), $\lor$ was logical or (a disjunction) and $\Rightarrow$ was an implication.

As a most basic fact note that the formulas $(A \Rightarrow B), \neg A \lor B$ and $\neg B \Rightarrow \neg A$ are equivalent. As a consequence, the formula $(A \Rightarrow \neg A)$ is true iff $A$ is false. The formula $(A \Rightarrow (B \Rightarrow C))$ is equivalent to $((A \& B) \Rightarrow C)$. We will also need the de Morgan’s rules:

- $\neg(A_1 \& \ldots \& A_m)$ is equivalent to $\neg A_1 \lor \ldots \lor \neg A_m$
- $\neg(A_1 \mid \ldots \mid A_m)$ is equivalent to $\neg A_1 \& \ldots \& \neg A_m$.

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

- $(A_1 \Rightarrow (A_2 \Rightarrow (\ldots (A_m \Rightarrow (B_1 \mid \ldots \mid B_n))\ldots)))$
- $((A_1 \& \ldots \& A_m) \Rightarrow (B_1 \mid \ldots \mid B_n))$
- $((-A_1 \& \ldots \& -A_m) \mid (B_1 \mid \ldots \mid B_n))$
- $(-A_1 \mid \ldots \mid -A_m \mid B_1 \mid \ldots \mid B_n)$

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
\((A \lor B) \land (B \lor -C \lor C) \land -D \land (A \lor C)\)
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 
\((\text{lit1} \lor \text{lit2} \lor \ldots \lor \text{litK})\), where each \text{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
\((A \Rightarrow -A)\) and \((-B \Rightarrow B)\). 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 \((A \lor B)\) becomes \((-A \Rightarrow B)\). 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 \((-B \Rightarrow A)\), 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.

\[
\begin{align*}
\text{Our graph for the following formulas:} & \quad (A \Rightarrow -B), (B \lor -C), (B \lor C), (D \Rightarrow -B) \text{ and } (C \lor D). \\
\end{align*}
\]

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 straightforward 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.
Building roads

Author(s) of the problem: Martin Pál
Contest-related materials by: Richard Kráľovič, Michal Forišek

In our solution we will use a dynamic programming approach. There are more possible solutions with roughly the same time complexity, we chose the one of them that is in our opinion easiest to implement.

First of all, let’s sort the villages and also the tents according to their $x$ coordinates. Let’s renumber the villages from 1 to $N$ and the tents from 1 to $M$ in the sorted order.

Note that if we decide where to build the bus stations, the rest is straightforward – each tent will be connected to the nearest station. It follows that the tents connected to one particular station have consecutive numbers.

We will denote the distance between the village $v$ and the tent $t$ as $D(v, t)$ and the cost of building a bus station in the village $v$ as $C(v)$. Now let $P(v, t)$ be the best solution for villages from 1 to $v$ and tents from 1 to $t$. The values $P(v, 0)$ are all zero, the values $P(0, t)$ are $\infty$ for $t > 0$.

Now suppose we know all the values $P(v, t)$ for some fixed $v$. How to compute the values $P(v + 1, t)$? We know $P(v + 1, 0)$. If we know $P(v + 1, t)$, we may compute $P(v + 1, t + 1)$ as follows: In the best solution for villages from 1 to $v + 1$ and tents from 1 to $t + 1$ either there is a bus station in the village $v + 1$ or it is not. If there is not a bus station, the best solution is $P(v, t + 1)$. If there is a bus station, there has to be some $x$ such that in the best solution exactly the tents from $x$ to $t + 1$ are connected to the village $v + 1$. The following recurrent formula follows:

$$P(v + 1, t + 1) = \min \left( P(v + 1, t), \min_{x=1..(t+1)} \left( P(v, x - 1) + C(v + 1) + \sum_{i=x}^{t+1} D(v + 1, i) \right) \right)$$

Determining one particular value $P(v, t)$ takes $O(M)$ time, we have to determine $O(MN)$ such values, therefore the time complexity of this solution is $O(NM^2)$. 
Travel cheap!

Author(s) of the problem: Michal Forišek
Contest-related materials by: Branislav Katreniak, Marián Dvorský

Given was a weighted undirected graph $G$ with $N$ vertices and $M$ edges. We will denote the weight of an edge $e$ as $w(e)$. The task was to find the length of the shortest walk in $G$, having $K$ edges and starting in the vertex 1. We will call each such walk the $K$-step shortest walk.

How does the $K$-step shortest walk $W$ look like? Let $e_i$ be the $i$-th edge of the walk. Now suppose that $e_x$ (with end vertices $u$, $v$) is the cheapest of all the edges in the walk. If there are more equally cheap edges, let $e_x$ be the first of them. Now if we replace the edges $e_{x+1}$ to $e_{N}$ by $e_x$, we get another walk that isn’t more expensive. Thus it is also a $K$-step shortest walk.

The new walk will look like this: We walk along the first $x$ edges of $W$ and then walk back and forth between $u$ and $v$ until we make $K$ steps. The edges $e_1$ to $e_{x-1}$ of the new walk form a path into one of the vertices $u$, $v$. We know that there is a $K$-step shortest walk having the form described above, therefore it suffices to find the length of the shortest of such walks.

As we already stated above, the $K$-step shortest walk will consist of two parts. We may describe them informally as “follow a path into some vertex $v$” and “walk along an edge incident with $v$”. Let’s choose a fixed vertex $v$. The edge will obviously be one of the cheapest of the edges incident with $v$, let’s call any of these edges $e_v$.

Let $l$ be the number of edges of the $1-v$ path in the first part. The number $l$ is between 0 and $N-1$, because a path may visit each vertex at most once. Also, $l$ is at most $K$. Let’s consider a fixed $l$. If we knew the length $d_{v,l}$ of the shortest $1-v$ path having exactly $l$ edges, the $K$-step shortest walk (for this fixed $v$ and $l$) has the length $d_{v,l} + (K-l)w(e_v)$.

But then the solution to our problem is the minimum (through all $v$ and $l$) of $d_{v,l} + (K-l)w(e_v)$. The only problem left is to determine the numbers $d_{v,l}$.

To do this, we will use the Bellman-Ford algorithm that works as follows: In the beginning, we know the numbers $d_{v,0}$: $d_{1,0}$ is 0, all other are $\infty$ (this means that no such path exists; in the implementation we use a large constant, so that these numbers don’t change the minimum length of the walk we seek). Now suppose we know all the values $d_{v,l}$ for some $l$. How to compute the values $d_{v,l+1}$?

Take any shortest $1-v$ path with $l+1$ edges, let its last edge be $uv$. Then the first $l$ edges of this path form some shortest $1-u$ path. Therefore to find the values $d_{v,l+1}$ it suffices to do the following: First, we set all the values equal to $\infty$. Then we process the edges of the graph in an arbitrary order, and for each edge $xy$ we check:

- if $d_{x,l} + w(xy) < d_{y,l+1}$, set $d_{y,l+1}$ equal to $d_{x,l} + w(xy)$
- if $d_{y,l} + w(xy) < d_{x,l+1}$, set $d_{x,l+1}$ equal to $d_{y,l} + w(xy)$

Note that $\min_{l \in \{0,\ldots,N-1\}} d_{v,l}$ is the length of the shortest $1-v$ path. This algorithm even works if the graph contains edges with negative weight.
Determining the values $d_{v,l}$ for one $l$ takes $O(M)$ time, there are $O(N)$ values of $l$ we need, so the time complexity of the Bellman-Ford algorithm is $O(MN)$. After we know the values $d_{v,l}$, we can easily compute the result in $O(N^2)$ time. So the total time complexity of our solution is $O(MN)$.

We can optimize our solution a bit (not asymptotically) as follows. We compute the values $d_{v,l}$ as in the previous solution. If $K < N$, we are done. In the other case for each vertex we know the length of the shortest path having $N - 1$ edges. After $N - 1$ steps we surely are in the second phase – we walk along some edge. Also it suffices to find the minimum of $(K - N - 1).w(xy) + d(x, N - 1)$ through all edges $xy$. 
Turtle breeding

Author(s) of the problem: unknown
Contest-related materials by: Martin Macko, Dávid Pál

We want to determine whether a given number $C$ is a descendant of the numbers $A$ and $B$. To do this, let’s try to describe all the descendants of $A$ and $B$. Note that $XY - X - Y + 2 = XY - X - Y + 1 + 1 = (X - 1)(Y - 1) + 1$. It follows that the immediate descendant of numbers $R + 1$ and $S + 1$ is the number $RS + 1$. In other words, suppose that instead of each turtle with the number $T$ we have a rabbit with the number $T - 1$. Then the number of the immediate descendant of any two rabbits will be the product of their two numbers. Instead of finding out whether there is a turtle with the number $C$, we just have to find out whether there is a rabbit with the number $C - 1$.

Let $D = A - 1$, $E = B - 1$, $F = C - 1$. From the facts mentioned above follows that the descendants of rabbits with numbers $D$ and $E$ are exactly the rabbits with numbers $D^x E^y$, $x, y \in \mathbb{N}_0$, $x + y > 0$. Also we just have to verify whether $F$ can be written as $D^x E^y$ for some adequate $x, y$.

Special cases first. For $F = 0$ the answer is YES iff $DE = 0$. For $F = 1$ the answer is YES iff $D = 1$ or $E = 1$. Now suppose $F \geq 2$. The case $D = 0$ is now equivalent to $D = 1$, similarly $E = 0$ is equivalent to $E = 1$.

From now on let’s suppose that $F \geq 2$, $D, E \geq 1$. We know that $D, E < 10000$ also we can easily write them as products of prime numbers. Now let $p_1, \ldots, p_r$ be the primes dividing $DE$. We will now divide $F$ by the primes $p_i$ until we get either 1 or a number that is not divisible by any of these primes. In the second case the answer is obviously NO. In the first case we now know how to write $F$ as a product of prime numbers. For easier notation, let:

- $D = p_1^{a_1} \cdot p_2^{a_2} \cdot \ldots \cdot p_r^{a_r}$
- $E = p_1^{b_1} \cdot p_2^{b_2} \cdot \ldots \cdot p_r^{b_r}$
- $F = p_1^{c_1} \cdot p_2^{c_2} \cdot \ldots \cdot p_r^{c_r}$

where $a_i, b_i, c_i \geq 0$. The equation $D^x E^y = F$ now becomes a system of equations, one for each prime number. Concretely, for the prime $p_i$ the equation $(p_i^{a_i})^x \cdot (p_i^{b_i})^y = p_i^{c_i}$ must hold. We may rewrite this equation as $a_i x + b_i y = c_i$. Thus we get a system of linear equations.

If there are two linear equations such that the second one is not a multiple of the first one, there is at most one pair $[x, y]$ of real numbers satisfying these two equations. The answer is YES iff these numbers are non-negative integers and satisfy all the other equations.

What remains is the case where all the equations are equivalent. Let’s take any of them, let it be $ax + by = c$. This is a Diophantine equation. It can have many integer solutions, however, we are only interested in finding one of them. The numbers $a, b$ are non-negative, also the bigger is $x$, the smaller is $y$. Let’s find the smallest non-negative $x$ for which there is some $y$ such that the pair $[x, y]$ solves the equation. If the $y$ is also non-negative, the answer is YES, otherwise (including the case where no such $x$ exists)
then the answer is NO.

It remains to show how to find the smallest non-negative $x$ solving (with some $y$) the equation $ax + by = c$. Let’s take the equation $ax + by = c$ modulo $b$, we get $ax \equiv c \pmod{b}$. Clearly if this equation has a solution, it has a solution in the set $\{0, 1, \ldots, b - 1\}$. Also it suffices to try all the numbers in this set and take the first of them that solves the equation.

However, there is a faster way how to solve the equation $ax \equiv c \pmod{b}$. One way is to modify Euclidean’s algorithm to find the solution in $O(\log b)$ time. The other (and in this case even faster) way looks as follows:

Let $d = \gcd(a, b)$. Clearly the solution $ax + by = c$ has no solution if $d$ doesn’t divide $c$. In the other case, let $a' = a/d$, $b' = b/d$. Now we may compute the value $(a')^{-1}$ – the only number from the set $\{0, 1, \ldots, b' - 1\}$ such that $(a')^{-1}a' \equiv 1 \pmod{b}$. We compute $(a')^{-1}$ only once, we may use either exhaustive search or the modified Euclidean’s algorithm to compute it.

Now for each input number $C$ if this case occurs ($C - 1$ is divisible only by those primes that divide $A - 1$ or $B - 1$, all the equations we get by comparing the exponents of the primes are equivalent to $ax + by = c$ for some $c$ and $d|c$), the solution of the equation $ax \equiv c \pmod{b}$ is $(a')^{-1}(c/d)$.

There were many other approaches on how to solve this task, most of them should have performed quite well, because the bottleneck in this problem was dividing $F$ by the primes – this alone takes $O(\log F)$ time.