POLISH OLYMPIAD IN INFORMATICS

The 7th Baltic Olympiad in Informatics
BOI 2001
Sopot, Poland

Tasks and Solutions

Edited by Marcin Kubica

WARSAW, 2001

Authors:

Marcin Kubica
Ville Leppänen
Adam Malinowski
Oleg Mürk
Krzysztof Onak
Martins Opmanis
Mindaugas Plukas
Wolfgang Pohl
Ahto Truu
Piotr Sankowski
Marcin Sawicki
Marcin Stefaniak
Tomasz Waleń
Paweł Wolff

Proofreader: Weronika Walat

Volume editor: Marcin Kubica

Typesetting:

Krzysztof Onak
Tomasz Waleń

Sponsored by Polish Ministry of National Education.

ISBN 83–906301–8–4

Contents

Preface

Baltic Olympiad in Informatics (BOI) gathers the best teen-age programmers from countries surrounding the Baltic Sea. The 7-th BOI was held in Sopot, Poland, June 16–21, 2001. Eight countries took part in the competition: Denmark, Estonia, Finland, Germany, Latvia, Lithuania, Poland and Sweden. All of the countries, except Denmark and Poland, were represented by 6 contestants. Denmark was represented by one and Poland by 12 contestants. BOI'2001 was organized by Polish Olympiad in Informatics together with Prokom Software S.A., Combidata Poland Sp. z.o.o. and the city of Sopot. More information about this competition, can be found at `www.ii.uni.wroc.pl/boi/`.

The contest consisted of three sessions: a trial session and two competition sessions. During the trial session contestants had an occasion to become familiar with the software environment and to solve a warming up task 'Excursion'. During each of the competition sessions they had to solve three tasks within five hours of time: 'Box of Mirrors', 'Crack the Code' and 'Postman' during the first session and 'Knights', 'Mars Maps' and 'Teleports' during the second one. Four contestants were awarded gold medals, eight contestants were awarded silver medals and twelve contestants were awarded bronze medals. The results are as follows:

- Gold medalists:

Martin Pettai	Estonia	Paweł Parys	Poland
Michał Adamaszek	Poland	Krzysztof Kluczek	Poland

- Silver medalists:

Daniel Jasper	Germany	Ivars Atteka	Latvia
Tomasz Malesiński	Poland	Uldis Barbans	Latvia
Karol Cwalina	Poland	Piotr Stanczyk	Poland
Arkadiusz Pawlik	Poland	Tobias Polley	Germany

- Bronze medalists:

Teemu Murtola	Finland	Hendrik Nigul	Estonia
Michael Siepmann	Germany	Marcin Michalski	Poland
Marek Zylak	Poland	Girts Folkmanis	Latvia
Daniel Andersson	Sweden	Benjamin Dittes	Germany
Viktor Medvedev	Lithuania	Kristo Tammeoja	Estonia
Bjarke Roune	Denmark	Fredrik Jansson	Finland

This booklet presents tasks from BOI'2001 together with the discussion of their solutions. Some more materials, including test data used during the evaluation of solutions, can be found at `www.ii.uni.wroc.pl/boi/`. It was prepared for the contestants of various programming contests to help them in their exercises.

Marcin Kubica

Task by
Mindaugas Plukas

Solution description by
Marcin Stefaniak

Excursion

A group of travelers has an opportunity to visit several cities. Each traveler states two wishes on what city he/she does want or does not want to visit. One wish expresses a will to visit or not to visit exactly one city. It is allowed that both wishes of one traveler are the same or that they are opposite—i.e. I want to visit city A, and I do not want to visit city A.

Task

Your task is to write a program that:

- *reads the travelers' wishes from the input file* `exc.in`,
- *determines whether it is possible to form such a list of cities to be visited (the list can be empty), that at least one wish of every traveler is satisfied,*
- *writes the list of cities to be visited to the output file* `exc.out`.

If there are several possible solutions, your program should output anyone of them.

Input

The first line of the input file `exc.in` *contains two positive integers* n *and* m *(*$1 \leq n \leq 20\,000$*,* $1 \leq m \leq 8\,000$*);* n *is the number of travelers, and* m *is the number of cities. The travelers are numbered form 1 to* n*, and the cities are numbered from 1 to* m*. Each of the following* n *lines contains two nonzero integers, separated by single space. The* $i+1$*-th line contains numbers* w_i *and* w_i' *representing wishes of the* i*-th traveler,* $-m \leq w_i, w_i' \leq m$*,* $w_i, w_i' \neq 0$*. A positive number means that the traveler wishes to visit that city, and a negative number means that the traveler does not wish to visit the city specified by the absolute value of the number.*

Output

Your program should write one nonnegative integer l*, the number of cities to be visited in the first line of the output file* `exc.out`*. In the second line of the file there should be written exactly* l *positive integers in the ascending order, representing cities to be visited.*

In case it is not possible to form such a list of cities, your program should write in the first and only line the word `NO`*.*

Example

For the input file `exc.in`*:*

```
3 4
1 -2
2 4
3 1
```

the correct result is the output file `exc.out`*:*

```
4
1 2 3 4
```

Solution

A bit of logic

In the task 'Excursion', the goal is to find such an excursion route—a subset of a given set of cities, that satisfies every traveler. There is something special about the travelers' wishes, namely, every wish is an alternative of exactly two simple conditions, like 'to visit the city' or 'not to visit the city'. We can easily express these wishes in the language of propositional logic.

For example, the input file:

```
3 4
1 -2
2 4
3 1
```

means that:

- there are 3 travelers and 4 cities,
- the first traveler wants to visit the 1-st city or not to visit the 2-nd city,
- the second traveler wants to visit the 2-nd city or the 4-th city,
- the third traveler wants to visit the 3-rd city or the 1-st city.

Let us denote by a_i the sentence 'the excursion is to visit the city number i'. These are called *variables*, since their logical values (whether they are true or false) are not fixed.

In our example, there are four variables a_1, a_2, a_3, a_4. Now, the travelers' conditions can be expressed by the following formulae:

$$\begin{cases} a_1 \vee \neg a_2 \\ a_2 \vee a_4 \\ a_3 \vee a_1 \end{cases}$$

Since all the travelers are going to the same excursion, all of these conditions have to be satisfied together:

$$(a_1 \vee \neg a_2) \wedge (a_2 \vee a_4) \wedge (a_3 \vee a_1)$$

This formula is equivalent to the excursion problem from the example file. To solve the problem one should *satisfy* this formula (assign logical values—true, false—to the variables, so that the formula is true), or show that there is no such assignment (implying that the excursion is impossible).

In general, the input file can be rearranged and transformed into a logical formula, whose satisfaction is equivalent to solving the excursion problem. The problem of satisfying a given logical formula is well known to be NP-complete[1]and hardly ever is an NP-complete problem believed to be solvable in a polynomial time. Fortunately, the formula obtained in this task is of a very special form:

$$(l_1^1 \vee l_1^2) \wedge (l_2^1 \vee l_2^2) \wedge \ldots \wedge (l_m^1 \vee l_m^2)$$

where l_j^1, l_j^2 are called *literals* and each of them stands for some variable a_i or negated variable $\neg a_i$. The literals l_j^1 and l_j^2 correspond to the conditions of the j-th traveler. The formula is a conjunction of alternatives of exactly two literals. Therefore it is said to be in the *second conjunctive normal form*, (i.e. 2-CNF).

We are particularly lucky that the tourists stated at most two conditions. If they were allowed to state three of them, the alternatives in the formula would consist of three literals, hence the formula would not be in 2-CNF, but in 3-CNF form, and it has been proven that satisfying 3-CNF formula is another NP-complete problem. That proof and more information on NP-completeness can be found in [1]. However, the excursion problem is equivalent to satisfying a formula of 2-CNF form, and we shall show that this problem can be solved in a polynomial, linear time.

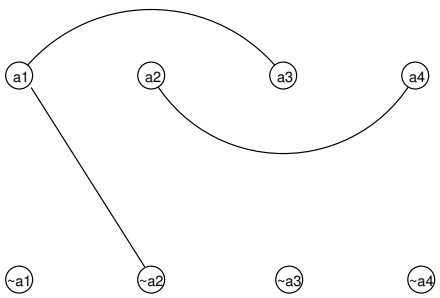
Excursion and graphs

We shall consider an undirected graph $G = (V, E)$ with vertices corresponding to all possible literals:

$$V = \{a_1, \neg a_1, a_2, \neg a_2, \ldots, a_n, \neg a_n\}$$

and edges connecting pairs of literals which appear in alternatives of the formula:

$$E = \{(l_i^1, l_i^2) : i = 1, 2, \ldots, m\}$$



The graph G for the example input.

[1]We consider here propositional logic (without quantifiers and predicates, which would make things even worse).

Our goal is to select a subset $W \subset V$ containing vertices corresponding to these literals which are true when the formula is satisfied. For that, the following holds and must hold:

- (♠) either $a_i \in W$, or $\neg a_i \in W$ (exclusively), for $i = 1, 2, \ldots, n$,
- (♣) for any edge $(u, v) \in E$, $u \in W$ or $v \in W$ (i.e. $u \vee v$),

We say vertices a_i, and $\neg a_i$ are opposite, because one and only one of them must belong to the set W.
Suppose $(u, v) \in E$, or $u \vee v$. By laws of logic we obtain useful implications:

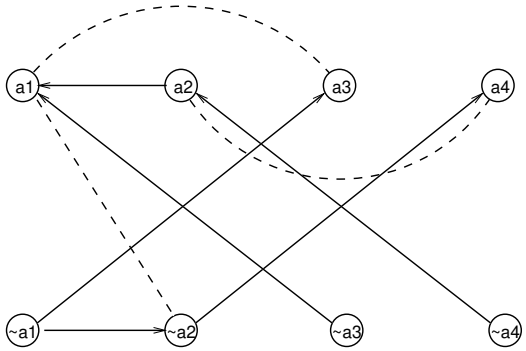
$$u \vee v \iff \neg u \Rightarrow v \iff \neg v \Rightarrow u$$

We construct a directed graph $G' = (V, E')$, whose edges represent these implications:

$$E' = \{(\neg u, v) : (u, v) \in E \vee (v, u) \in E\}$$

We shall call this the *inference graph*, because it can be used to find which vertices one has to choose to the set W, provided that some given vertex has already been chosen.

The inference graph has $2n$ vertices and $2m$ edges, where n is the number of cities and m denotes the number of travelers (this differs from the task statement, but we shall stick here to the usual convention).



The example of inference graph G'. Edges of the original graph are dashed.

In our example, we can now clearly see, that if one adds the vertex $\neg a_4$ to the set W, one must also include vertices a_2, and consequently a_1, because these vertices can be reached from $\neg a_4$ through the edges of the inference graph.

Every $W \subset V$ which is a correct solution of the problem must adhere to the inference graph, in such a way that:

$$(\diamondsuit)(w \in W \wedge (w, v) \in E') \Rightarrow v \in W.$$

At the same time, any W that satisfies ($\diamondsuit$) and (♠), forms a correct solution to the excursion problem, since ($\diamondsuit$) $\iff$ (♣), which is an obvious consequence of the way in which the graph G' was constructed. Therefore it is enough to search for a $W \subset V$ adhering to the inference graph, and not contradicting the exclusiveness condition (♠).

Let us denote

$$\mathit{Induced}(w) = \{v : w \rightarrow^* v\},$$

where $\rightarrow^*$ means that there is a path from w to v in the graph G'. The following property holds, since it is equivalent to ($\diamondsuit$):

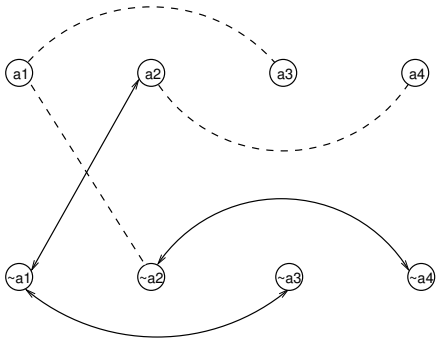
$$w \in W \Rightarrow \mathit{Induced}(w) \subseteq W.$$

If a set $\mathit{Induced}(w)$ contains two opposite vertices (which is not allowed in a solution), then we say that the vertex w is *troublesome*.

Another graph which is useful in solving this problem is the conflict graph. For each $(u, v) \in E \iff u \vee v$, by laws of logic

$$u \vee v \iff \neg(\neg u \wedge \neg v),$$

hence the vertices $\neg u$ and $\neg v$ are in conflict, and cannot be both present in the solution set W.



The conflict graph for the example input. The edges of the graph G are dashed.

Simple algorithm

The following algorithm solves the problem:

```
1:   W = ∅;
2:   while W < n do
3:   begin
4:     let c be a city, such that c, ¬c ∉ W;
5:     if both c and ¬c are troublesome then
6:       solution does not exist, STOP
7:     else
8:       begin
9:         let v be a non-troublesome vertex c or ¬c;
10:        W := W ∪ Induced(v);
11:      end
12:  end
13:
```

The algorithm chooses a vertex accepting or rejecting a city, and then expands the set W by induced vertices. Notice that cities that are not mentioned in any way in the set of induced vertices do not conflict with the cities that are. This is the very key to the solution, as it allows to solve the problem without searching recursively the whole space of possible solutions.

Independence property

Let $U = Induced(v)$ and let M be such a set of vertices that neither they nor the vertices opposite to them are in the set U. If the vertex v is not a troublesome one, then there is no edge in the conflict graph between U and M.

This fact might seem quite hard to see at first glance, but it is easy to show. Should a conflict happen between $m \in M$ and $u \in U$, then $u \to \neg m$, so $\neg m \in Induced(u) \subseteq Induced(v) = U$. But $\neg m \notin U$ by the definition of M.

As the result of the independence property, the algorithm, in each turn of the loop, searches for vertices in a similar way, as it does in the beginning. Furthermore, *any* algorithm, which does not accept troublesome vertices and would not accept a pair of opposite or conflicting vertices, eventually finds a correct solution.

Strongly connected components

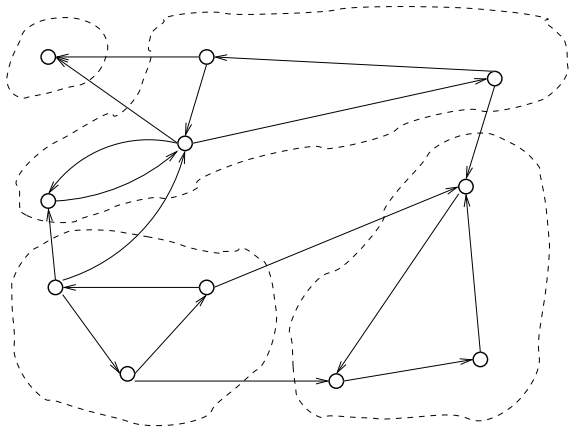
The algorithm presented in section 'Simple algorithm' is not the fastest one. The time complexity of checking if a vertex is troublesome may take time proportional to the size of the graph, so the overall time complexity is $O(n * (n + m))$. We shall show a more efficient algorithm, which solves the problem by calculating strongly connected components of the inference graph.

Some of the readers may be familiar with the concept of strongly connected components in a directed graph. We say that two vertices u, v belong to the same strongly connected component, when there is a path from u to v as well as from v to u. Certainly, being in the same strongly connected component is an equivalence relation, hence it divides the set of vertices into disjoint groups. These groups are called, obviously, strongly connected components.

Strongly connected components of an inference graph have a useful property: for any component $C \subseteq V$, either $C \subseteq W$, or $C \cap W = \emptyset$. In other words, we cannot choose a component partially; we can only take

all vertices from that component, or none of them. Therefore we shall consider the graph of components $G_c = (V_c, E_c)$, whose vertices are strongly connected components of the graph G', and edges are inherited from the graph in a natural way.

There is an algorithm calculating strongly connected components in a graph in a linear time in term of the size of the graph, $O(n + m)$. The algorithm is quite simple, but tricky. One can find detailed explanations in e.g. [1].



An example of strongly connected components in a graph.

Let us suppose for a while that there is a cycle in the graph of components, then we could easily see that all vertices in that cycle should belong to the same component—an obvious contradiction. Hence, the graph of components is a directed acyclic graph (DAG).

We shall sort topologically the DAG of components, so that we could browse its vertices in a not-descending order, according to the graph of components. This can also be done in a linear time.

We say that we accept a component, when (while performing the algorithm) the component is chosen and included to the set of true literals W. We also say that we reject a component, if we decide not to choose the component anyway (perhaps due to some contradictions).

Note that rejecting a component entails rejecting all its ascendants in DAG G_c, and similarly accepting a component leads to accepting all its descendants. In fact, the graph G_c is the graph of inductive reasoning, whereas the reversed graph G_c^T can be viewed as the graph of deductive reasoning; these metaphors might explain the name 'inference graph'.

Algorithm

The following are the main steps of the algorithm solving the problem:

1. read the input file and generate the inference graph G',
2. find strongly connected components and the components graph G_c,
3. if there are two opposite vertices in a component, then reject this component and all its ascendants,
4. sort topologically the components, and process them in ascending order:
   (a) if the current component has not been rejected, then accept it ,
   (b) for each vertex in the accepted component reject the component containing the opposite vertex (and consequently all its ascendants),
5. if exactly n vertices have been accepted, then they form a correct solution; otherwise solution does not exist.

Note that because of the topological ordering of components, each time we accept a component all of its descendants have already been accepted (had one been rejected before, then the current component would have been rejected).

We shall show that this algorithm really solves the problem. It is easy to see that if half of the vertices are accepted, then this set of vertices is a correct solution. Indeed, it does not contain any conflicts, because of rejecting components in steps 3 and 4.b, and is consistent with the inference graph, as we have pointed out. What we have to show then, is that if there exists a solution, then the algorithm will find one.

Notice that when a vertex is accepted the opposite one is rejected. As it was explained in section 'Independence property', accepting a vertex is harmless provided that it is not troublesome. But all the troublesome

vertices are rejected in the step 3, and since we are processing components in ascending order, we are avoiding conflicts.

Therefore, from the independence property comes the correctness of the solution algorithm.

Other solutions

Naive solution

One could solve this problem by crude backtracking. In this algorithm, we consider a city i, which is yet neither accepted nor rejected. We try to accept it and using the inference graph G' we can find its descendant vertices, which are also accepted. Meanwhile, we check if a contradiction has occurred. If so, we try the second option, namely rejecting the city. In case of a successful try, we consider another city, and so on; if both decisions failed, we backtrack.

The time complexity of this algorithm is exponential, so it is a very slow solution.

Tests

The tests have been generated randomly for various number of cities and travelers, increasing gradually in order to distinguish different solutions. The tests are grouped in order to detect programs answering always 'NO'. (A program could score points only for solving all tests in a group). The test group number 8 was especially designed to discern non-linear solutions $O((n+m)\cdot m)$ from the linear ones.

No	No of travelers	No of cities	Remarks
1	27	10	
2	107	33	
3	1010	100	
4a	48	100	answer NO
4b	106	100	
5a	257	333	answer NO
5b	3022	400	
6a	829	1000	
6b	1006	1000	answer NO
7	20000	8000	
8a	15998	8000	
8b	15998	8000	
8c	16000	8000	

Task by
Martins Opmanis

Solution description by
Piotr Sankowski

Box of Mirrors

Mathematician Andris likes different puzzles and one of his favorites is a covered box of mirrors. If we look at a horizontal cross-section of such a box, we can see that its bottom part contains $n \times m$ square cells (n rows, and m columns). In each cell there can be placed a mirror which is oriented diagonally from lower left corner to upper right corner. Both sides of the mirror reflect light. At the box edges located at both ends of cell rows and columns there are gaps through which you can light a beam into the box or a beam can come out of the box. Through each gap you can light a beam in only one direction—perpendicular to the edge containing the gap. Therefore, a beam reflected from a mirror changes its direction by 90 deg. When the beam goes through empty cells, its direction doesn't change. Gaps are numbered consecutively from 1 to $2 \cdot (n + m)$, around the box, counter-clockwise, starting from the gap on the left side of the upper left cell and going downwards. Since the arrangement of mirrors in the box is not visible, the only way to determine it is by lighting beams into some gaps and watching where the light comes out.

Task

Write program that:

- *reads the size of the box and gaps describing beams coming in and out of the box from the input file* `box.in`,
- *determines in which cells there are mirrors and which cells are empty,*
- *writes the result to the output file* `box.out`.

If there are several possible solutions, your program should output anyone of them.

Input

First line of input file `box.in` *contains two positive integers: n (the number of cell rows, $1 \leq n \leq 100$) and m (the number of cell columns, $1 \leq m \leq 100$) separated by a single space. Each of the following $2 \cdot (n + m)$ lines contains one positive integer. The number in the $i + 1$-th line denotes the number of the gap from which the light comes out if it is lightened into gap number i.*

Output

Your program should write to the output file `box.out` *n lines, each of them containing m integers separated by single spaces. The j-th number in the i-th line should be 1, if there is a mirror in the cell in the i-th row and j-th column of the box, or it should be 0 if the cell is empty.*

Example

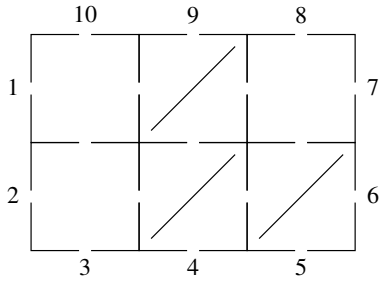
For the input file `box.in`*:*

```
2 3
9
7
10
8
6
5
2
4
1
3
```

the correct result is the output file `box.out`*:*

```
0 1 0
0 1 1
```

14 *Box of Mirrors*



Solution

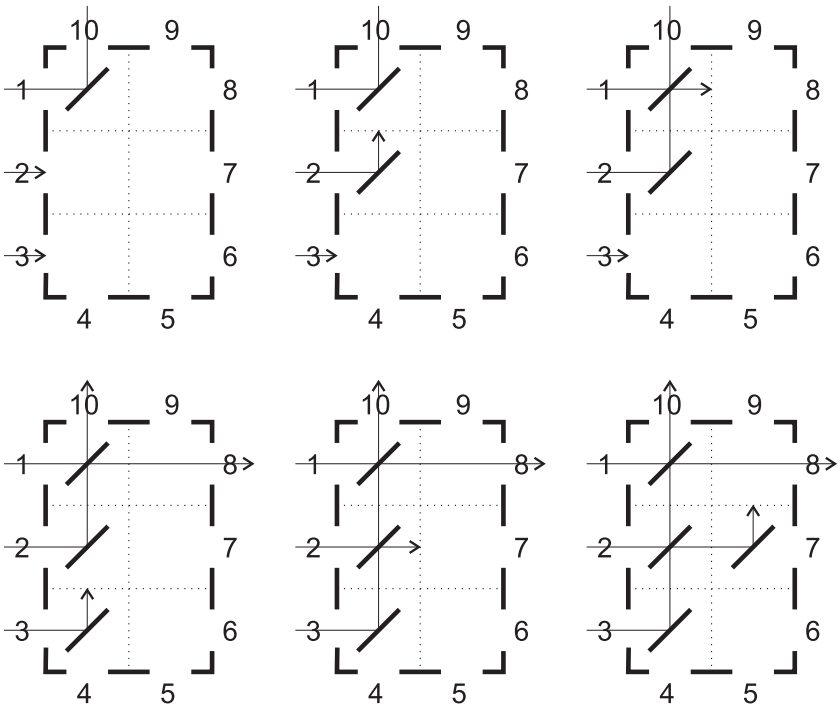
The solution of this problem is very easy, but as is often the case, it's not so easy to find. The problem can be solved by a greedy algorithm. A greedy algorithm might also be called a 'single-minded' algorithm, an algorithm that takes care of only one thing. It performs a single procedure in a recipe over and over again until such a process cannot be continued anymore. In every step it enlarges the solution by the best element or just by an element that fulfills some property. In most cases such an algorithm doesn't produce an optimal solution, but in some it does. Many such cases are described by a matroid theory. For more information on this topic see [1]. The matroid theory provides a strict way of proving the correctness of a greedy algorithm. One only has to check if every step satisfies some given conditions.

The correctness of the algorithm for this problem will be proved in a similar way. We will show that every step of the algorithm will not hinder the following steps. Every step will consist of finding a track for one beam, and it will be done in such a way that it won't make it impossible to find tracks for other beams. Our algorithm will successively track beams lighted into gaps with numbers $1, 2, ..., n + m$. At the beginning the box will not contain any mirrors, they will be placed where needed while tracking the beams. They will also be placed in such a way that the beam goes as much as possible upwards, in order to not 'disturb' light beams going underneath. Let us consider the following algorithm to track beams:

1. Let x and y be the current column and row of the tracked beam. Let x' and y' be the column and row of the gap through which the beam should leave.
2. If $x = x'$ and $y = y'$ then stop.
3. If the beam goes vertically go to step 5.
4. If there is no mirror above the current position and $x \neq x'$ or $y = y'$, then go one cell to the right: $x := x + 1$. Otherwise place a mirror in cell (x, y) and go one cell upwards: $y := y - 1$. Go to step 2.
5. If there is a mirror in the current cell, then go right: $x := x + 1$. Otherwise go up: $y := y - 1$. Go to step 2.

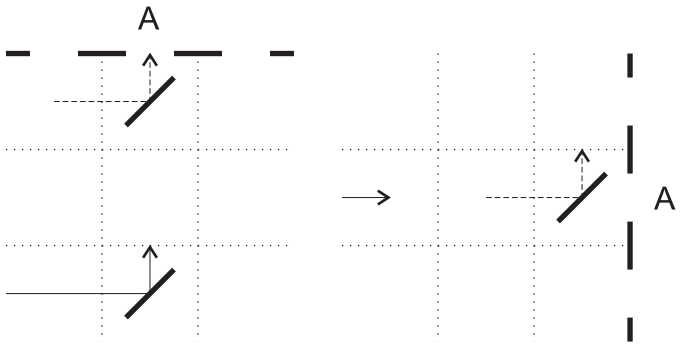
In step 4 of the algorithm we go right if we are in the row in which there is the hole through which the beam should leave, or we go up and place a mirror in the current cell if we are in the column in which there is the destination hole or there is some mirror above, so we go as much as possible upwards.

The following pictures show how the first three beams are tracked for a box with 2 columns and 3 rows and light beams $1 \to 10, 2 \to 8, 3 \to 9, 4 \to 6$ and $5 \to 7$.



Example of how the algorithm works for a a box with 2 columns, 3 rows and light beams $1 \to 10, 2 \to 8, 3 \to 9, 4 \to 6$ and $5 \to 7$.

Note that, when a mirror is placed the light beam is reflected on the left side of the mirror (step 4 of the algorithm). Other steps contain no requirements, in other words nothing may go wrong with them, they only track the beam. Consider the condition in point 4. There are only two possible situations when the algorithm might produce incorrect solutions, both are shown in the following picture. We track the beam and in a certain step it is in the row or column of the hole through which it should leave, but the tracked beam cannot be lighted out through the proper hole (marked with A) because there is a mirror in its way.



Incorrect situations. The beam should leave through gap A, but there is a mirror in its way.

Such situations cannot happen in our algorithm, because there must be already some other light beam reflected form the left side of the mirror, and in the case shown on the left it has already left through this hole, but in our algorithm beams leave only through correct holes, and two beams cannot be lighted out through the same hole. In the case on the right, there already is some other beam going in this row, so it is impossible too.

Contestants' Solutions

The results scored by the solutions of this problem were mostly either near maximum or not more than 20% of the points. Contestants usually either came to the described solution and scored all points or they tried to implement some backtracking algorithm which worked in an exponential time and passed mostly two smallest tests.

Tests

In this problem there are no special cases, and so there are no special test cases. All tests for this problem are random—in a box of some size there were placed some mirrors at random positions. The main goal of the

tests was to distinguish between optimal solutions and exponential ones, and to check the correctness of the solutions.

Task by
Oleg Mürk and Ahto Truu

Solution description by
Paweł Wolff

Crack the Code

Cryptography is the science and technology of coding messages so that only the intended recipient can read them. Cryptanalysis, on the other hand, is the science of breaking the codes. For this problem, assume you're a cryptanalyst hired to break open a series of encrypted messages captured in a police raid on the headquarters of the local mafia.

Your colleagues have already reverse engineered the encryption program (see `crack.pas` *or* `crack.c` *for the code), and the only thing left to do is to reverse the algorithm and guess the key used to encrypt the files.*

Along with the encrypted files, there are also some plaintext files that are believed to originate from the same source as the encrypted files and thus have a similar structure in terms of language, word usage, etc.

Task

Your task is to decrypt the messages given and to save them in the specified files. You do not have to provide any program—just the decrypted messages.

Input

You are given several data sets. They consist of files `cra`*n*`.*`*, *where n is the number identifying the data set. Each data set consists of the files:*

- *cra*`*.in`*, encrypted message,*
- *cra*`*.txt`*, plaintext files of the same origin, as the encrypted message.*

Output

For each encrypted message cra`*.in`*, you should save the decrypted message in the file cra*`*.out`*.*

Solution

Encryption program

In this section we are going to explain, how the encryption program `crack.pas` (`crack.c`) works. First we describe a simple kind of cipher, known as the Caesar's code. The process of encrypting the text with such a code is following. Let us fix the encryption key—in the case of this cipher it is a single integer number k. The encrypted text results from changing every single letter in the input text to letter which stands in the English alphabet (cyclic-formed) k positions further. For example, if we have chosen number 4, as k then `'CIW, SJ GSYVWI'` is the encrypted version of `'YES, OF COURSE'`. (Notice, that such characters as spaces and commas are unaffected). If we apply this algorithm with the key $-k$ to the text which has been encrypted in the way described above, it will give us back the original text. Therefore as the result of coding the text `'CIW, SJ GSYVWI'` with the key -4 (which means, that every letter is changed to the letter standing 4 positions before it) we will obtain the primary text `'YES, OF COURSE'`.

The encryption algorithm which was implemented in the program `crack.pas` (`crack.c`) is a bit more complicated. Very briefly we can say that it is a combination of ten various Caesar's codes. More precisely, this cipher can be described as follows. Let us omit for a while all the characters in our text which are not letters. We divide the input text into several pieces, each consisting of ten letters, except for the last one which might be shorter. Now we form these pieces into an array, such that one piece is represented by one row and they are set one under another. The following example illustrates this:

I	T	W	A	S	A	F	I	N	E
S	P	R	I	N	G	M	O	R	N
I	N	G	I	N	T	H	E	F	O
R	E	S	T	A	S	H	E	S	T
A	R	T	E	D	O	U	T		

The split text is: 'IT WAS A FINE SPRING MORNING IN THE FOREST AS HE STARTED OUT.'

In the previous encryption algorithm the encryption key was a single integer number. Here the key is a sequence of ten integers $(k_1, \ldots, k_{10})$. Encrypting a text with such a key consists of applying Caesar's code to each column of the array separately. It means that the first column, considered as a text, is being encrypted by the Caesar's code with key k_1, the second column—with key k_2, etc. For instance, coding the text from the example with the key $(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)$ gives the following array:

J	V	Z	E	X	G	M	Q	W	O
T	R	U	M	S	M	T	W	A	X
J	P	J	M	S	Z	O	M	O	Y
S	G	V	X	F	Y	O	M	B	D
B	T	W	I	I	U	B	B		

The encrypted version is: 'JV ZEX G MQWO TRUMSM TWAXJPJ MS ZOM OYSGVX FY OM BDBTWII UBB.'

What remains now is to join all the rows of the modified array and put all the forgotten characters into appropriate places. What we obtain is an encrypted version of the input text. (Note that the method described above can be explained in a different way: the 1st, 11th, 21st, ... letter occurring in the text is encrypted by the Caesar's code with key k_1; the 2nd, 12th, 22nd, ... letter occurring in the text—with the key k_2; etc. and at last 10th, 20th, 30th, ... letter occurring in the text—with the key k_{10}. The characters which are not letters are unaffected—like in the Caesar's code).

Methods of cracking the code

First let us consider methods which enable one to decrypt a message which is known to have been encrypted by Caesar's code. We will see later, that some of these methods can by easily applied to the case of the cipher which is used in our task.

The simplest method is to consider all values from the interval $[0, 25]$ as a value of the key. Considering other values is unnecessary, because using the key k gives the same result as using the key of value equal to the remainder of the division of k by 26—the total number of letters in the English alphabet. This remainder obviously belongs to the interval $[0, 25]$. In case the text is written in some natural language and its length is more than a few characters, it is quite certain that for exactly one value of the key we obtain some text which does make sense. Because of the abovementioned small number of various keys, it is possible to verify 'manually' which key is the right one. However this method cannot be used effectively in the case of messages encrypted by the second kind of cipher. In that situation we should guess not one, but ten numbers. It doesn't seem that it is possible to guess every key separately using 'manual' verification, because guessing the first number of a ten-number key deciphers only every tenth letter of the text. The decoded part of the text surrounded by 90% of the coded one couldn't be easily seen to be deciphered correctly (in contrast with the situation, where a one-number key deciphers 100% of letters in the text). Verification is easy only if we try to decipher the whole message by choosing all ten elements of the key. Nevertheless there are 26^{10} combinations to check, therefore manual verification of all these cases is impossible.

Let us then show other methods. We make an assumption that we have at our disposal some plaintext as well as a message which is believed to be encrypted by Caesar's code and to have the same origin as the first one. We wish our methods to find the key (at this time automatically) which deciphers the given message. These methods are based on the fact that in a natural language the frequency of occurrence of each letter of the alphabet is different. For instance in English the frequencies are as follows:

Letter(s)	Frequency of occurrence
E	11%
T	9%
A	8%
I, O, S	7%
...	...
J, K, Q, X, Z	less than 1%

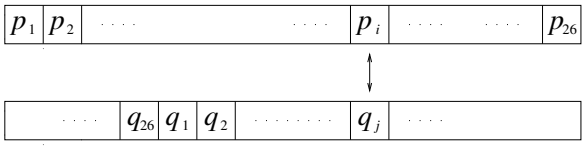
Hence, if the plaintext is in English then E should be the most common. So it follows that if J is the most common letter in the encrypted message, we may suppose that the key 5 had been used (because J stands 5 positions further than E). In this case the key $-5 \equiv 21 (\mathrm{mod}\ 26)$ should be used for decrypting the message. This method does not always give the correct solution. Let us consider a situation, where some two letters occur with similar frequencies (e.g. E and T in English). If the encrypted message is not too long it may happen that T appears more often than E. A message like that would be deciphered incorrectly. For example, if the

encrypted message is `'ZNOY OY G YZGZKSKTZ'` then `Z` appears the most often and it would be decoded as `E`, giving the message `'ESTD TD L DELEPXPYE'`, which certainly doesn't make sense. In fact `Z` should be decoded as `T` which would give the answer `'THIS IS A STATEMENT'`.

Let us try to find a better method of deciding whether a given key is the right one. It will be convenient to introduce the following definitions:

- p_i — frequency of occurrence of i-th letter of alphabet in plaintext (i.e. p_1 is related to letter `A`),
- q_i — frequency of occurrence of i-th letter of alphabet in encrypted message,
- P — sequence of numbers $(p_1, p_2, \ldots, p_{26})$,
- Q — sequence of numbers $(q_1, q_2, \ldots, q_{26})$,
- $Q^{(k)}$ — sequence of numbers $(q_1{}^{(k)}, q_2{}^{(k)}, \ldots, q_{26}{}^{(k)})$ obtained as a cyclic shift of the sequence Q k positions to the left (i.e. $Q^{(3)} = (q_4, q_5, \ldots, q_{26}, q_1, q_2, q_3)$).

The previous method was based on the examination carried out for one pair of indexes i, j for which the value p_i is the greatest among $p_1, p_2, \ldots, p_{26}$ and the value q_j is the greatest among $q_1, q_2, \ldots, q_{26}$. The key used for encryption was expected to equal the number k, for which the maximum values in sequences P and $Q^{(k)}$ appear at the same position.



Now the main idea is to choose such a sequence $Q^{(k)}$ among $Q^{(0)}, Q^{(1)}, \ldots, Q^{(25)}$ that it is the 'closest' to the sequence P. In fact the previous method is an example of this kind of method. In that method two sequences were recognized as similar if their maximum values appeared at the same position. What we are going to do now is to work out some better criterion of deciding which sequence $Q^{(k)}$ is the closest to P. In order to do that we will find some function whose arguments are two sequences and its value is a non-negative real number. We also demand that the value of such a function is close to zero if and only if the sequences are close to each other in an intuitive sense. (In other words: we expect that this function would be a measure of similarity of the sequences).

Let us consider the following example of such a function:

$$S(P, Q^{(k)}) = \sum_{i=1}^{26} |q_i{}^{(k)} - p_i|. \tag{1}$$

The function $S(P, Q^{(k)})$ recognizes sequences P and $Q^{(k)}$ as similar if the sum of differences between the corresponding elements of these sequences is small. At first this function may seem to meet our expectations. Unfortunately, the following example shows that such a definition of similarity of sequences has some faults.

For the purpose of simplification we shall consider three sequences which contain five elements.

$$\begin{aligned} P &= \left(\frac{2}{19}, \frac{3}{19}, \frac{4}{19}, \frac{5}{19}, \frac{5}{19}\right) \\ Q' &= \left(\frac{1}{19}, \frac{2}{19}, \frac{3}{19}, \frac{4}{19}, \frac{9}{19}\right) \\ Q'' &= \left(\frac{2}{19}, \frac{3}{19}, \frac{4}{19}, \frac{9}{19}, \frac{1}{19}\right) \end{aligned}$$

(Note that Q'' is just a cyclic shift of Q'.)

$$\begin{aligned} S(P, Q') &= (1+1+1+1+4)/19 = 8/19 \\ S(P, Q'') &= (0+0+0+4+4)/19 = 8/19 \end{aligned}$$

As we can see the function S doesn't distinguish Q' from Q'' although intuitively it is obvious that Q' is closer to P than Q''.

By reason of the above example we should improve our definition of the function S. First we change formula (1) so that it uses the symbol $\alpha_i = \frac{|q'_i - p_i|}{p_i}$, which states the relative difference of q'_i and p_i (relative to p_i):

$$S(P, Q') = \sum_{i=1}^{26} p_i \frac{|q'_i - p_i|}{p_i} = \sum_{i=1}^{26} p_i \alpha_i. \tag{2}$$

Now we can see that S(P, Q') can be interpreted as a weighted mean of α_i with weight factors p_i.

Let us consider the following modification of (2):

$$\widehat{S}(P, Q') = \sum_{i=1}^{26} p_i {\alpha_i}^2 = \sum_{i=1}^{26} \frac{|q'_i - p_i|^2}{p_i}. \tag{3}$$

Here $\widehat{S}$ is just a weighted mean of the squares of α_i. In this formula big relative differences between q'_i and p_i influence the whole sum much more than in the case of formula (2).

Let us recall our latest example to check whether the function $\widehat{S}$ works better than the function S:

Sequences Q' and Q'' are the same as in the previous example.

$$\begin{aligned} \widehat{S}(P, Q') &= (1/2 + 1/3 + 1/4 + 1/5 + 16/5)/19 \approx 4.48/19 \\ \widehat{S}(P, Q'') &= (0 + 0 + 0 + 16/5 + 16/5)/19 = 6.4/19 \end{aligned}$$

As we can see our new function manages this example quite well. However there might appear some problems, when for some i the number p_i was very small, for example less than $1/10n$ (n is a number of letters used to calculate the sequence Q, i.e. in the case of Caesar's code it is a number of all letters in the encrypted text, and in the case of the second kind of cipher it is about 10% of the total number of letters). Then it is quite likely that the value ${q_i}^{(k)}$ (where k is the key which has been used for encrypting) won't be close to p_i, because of a small number of letters (small relative to $1/p_i$) used for calculating ${q_i}^{(k)}$. (It is a general problem — we can't approximate the probability of some rare event if we perform only a small number of experiments in order to do that). Therefore we wish this kind of components of the sum (see formula (3)) not to influence this sum too much. The solution is to put several letters into one group and to treat them as a single symbol. E.g. in English each of the letters `J`, `K`, `Q`, `X`, `Z` appears very rarely, but the frequency of occurrence of `J`, `K`, `Q`, `X` and `Z` together may be big enough. Certainly, it causes some modifications of sequences P and Q. We can imagine that other elements p_{27} and q_{27} would be added to the sequence P and Q respectively and that appropriate elements would be removed from these sequences (e.g. elements related to letters `J`, `K`, `Q`, `X`, `Z`). The number p_{27} would be a sum of p_i which have been removed from P and q_{27} would be an analogous sum of some q_i. (For the implementation details see the source code of the program `cra.pas`).

Now it is easy to apply the idea described above in order to crack the code which is used by the program `crack.pas` (`crack.c`). We consider each column of an array (see subsection 'Encryption program') separately. For the i-th column we calculate the sequence Q and choose such a number k_i that the value of $\widehat{S}(P, Q^{(k_i)})$ is minimal. After all that we should get the key $(k_1, k_2, \ldots, k_{10})$. All it takes now is to code our ciphered message with the key $(-k_1, -k_2, \ldots, -k_{10})$ in order to obtain the deciphered version of the message.

About statistical test χ^2

What we have finally obtained in formula (3) is a formula for so called statistics of the statistical test χ^2. The most common application of this statistical test is to verify whether some random variable (e.g. height of a human body) has a probability distribution similar to some given probability distribution (e.g. Gaussian distribution). In our task the probability distribution of the random variable was represented by the sequence $Q^{(k)}$ and the given probability distribution was represented by P.

Data sets

Five data sets were used to score the contestants' solutions. It should be easy to see that a small number of letters in the encrypted message (or in the plaintext file) may cause some statistical methods to give incorrect results. Therefore test cases with a small number of letters in their files can be more difficult to solve than others.

No test case	letters in `cra*.in`	letters in `cra*.txt`
1	891	460
2	1412	1422
3	7422	1756
4	404	170
5	225	100

Task by
Ville Leppänen

Solution description by
Tomasz Waleń

Postman

A country postman has to deliver post to customers who live in villages as well as along every road connecting the villages.

Your task is to help the postman design a route that goes though every village and every road at least once—the input data are such that this is always possible. However, each route also has a cost. People in the villages wish that the postman visit their village as early as possible. Therefore, each village has made the following deal with the post: If the village i is visited as the k-th **different** *village on the tour and $k \leq w(i)$, the village pays $w(i) - k$ euros to the post. However, if $k > w(i)$, the post agrees to pay $k - w(i)$ euros to the village. Moreover, the post pays the postman one euro for each road on the tour.*

There are n villages, numbered form 1 to n. The post is located in village number one, so the route should start and end in this village. Each village is placed on the crossing of two roads, on the crossing of four roads, or there is a road going through the village (i.e. there are 2, 4, or 8 roads going out of each village). There can be several roads connecting the same villages or a road can be a loop, i.e. connect a village with itself.

Task

Your task is to write a program that:

- *reads the description of the villages and the roads connecting them, from the input file* `pos.in`,
- *designs such a route that goes through each village and road at least once and maximizes the total profit (or minimizes the loss) of the post,*
- *writes the result to the output file* `pos.out`.

If there are several possible solutions, your program should output just one of them.

Input

In the first line of the input file `pos.in`*, there are two integers n and m, separated by a single space; n, $1 \leq n \leq 200$, is the number of villages and m is the number of roads. In each of the following n lines there is one positive integer. The $i+1$-th line contains $w(i)$, $0 \leq w(i) \leq 1\,000$, specification of the fee paid by the village number i. In each of the following m lines there are two positive integers separated by a single space—villages connected by consecutive roads.*

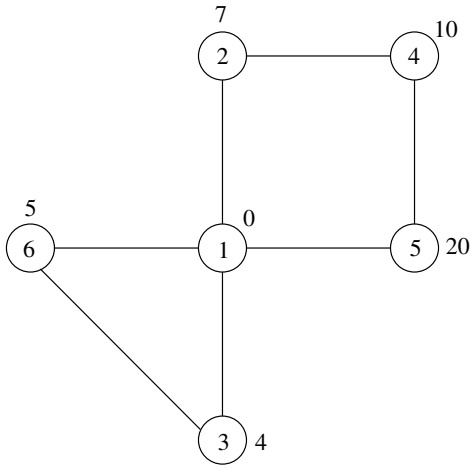
Output

Your program should write one positive integer k, the length of the route, to the first line of the output file `pos.out`*. The following line should contain $k+1$ numbers of consecutive villages on the route $v_1, v_2, \ldots, v_{k+1}$, separated by single spaces, with $v_1 = v_{k+1} = 1$.*

Example

For the input file `pos.in`*:*

```
6 7
0
7
4
10
20
5
2 4
1 5
2 1
4 5
3 6
1 6
1 3
```



the correct result is the output file `pos.out`*:*

```
7
1 5 4 2 1 6 3 1
```

Solution

The solution of this task is a little bit tricky. The main observation is that the total profit/loss does not depend on the route chosen. Suppose that we have computed some postman's route, such that every edge is visited at least once, $v_{i_1}, v_{i_2}, \ldots, v_{i_l}, v_{i_1}$, $l \leq m$, m is the number of edges. Let $k(j)$ be the number of different villages visited before the first visit to village number j. Then, the profit can be described by the following formula.

$$profit = \sum_{j=1}^{n}(w(j) - k(j)) - \sum_{j=1}^{l} 1 = \sum_{i=1}^{n} w(i) + \sum_{i=1}^{n} k(i) - l$$

Since the route should include every village, the sequence $k(j)$ is a permutation of the set $\{1 \ldots n\}$, we can write:

$$profit = \sum_{i=1}^{n} w(i) + \frac{n(n+1)}{2} - l$$

For the given test data, every part of this equation except l is constant, so the total cost depends on the length of the chosen route, and not on the order in which the villages are visited. The value l is always grater or equal to m, so the maximal $profit$ equals $\sum_{i=1}^{n} w(i) + \frac{n(n+1)}{2} - m$. So we have to find a route of length m, which includes every edge exactly once. Luckily for us, for such graphs as given in the problem description, such a route can always be constructed.

Solution of the task will go along the following schema:

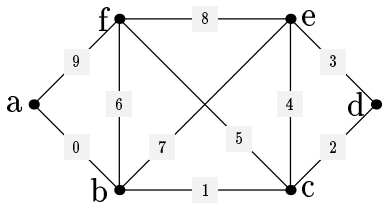
- read the input data,
- compute a route which traverses each edge exactly once (such a route is called an Euler route),
- write the computed route.

Euler route

Euler route is a very old and well known problem of the Graph Theory, it was posed in the XVIII century by a famous mathematician L. Euler.

Definition

An *Euler Route* of a connected graph $G = (V, E)$ is a cycle that traverses each edge of G exactly once. Graph G does not contain a vertex of degree 0.



Img. 1. Example graph with marked Euler route: (a,b,c,d,e,c,f,b,e,f,a).

Theorem

The *Euler Route* can be constructed if and only if the following conditions are satisfied:

- the degree of each vertex is even,
- it is possible to construct a path between every pair of vertices.

If a graph satisfies these conditions, it is called an Euler graph.

Proof

It is clear that if one of the above conditions is not satisfied, we cannot construct an Euler route:

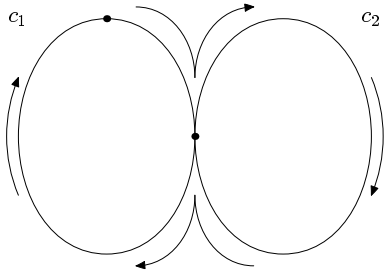
- If it is not possible to construct a path between some pair of vertices, it is also impossible to construct a cycle which traverses every edge (the graph is divided into more than one distinct component).
- If there is a $v \in V$ such that $deg(v)$ is odd, it is also impossible to construct an Euler cycle. In any cycle the number of edges entering a vertex is always equal to the number of outgoing edges, so the degree of any vertex should be even.

The aforementioned conditions are sufficient—the following algorithm proves it.

Algorithm

The algorithm is based on two facts:

- for a given cycle in G, if we remove it from G, it is still possible to construct an Euler cycle in the remaining graph (or graphs). This comes from the fact that removing a cycle from G decrements the degrees of some vertices by even numbers, so all the necessary conditions are still satisfied,
- for given two distinct cycles c_1, c_2 with at least one common vertex, it is possible to construct a new cycle c containing all the edges from c_1, c_2. This can be done by inserting the cycle c_2 into the cycle c_1 at the common vertex.



Img. 2. Merging two cycles.

This gives us the following pseudo-code:

- find any cycle in the graph (for example, using DFS),
- remove from the graph the edges belonging to this cycle,
- compute an Euler route for the remaining graph (or subgraphs if removing the edges divides the graph into more than one component),
- merge the computed cycles into one cycle and return it.

More careful implementation gives us the following algorithm:

```
1: procedure Euler-Route(start);
2: begin
3:   for v ∈ adj(v) do
4:   if not marked(start,v) then begin
5:       mark_edge(start,v);
6:       mark_edge(v,start);
7:       Euler-Route(v);
8:       Result.Push(start,v);
9:   end
10: end
```

Where *Result* is a stack containing the edges of the Euler route. Running time of this algorithm is $O(m+n)$, because every edge is traversed only once; we need $O(m)$ additional memory for marking edges.

Tests

Ten tests were used during the evaluation of contestants' solutions. They are briefly described below.

no	n	Remarks
1	7	simple random test
2	10	small random test
3	20	'ladder' like test with some additional random edges
4	40	random test
5	100	loop
6	100	random test
7	120	'flower' like test (4 loops connected in 1 vertex)
8	150	large random test
9	200	'ladder' like test
10	200	large random test

Task by
Adam Malinowski

Solution description by
Marcin Sawicki

Knights

We are given a chess-board of the size $n \times n$, from which some fields have been removed. The task is to determine the maximum number of knights that can be placed on the remaining fields of the board in such a way that no two knights check each other.

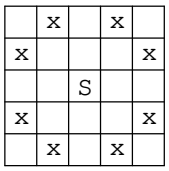


Fig. 1: A knight placed on the field S checks the fields marked with x.

Task

Write a program, that:

- *reads the description of a chess-board with some fields removed, from the input file* `kni.in`,
- *determines the maximum number of knights that can be placed on the chess-board in such a way that no two knights check each other,*
- *writes the result to the output file* `kni.out`.

Input

The first line of the input file `kni.in` *contains two integers n and m, separated by a single space, $1 \leq n \leq 200$, $0 \leq m < n^2$; n is the chess-board size and m is the number of removed fields.*

Each of the following m lines contains two integers: x and y, separated by a single space, $1 \leq x, y \leq n$—these are the coordinates of the removed fields. The coordinates of the upper left corner of the board are (1,1), and of the bottom right corner are (n, n). The list of the removed fields does not contain any repetitions.

Output

The output file `kni.out` *should contain one integer (in the first and only line of the file). It should be the maximum number of knights that can be placed on a given chess-board without checking each other.*

Example

For the input file `kni.in`*:*

```
3 2
1 1
3 3
```

the correct result is the output file `kni.out`*:*

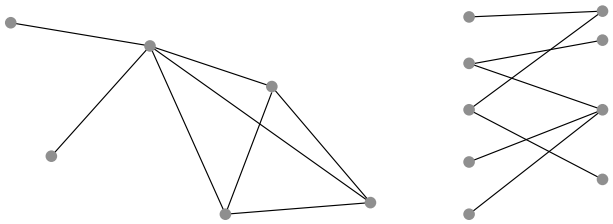
```
5
```

Solution

Graphs

Certainly, each reader is familiar with the following definition: a *graph* is an ordered pair $G = \langle V, E \rangle$, where V is an arbitrary set whose elements are called vertices and E is a set of unordered pairs of elements of V—the edges of G (thus an *edge* is a connection between two vertices).

Most of you probably also know that a *bipartite graph* is such a graph that its set of vertices V can be represented as a disjoint union of two sets, $V = X \cup Y$, $X \cap Y = \emptyset$, where all edges have one end in X and the other in Y. Bipartite graphs are often represented as ordered triples $G = \langle X, Y, E \rangle$, though they also fulfill the usual definition of a graph.

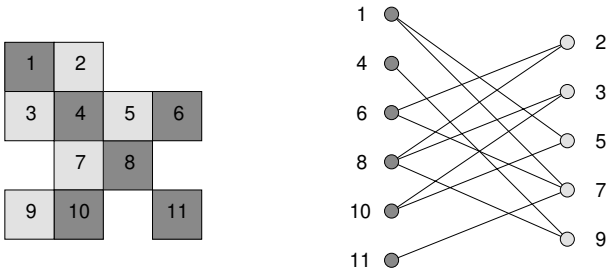


A graph (on the left) and a bipartite graph (on the right). The points represent the graphs' vertices and the lines represent the edges.

Independent set of vertices

For any graph $G = \langle V, E \rangle$, we can pose the following question: what is the maximal number of elements of a subset $I \subseteq V$ such that no two elements of I are connected by an edge? (Such a set is called an independent set of vertices.) This problem corresponds to a situation where the edges represent an exclusive choice among the vertices: one of them might be selected, but not both. It is exactly what we have in our problem: if we build a graph whose vertices are fields of the chess-board and edges represent all possible single moves of a knight, then we have to calculate the maximal cardinality of an independent set of vertices.

Please note that the graph obtained is a bipartite one, since a knight always jumps from a white field to a black one and vice-versa.



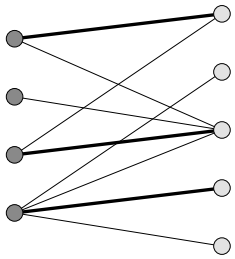
An example 4×4 chess-board with several fields removed and the corresponding bipartite graph of knight's moves.

The problem of a maximum independent set has its twin: minimum vertex cover. In this problem, we need to find the minimal cardinality of a set C of vertices such that each edge is 'covered' by at least one element of this set, that is at least one end of each edge falls into this set. It is an easy exercise to show that a complement of a vertex cover is an independent set and vice-versa, hence a complement of a minimum vertex cover is a maximum independent set. This means that if a graph has n vertices, $n = |V|$, then $|I| = n - |C|$, where C is the minimum vertex cover and I is the maximum independent set.

It is a well-known fact that the problems of the maximum independent set and of the vertex cover for arbitrary graphs are NP-complete, hence it is not known how to solve them in a polynomial time, or even if it is possible or not. For further information on this topic see e.g. [1]. However, we have already mentioned that we only consider bipartite graphs, not arbitrary ones. Luckily, in this case there exists a sufficiently fast solution. To find it, we need to learn a bit about matchings.

Matching

Finding the maximum matching is another graph problem of great importance. In a graph $G = \langle V, E \rangle$, a matching M is a subset of E such that no two different edges belonging to M have a vertex in common. The problem is of course to find the maximal number of elements of a matching in a given graph.



An example of a maximum matching in a bipartite graph.

In a bipartite graph, there exists a connection between the maximum matching M, minimum vertex cover C and the maximum independent set I. It turns out that the cardinality of the maximum matching exactly

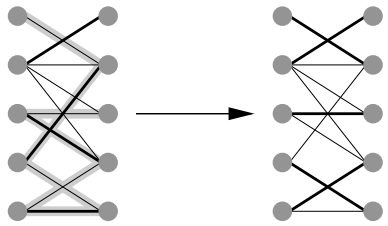
equals the cardinality of the minimum vertex cover, $|M| = |C|$, hence $|I| = |V| - |M|$. This equality, known as the 'Hungarian theorem' (by König and Egerváry), will be discussed later. Now, let us concentrate on finding the maximum matching.

Finding the maximum matching in general graphs can be done in time $O(n^{5/2})$. The algorithm is by Even and Kariv and is described in [2]. However, for bipartite graphs we know a much simpler and at least equally fast algorithm, namely the Hopcroft-Karp algorithm. Let us present it briefly.

Augmenting path

Consider any non-maximum matching M in a bipartite graph $G = \langle X, Y, E \rangle$. If a vertex is not matched, i.e. it is not an end of any of the edges belonging to M, than we shall call it *free*. Since G is bipartite, every path in this graph has even vertices in X and odd in Y, or vice-versa. An *alternating path* is a loop-free path in G starting at a free vertex in X and such that every even edge in this path belongs to M (hence every odd edge does not). If it also ends in a free vertex in Y, we call it an *augmenting path*.

An observant reader could have already guessed where the name of an augmenting path comes from. If we exchange matched and unmatched edges along an augmenting path, we obtain a matching that has one more edge.



An augmenting path in a bipartite graph.

Of course, if a matching is a maximum one, there are no augmenting paths. Luckily, the opposite also holds: if there are no augmenting paths, we have the maximum matching. (This fact is known as Berge's theorem and can be proven e.g. by considering the 'exclusive or' of a maximum matching and a matching that has fever elements.) This leads us to an algorithm for finding a maximum matching: search the graph for augmenting paths as long as there are any. If we use the breadth first search (BFS) method to find the shortest augmenting paths, then the duration of the entire computation would be $O(n(n+m))$, where $n = |V|$ and $m = |E|$[1], since BFS takes $O(n+m)$ steps and the size of the matching (that is, the number of augmenting paths found) is of course not greater than $\frac{n}{2}$. In our case, the arity of each vertex is 8 or less, hence we have at most 40 000 vertices and 160 000 edges. It leads to approximately $6.4 \cdot 10^9$ steps.

The Hopcroft-Karp algorithm

The idea of the Hopcroft-Karp algorithm is to enlarge the matching not by a single shortest augmenting path, but at once by the maximal set of vertex-disjoint augmenting paths of the shortest possible length. (Please note that by the maximal set we mean such a set of augmenting paths that no other augmenting path of the same length is disjoint with its members, so the set cannot be extended by adding any path, which does not imply that the set has the maximal possible number of elements.)

How to do it effectively? Assume $G = \langle X, Y, E \rangle$ is the graph, M is the matching to be augmented and l is the length of the shortest possible augmenting path. First, use BFS to build an auxiliary directed graph H as the union of all alternating paths of length less or equal to l (not necessarily disjoint). Note that if $x \in X$ and $y \in Y$ are free in M, then a path from x to y in H is an augmenting path of length l (if only it exists). So now we can repeat the depth first search (DFS) in H to find out consecutive augmenting paths and enlarge M using each of them in turn. Since we mark out every visited vertex, the paths found are pairwise disjoint indeed.

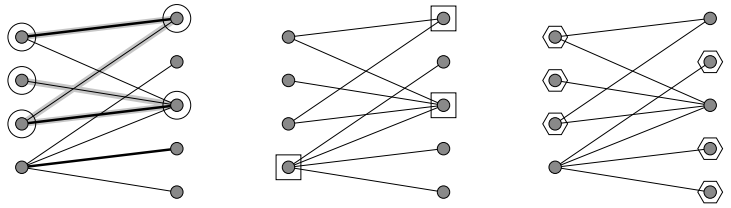
The BFS phase takes at most $O(n+m)$ time units. DFS is repeated many times, but it examines each vertex at most once, so it also takes $O(n+m)$ steps to accomplish. Hence we can have many augmenting paths at the same price as only one! Moreover, it can be proven that in such case we only need $O(\sqrt{n})$ repetitions of the above procedure to obtain the maximum matching. Hence the total computation time is $O(\sqrt{n}(n+m))$. In our case, where n can be as large as 40 000, it leads to a solution 200 times faster than the straightforward approach presented before.

[1] Please note that we use the standard notation: n for the number of vertices and m for the number of edges in a graph, though n and m have a different meaning in the problem 'Knights'.

'Hungarian theorem'

Now we already know how to solve the problem 'Knights', since we only need to calculate the size of the maximum independent set. It equals $|V| - |M|$. However, what would we do if we were to actually determine the independent set of maximal size, i.e. the positions of the knights on the chess-board?

Let us assume that we have a bipartite graph $G = \langle X, Y, E \rangle$. Let M be the maximum matching. Consider the union A of all sets of vertices of alternating paths in G. (Recall that an alternating path starts in a free vertex of X and first follows an unmatched edge, then a matched one etc.) It turns out that $C = (X \setminus A) \cup (Y \cap A)$ is the minimum vertex cover. Indeed, one can prove that C is a vertex cover (the proof is rather straightforward from the definition). Then we notice that no free (unmatched) vertices belong to C (this would contradict the fact that M is a maximum matching) and that for every edge $e \in M$, at most one of its ends belongs to C. Hence $|C| \leq |M|$. But every edge belonging to M has to be covered, and since these edges have no vertices in common, then there must be at most $|M|$ vertices in any vertex cover, in particular in C. Hence C is a minimum vertex cover, and its complement I is a maximum independent set.



On the left—a graph with a maximum matching and only one alternating path. The set A of its vertices is denoted by circles. In the middle—a minimum vertex cover of the graph. On the right—a maximum independent set.

Other solutions

All reasonable solutions of the problem are based upon the 'Hungarian theorem' and lean on finding the maximum matching. We have already mentioned the simple algorithm for maximum matchings of the complexity $O(n(n+m))$. If we give up finding the shortest augmenting path, we can use DFS instead of BFS to find the paths and use the first path found. Due to high regularity of the graph in our problem, if we always start our search at the vertex next to the last vertex visited in the previous search, we have a good chance to find a really short augmenting path in just several steps. It is much cheaper than performing an exhaustive search of the entire graph to make sure that our path is as short as possible. In fact, this simple heuristic often works as fast as the sophisticated Hopcroft-Karp algorithm or even faster, and could score a maximal number of points.

Tests

There were 10 input sets, among which only 3 were really large. Each of these tests was worth equal number of points. The following table describes briefly all of them (please note that now n and m denote the length of the edge of the chess-board and the number of fields missing respectively, hence $|V| = n^2 - m$).

no	n	m	$\lvert V\rvert$	description
1	2	0	4	full 2×2 board
2	2	1	3	2×2 board with one field missing
3	10	67	33	10×10 board, random fields missing
4	20	200	200	20×20 with regular cuts along two edges and some random fields missing
5	28	276	508	as the previous one, 28×28
6	40	913	687	as the previous one, 40×40
7	80	4233	2167	as the previous one, 80×80
8	180	14632	17768	as the previous one, 180×180
9	200	4	39996	200×200 with four fields near the corner missing (in the shape of letter L)
10	200	10100	29900	200×200, random fields missing

Task by
Wolfgang Pohl

Solution description by
Tomasz Waleń

Mars Maps

(This task was inspired by task 'Atlantis' of the Mid–Central European Regional ACM–ICP Contest 2000/2001.)

In the year 2051, several Mars expeditions explored different areas of the red planet and produced maps of these areas. Now, the BaSA (Baltic Space Agency) has an ambitious plan: they would like to produce a map of the whole planet. In order to calculate the necessary effort, they need to know the total size of the area for which maps already exist. It is your task to write a program that calculates this area.

Task

Write a program that:

- *reads the description of map shapes from the input file* `mar.in`,
- *computes the total area covered by the maps,*
- *writes the result to the output file* `mar.out`.

Input

The input file `mar.in` *starts with a line containing a single integer* N *(*$1 \leq N \leq 10\,000$*), the number of available maps. Each of the following* N *lines describes a map. Each of these lines contains four integers* x_1, y_1, x_2 *and* y_2 *(*$0 \leq x_1 < x_2 \leq 30\,000$, $0 \leq y_1 < y_2 \leq 30\,000$*). The values* (x_1, y_1) *and* (x_2, y_2) *are the coordinates of, respectively, the bottom-left and the top-right corner of the mapped area. Each map has rectangular shape, and its sides are parallel to the* x*- and* y*-axis of the coordinate system.*

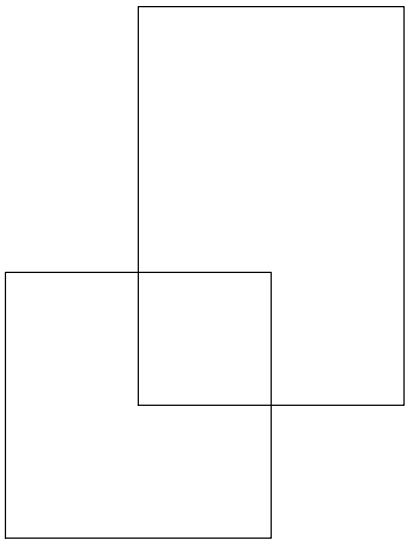
Output

The output file `mar.out` *should contain one integer* A*, the total area of the explored territory (i.e. the area of the union of all the rectangles).*

Example

For the input file `mar.in`*:*

```
2
10 10 20 20
15 15 25 30
```



the correct result is the output file `mar.out`*:*

```
225
```

Solutions

The task 'Mars Maps' has many possible solutions. As we shall see, some of them are very simple. We will present them starting with a simple but inefficient one and we will gradually improve the solutions' efficiency.

Solution 1

Let us suppose, that we have an array of Booleans of size $30\,000 \times 30\,000$, and we treat it as a big 2D grid. We can 'draw' all the given maps on it and then count the area covered. This is a straightforward solution and it can be easily implemented. However, it requires a huge amount of memory (almost 100 MB) and processing power (time complexity can be as big as $30\,000^2 \cdot (n+2)$). It can be very ineffective even for small tests (e.g. those with maps covering almost the whole area). Figure 1 shows a sample array generated by this solution.

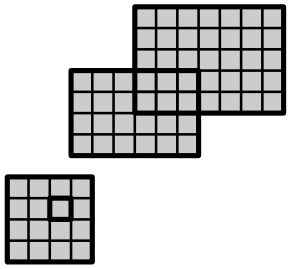


Fig. 1. Solution 1, drawing on the grid.

Solution 2

How can we improve the previous solution? The memory consumption should definitely be smaller and the solution should work faster (at least) for small tests. The main observation is that not all coordinates are used in the computations. The improved solution consists of two steps: The first step is creating a grid consisting of used coordinates only (i.e. coordinates of maps' corners), it is presented in fig. 2. The second step is exactly the same as in the previous solution: we draw the maps on the prepared grid. This time the computation of the covered area is slightly different. In the previous solution, the grid consisted of square fields 1×1. In this case, the area of each field of the grid has to be computed from the parameters of the grid.

This solution is slightly better, it does not depend on the range of the coordinates, but of course for large data it is still inefficient. The time complexity of this solution is $O(n^3)$ (since the 'compressed' grid might be as large as $n \times n$).

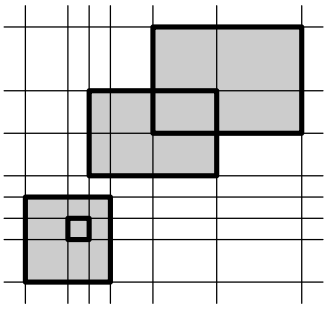


Fig. 2. Solution 2, compression of coordinates.

Solution 3

We can further improve the efficiency using a technique called 'scan–line sweeping'. It is based on scanning objects (maps) on a plane in certain order. In the previous solutions, the order was not important, but here we will make use of processing maps with an increasing x coordinate.

We can scan the plane of the Mars's surface with a vertical line (often called a 'brush'), from left to right. Each time the line encounters the left edge of some map, we mark it on the brush, and when the right edge is encountered we unmark it. For effective implementation, we need a fast method of computing the total length of marked edges on the brush.

Between subsequent edge encounters, the total length of the sum of marked edges is constant, so the map's area in this segment can be easily computed—it is $BrushLength \cdot \Delta x$, where Δx is the distance between x coordinates of subsequent edges encountered by the brush (see Fig. 3).

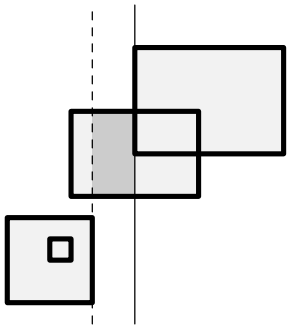


Fig. 3. Scan-line solution, the dashed line shows the previous position of the brush, the solid line shows the current one, and the darker area shows the maps' area lying between the two consecutive maps' edges.

Data structures

For the aforementioned algorithm, we need a data structure which implements the following operations:

- $MarkEdge(a, b)$—marking the edge $[a, b]$, on the brush,
- $UnmarkEdge(a, b)$—unmarking the edge $[a, b]$ on the brush (we can assume that such an operation always succeeds, i.e. the brush contains the given edge),
- $TotalLength$—computing the total length of the sum of all marked edges.

Let us denote by max_c the maximal value of the coordinates (in our case $max\_c = 30\,000$). From the problem specification, a and b are integers in the range $[0 \ldots max\_c]$.

The simplest implementation of the above data structure, is an array of integers of size $[0 \ldots max\_c]$. Each element of this array corresponds to one unit interval, and is equal to the number of marked edges containing it. The implementation of $MarkEdge(a, b)$ and $UnmarkEdge(a, b)$ is fairly simple: elements of the array in the range $[a, b-1]$ need only to be incremented or decremented by one. However, such an implementation is quite slow. In the pessimistic case it requires $O(max\_c)$ operations. Nevertheless it is an improvement compared to the previous solution, however it can be improved even further.

The way to do it is to use balanced trees. We can build a tree over the range $[0..max\_c]$ in such a way that its leaves correspond to one unit intervals, while the internal nodes correspond to the intervals being the sum of the intervals of their descendants.

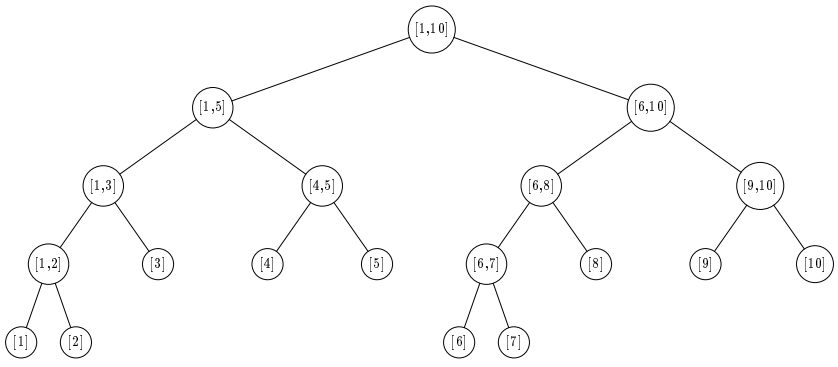


Fig. 4. An example of a balanced tree for the range [1..10].

Each node contains two values: $Length$—the length of the covered part in the node's corresponding interval, and $Count$—the number of edges covering it.

The $TotalLength$ operation returns simply the value of $Length$ of the tree root. The other operations are a bit more complex. We try to avoid operating on small intervals: each time we discover an operation that would cause changes in all the leaves of a node's subtree, we only need to change the values in the node.

More precisely, we follow the following schema:

- For a given edge $[a, b]$ we start from the root (node number 1).
- Let v be the current node, corresponding to the interval $[l, r]$, and let v' and v'' be its direct descendants, corresponding to the intervals $[l, m]$ and $[m+1, r]$ respectively (where $m = (l + r)\ div\ 2$). We may encounter the following cases:

- the interval $[l, r]$ is fully contained in $[a, b]$—in such a case we modify the values in the current node v and return to its parent,
- the intervals $[l, r]$ and $[a, b]$ intersect—if $[l, m]$ and $[a, b]$ intersect, we descend to the left child v', if $[m+1, r]$ and $[a, b]$ intersect, we go to the right child v''. It may be necessary to go in both directions, however it will not happen too often as we will see later on. After visiting the children we return to the parent.
- the intervals $[l, r]$ and $[a, b]$ do not intersect—this situation is impossible, since we never descend to such a node.

Moreover, each time we return from a child node it is necessary to recompute the value of *Length* from the values stored in the child nodes. Computing the *Length* value is rather straightforward: if the node's *Count* is positive, then the *Length* value is equal to $b + 1 - a$, otherwise it equals $v'.Length + v''.Length$.

We can implement these operations in the following way:

```
1: procedure modifyInterval(node, y0, y1, node_y0, node_y1, delta)
2: begin
3:   if (y0 ≤ node_y0) and (y1 ≥ node_y1) then begin
4:     v[node]=v[node]+delta;
5:   end else begin
6:     m:= (node_y0+node_y1) div 2;
7:     if (y0≤m) then modifyInterval(node*2, y0, y1, node_y0, m, delta);
8:     if (m<y1)  then modifyInterval(node*2+1, y0, y1, m+1, node_y1, delta);
9:   end
10:  correct(node, node_y0, node_y1);
11: end;
```

```
1: procedure correct(node, node_y0, node_y1);
2: begin
3:   if (v[node]>0) then sum[node] := 1+node_y1−node_y0;
4:   else begin
5:     sum[node] := 0;
6:     if (node_y0<node_y1) then
7:       sum[node] := sum[node*2]+sum[node*2+1]
8:   end;
9: end;
```

We still have to prove that the procedure *modifyInterval* runs in time $O(\log maxc)$. It may happen that a descent to both the left and the right node will be needed, but it causes some additional cost only when it happens for the first time. After that first 'split', every following one is very simple: the cost of processing one of the children is constant, since we return to the parent immediately after descending to it (see fig. 5). The time complexity comes from the observation that the total height of the tree is equal to $O(\log maxc)$.

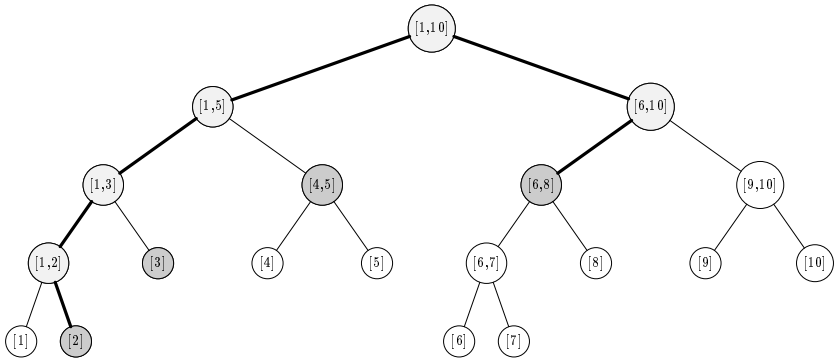


Fig. 5. The result of adding the interval $[2, 8]$, visited nodes are coloured gray, and those with modified values are darkened.

The Algorithm

Having implemented an effective data structure, the main algorithm might be written in the following way:

```
1:  last_x := 0;
2:  for e = GetLeftmost(Edges) do begin
3:    (x, y0, y1, sign) := e;
4:    area := area+sum[1]*(x-last_x);
5:    last_x := x;
6:    if (sign = +1) then
7:      modifyInterval(1, y0, y1, 0, max_x, +1);
8:    else
9:      modifyInterval(1, y0, y1, 0, max_x, -1);
10: end;
```

The variable *Edges* contains an ordered sequence of vertical edges of Mars Maps; it is ordered by the ascending x coordinate and (for the edges with the same coordinate) by the sign ($+1$ for left edges, -1 for right edges).

Since we have the data structure which implements the procedure $ModifyInterval$ with complexity $O(\log maxc)$, and in the whole run of the algorithm it will be called exactly $2n$ times, then the total time complexity is $O(n \log maxc)$.

Tests

Ten test data sets were used during the evaluation of the contestants' solutions. They are briefly described below.

no	n	Remarks
1	4	simple test
2	7	simple test
3	13	image composed of rectangles with text: 'BOI'
4	100	random rectangles
5	200	snail like shape
6	1000	random rectangles
7	2000	large rectangles placed in X shape
8	5000	many vertical and horizontal bars
9	10000	two groups of squares
10	10000	rhombus shape

Task and solution description by
Krzysztof Onak

Teleports

Great sorcerer Byter created two islands on the Baltic See: Bornholm and Gotland. On each island he installed some magical teleports. Each teleport can work in one of two modes:

- *receiving—one can be teleported to it,*
- *sending—anyone who enters the teleport is transfered to the specific destination teleport on the other island, provided that the other teleport is in the receiving mode.*

Once, Byter gave his apprentices the following task: they must set the teleports' modes in such a way, that no teleport is useless, i.e. for each teleport set in the receiving mode there must be at least one teleport sending to it, set in the sending mode; and vice versa, for each teleport set in the sending mode, the destination teleport must be set in the receiving mode.

Task

Write a program that:

- *reads the description of the teleports on both islands, form the input file* `tel.in`,
- *determines the appropriate modes for the teleports,*
- *writes the result to the output file* `tel.out`.

If there are several possible solutions, your program should output just one of them.

Input

In the first line of the text file `tel.in`, *there are two integers m and n, $1 \leq m, n \leq 50\,000$, separated by a single space; m is the number of teleports on Bornholm, and n is the number of teleports on Gotland. Teleports on both islands are numbered from 1 to m and n respectively. The second line of the file contains m positive integers, not greater than n, separated by single spaces—the k-th of these integers is the number of the teleport on Gotland that is the destination of the teleport k on Bornholm. The third line contains analogous data for the teleports on Gotland, i.e. n positive integers, not greater than m, separated by single spaces—k-th of these integers is the number of the teleport on Bornholm that is the destination of the teleport k on Gotland.*

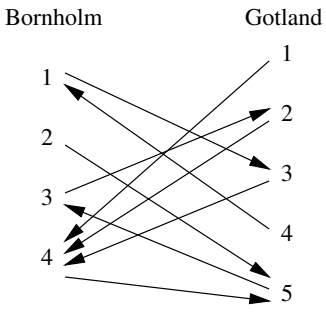
Output

Your program should write two lines describing the modes of the teleports, respectively, on Bornholm and Gotland, to the output file `tel.out`. *Both lines should contain a string of, respectively, m and n ones and/or zeros. If the k-th character in the string is* `1`, *then the k-th teleport is in the sending mode, and if it is* `0`, *then the corresponding teleport is in the receiving mode.*

Example

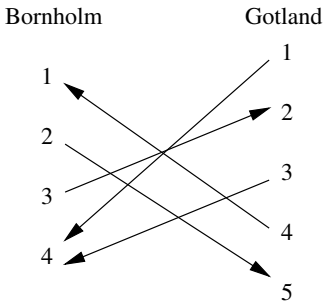
For the input file `tel.in`:

```
4 5
3 5 2 5
4 4 4 1 3
```



the correct result is the output file `tel.out`:

```
0110
10110
```



Solution

Let us start with an easy observation. Sometimes input data determine modes of some teleports even when there are many possible solutions (we will see later that a solution always exists). For example if there is a teleport on one of the islands, for which there is no teleport on the other island that could send anything to it, we know that such a teleport must be in the sending mode, otherwise it would be useless. Teleport no. 1 on Gotland in the example input data (fig. 1) is just such a device.

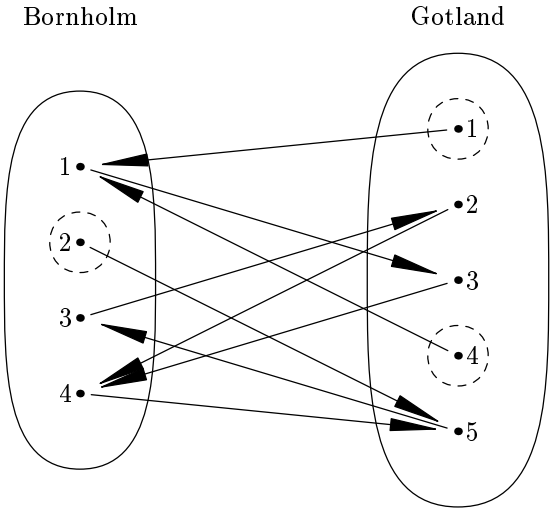


Fig. 1. Teleports in dashed circles are forced to be in the sending mode due to the lack of teleports that could send to them.

Moreover its destination teleport (no. 1 on Bornholm) must be in the receiving mode due to the requirements of the task. Besides, it is entailed that no teleport in the solution can send to device no. 3 on Gotland, so it must be in the sending mode and so on. Certainly, this simple fact and its further consequences will not help us to find a solution in every situation. See figure 2 for an example.

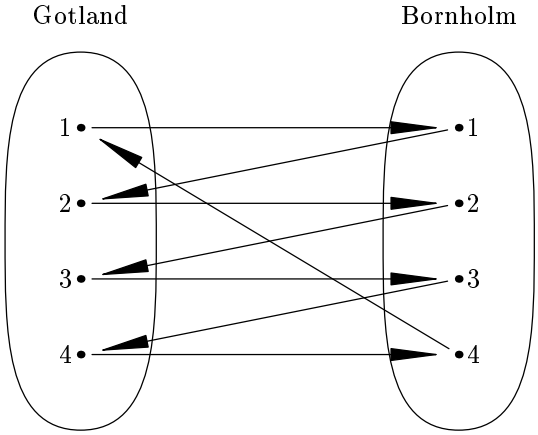


Fig. 2. None of the teleports is determined to be in a specific mode.

There are four teleports on each island and for each of them there exists a device sending to it. Besides, there are exactly two solutions—all teleports on Gotland are in the sending mode and all on Bornholm in the receiving mode and vice versa. It means that none of the teleports is determined to be in a specific mode. In spite of this our previous observation is not useless.

Algorithm

Let us try a simple method. First we set all teleports on Bornholm in the receiving mode, and all teleports on Gotland in the sending mode. Of course it does not have to be a solution, but we will try to repair it by looking for teleports for which there is no device sending to it as we have done before while analyzing the problem. Here is the algorithm:

```
1: set all teleports on Bornholm in the receiving mode;
2: set all teleports on Gotland in the sending mode;
3: while there is a useless teleport x in receiving mode on Bornholm do
4:   begin
5:     switch x to sending mode;
6:     switch destination teleport of x to receiving mode;
7:   end
```

Why does it terminate? With every step of the loop in lines 3–7 the number of teleports in the receiving mode on Bornholm decreases, and because it is always finite, at some point there will be no device x as specified in line 3.

Why is a state of teleports in row no. 8 a solution? There may be four groups of teleports and we will show that devices in each group cannot be useless.

- Receiving mode, Bornholm—the algorithm has stopped, so due to the condition in the fourth line none of them is useless.
- Sending mode, Bornholm—the destination teleport of every such a device has been switched to the receiving mode (row no. 6) and its state has not been altered anymore.
- Receiving mode, Gotland—a device is in this mode, because it was a destination teleport for some teleport on Bornholm that has been switched to the sending mode, so it is not useless.
- Sending mode, Gotland—destination teleports of theses devices were initially set in the sending mode and have not been altered.

This proves the correctness of the algorithm. Now we have to implement our solution choosing appropriate data structures.

Implementation

Our program has to write to the output file a number of characters proportional to the total number of teleports on both islands. It means that the time complexity of our solution cannot be better than $\Theta(n+m)$. Later on, we will see that such a complexity can be achieved. We will use the same names as in the example program.

At the beginning we read numbers n, m and destinations of all teleports—we store them in the array `dest` so that we can get the destination device for a specific teleport in constant time. In the array `sending` we keep the modes of teleports. It is initialized according to lines 1–2 of the algorithm. For each teleport on Bornholm we count for how many devices on Gotland it is a destination and store this information in the array `inDegree`.

Then, we prepare an empty structure `zeroDeg` which provides the following operations: inserting an item into the structure, getting out any of the items previously inserted and testing if the structure is empty. Each of the operations should take constant time. This structure can be implemented for example as a stack or a queue.

We put all the teleports on Bornholm that are currently useless, these are the ones for which `inDegree` is zero, into `zeroDeg`. All the steps done so far require $\Theta(n+m)$ time.

Now, the loop in lines 3–7 starts. Checking if there is a useless teleport on Bornholm is simply checking if `zeroDeg` is empty. If it is, we have finished, otherwise we take a teleport number from `zeroDeg`. Let us call this teleport x. We switch x to the sending mode and we set its destination device y in the receiving mode. Next we decrease `inDegree` for the destination z of the teleport y. If z becomes useless, it means that `inDegree` for z has reached 0 and we put z into `zeroDeg`. It is clear that every execution of the loop takes $O(1)$ time, and because it happens at most m times, we know that the time complexity of this part of the algorithm is $O(m)$.

Hence, the program needs $\Theta(n+m)$ memory and time and according to the previous observation it is optimal.

Inspiration

For an arbitrary set X let us denote by $\mathcal{P}(X)$ the set of all subsets of X. Now let us define for arbitrary sets X, Y and a function $f : X \to Y$ a function $\vec{f} : \mathcal{P}(X) \to \mathcal{P}(Y)$ as:

$$\vec{f}(A) = \{f(a) | a \in A\}$$

for all $A \subseteq X$.

As an author of the problem I can reveal that it was inspired by the theorem called 'Banach's lemma'. It says what follows:

> For arbitrary functions $f : A \to B$ and $g : B \to A$ there exist sets $A_1, A_2 \subseteq A$ and $B_1, B_2 \subseteq B$ such that
>
> - $A_1 \cup A_2 = A$, $A_1 \cap A_2 = \emptyset$,
> - $B_1 \cup B_2 = B$, $B_1 \cap B_2 = \emptyset$,
> - $\vec{f}(A_1) = B_1$,
> - $\vec{g}(B_2) = A_2$.

Speaking less formally it says that we can divide both A and B into two parts (respectively A_1, A_2 and B_1, B_2) such that applying f to A_1 we get exactly B_1 and applying g to B_2 we get A_2 (see figure 3).

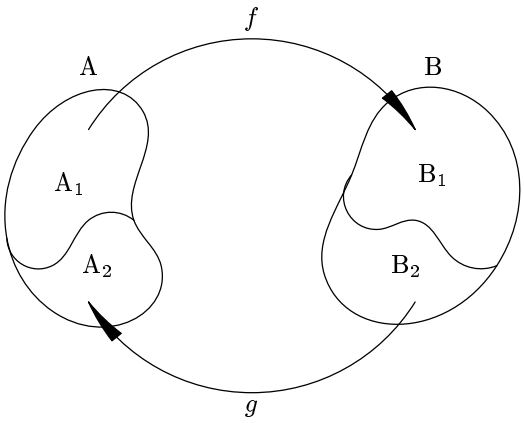


Fig. 3. The presentation of the idea of Banach's lemma.

We can treat Bornholm and Gotland as sets including teleports. Then relation origin-destination is a function in both directions. What is more, we have proved Banach's lemma for finite sets A and B. It is much harder to show that it is true even when both sets are infinite.

Tests

There were 10 tests used to evaluate contestants' programs. Most of them were random and a few were prepared against some simple heuristics and programs with worse complexity.

No	m	n
1	10	10
2	100	107
3	1 000	1 005
4	1 000	1 000
5	10 006	10 000
6	10 000	10 000
7	50 000	1 000
8	1 000	50 000
9	50 000	49 987
10	50 000	50 000

Bibliography

[1] Thomas H. Cormen, Charles E. Leisereson, Ronald L. Rivest. *Introduction to Algorithms.* The MIT Press, 1990.

[2] S. Even, O. Kariv. An $O(n^{2.5})$ algorithm for maximum matching in general graphs. In *Proc. 16th Annual Symp. on Foundations of Computer Sciece*, pages 100–112. IEEE, 1975.