TIUP 2011

Torneio Inter-Universitário de Programação

First Stage

http://mooshak.di.fc.ul.pt

April 20th, 2011

17:00 - 20:00

Scientific Committee

Francisco Martins <fmartins@di.fc.ul.pt>

François Aubry <fran.aubry@gmail.com>

Hugo Miranda <hmiranda@di.fc.ul.pt>

Isabel Nunes <in@di.fc.ul.pt>

João Neto <jpn@di.fc.ul.pt>

General Information

1. The contest follows ICPC rules (to know more, go to http://icpc.baylor.edu/).
2. It has a duration of 3 hours for the 5 problems.
3. The programs must read from the standard-input and write to the standard-output.
4. The generated test inputs use the following norms:
   - There are no spaces at the end of the lines and each line ends with a carriage return.
   - No multiple spacing is being used, unless it is explicitly mentioned on the problem description.
5. The generated test outputs follow the same rules.
6. All problems have 1 second timeout.
7. Each team (3 persons maximum) should use one PC and it is strictly forbidden to use any online resource other than Mooshak or what is given in the local PC disk.
8. Further information can be obtained by clicking on "Help" on your Mooshak account screen.
9. Please ask your questions to the jury using the "Questions" button.

Compilers

Language	**Compiler**	**Command Line**	**Extension**
C	GCC v. 4.2.4	`gcc -Wall -lm`	.c
C++	GCC v. 4.2.4	`g++ -Wall`	.cpp
Java	Sun JDK v. 1.6.0_11	`javac`	.java
Pascal	Free Pascal v. 2.2.2	`fpc -v0w -oprog`	.pas
Perl	Perl v. 5.8.8	`perl -c`	.pl
Python	python v. 2.4.3		.py

Note: Programs producing **warnings** or errors during compilation will be automatically rejected by the system.

Problem A

Addition Matrix

John loved the logic quizes that appear on every newspaper. At some point, he decided to make things harder and began to create algorithms that would equally solve the problems so that he could validate his solutions without having to wait for next day's newspaper edition.

John's favourite class of problems is the addition matrix. In this class of problems, a matrix like the one on the picture below is presented.

	+		-		+		-		=	7
+		-		+		-		+		
	-		+		+		-		=	-1
+		-		-		+		+		
	+		-		+		-		=	5
+		+		+		-		-		
	-		+		-		-		=	-1
-		+		-		+		-		
	+		-		+		-		=	3
=		=		=		=		=		
13		5		5		5		5		

The goal of these problems is to fill the empty spaces with numbers so that the results of the sequence of operations in each row and column is correct. For example, the problem above is solved by:

5	+	4	-	3	+	2	-	1	=	7
+		-		+		-		+		
4	-	3	+	2	+	1	-	5	=	-1
+		-		-		+		+		
3	+	2	-	1	+	5	-	4	=	5
+		+		+		-		-		
2	-	1	+	5	-	4	-	3	=	-1
-		+		-		+		-		
1	+	5	-	4	+	3	-	2	=	3
=		=		=		=		=		
13		5		5		5		5		

However, John soon found out that his amateur programming skills were not enough for solving the problem. Your task is to help John to develop the algorithm.

Problem Description

Given a sequence of operators and equation results, find integers that make all the equations correct.

Constraints

1. All cells must be filled with an integer in the interval [1..5]
2. The same number cannot appear twice in the same row or column
3. Only additions and subtractions are used
4. Each presented problem has exactly one solution

Input Description

Input starts with a line with two integers r, c ($3 \leq r, c \leq 9$), which are respectively the number of rows and columns of the matrix. The number of rows and columns does not include those used for equal signs or results.

r more rows follow, each describing the sequence of operations on that row (a + or a - sign). In every other row of the matrix, the sequence of operations is followed by the result that must be satisfied.

A final row contains c integers, each describing the result of one column.

Output Description

The output is composed of $\frac{r+1}{2}$ rows, with the digits that should appear on that row.

Input Example 1

```
9 9
+-+-7
+-+-+
-++--1
+--++
+-+-5
+++--
-+---1
-+-+-
+-+-3
13 5 5 5 5
```

Output Example 1

```
54321
43215
32154
21543
15432
```

Input Example 2

```
9 9
+-+-5
---+-
++--5
-++-+
+-+-3
--++-
-+---1
+-+++
--+--3
-3 -3 5 9 7
```

Output Example 2

```
45312
14523
51234
32451
23145
```

Input Example 3

```
3 7
+++11
-+-+
-+-1
1 4 -1 6
```

Output Example 3

```
5321
4135
```

Problem B

Avoiding explosions

A secret service developed a new kind of explosive that attain its volatile property only when a specific association of products occurs. Each product is a mix of two different simple compounds, which we call a binding pair. Mixing N different binding pairs containing N simple compounds creates a powerful explosive, if N>2. For example, the binding pairs A+B, B+C, A+C result in an explosive, while A+B, B+C, A+D (three pairs, four compounds) does not. You are not a secret agent but only a guy in a delivery agency with one dangerous problem: receive binding pairs in sequential order and place them in a cargo ship. However, you must avoid placing in the same room an explosive association. So, after placing a set of pairs, if you receive one pair which might produce an explosion with some of the pairs already in stock, you must refuse it. An example: let's assume you receive the following sequence: A+B, G+B, D+F, A+E, E+G, F+H. You would accept the first four pairs but then refuse E+G since it would be possible to make the following explosive with the previous pairs: A+B, G+B, A+E, E+G (4 pairs with 4 simple compounds). Finally, you would accept the last pair, F+H.

Problem Description

Compute the number of refusals given a sequence of binding pairs.

Input Description

Instead of letters we will use integers. Each input line (except the last) consists of two integers N, M separated by a single space. The file's last line only has the number –1.

Output Description

A single line with the number of refusals.

Constraints

1. $0 \leq N, M \leq 10^5$
2. No repeated binding pairs in the input.

Input Example

```
1 2
3 4
3 5
3 1
2 3
4 1
2 6
6 5
-1
```

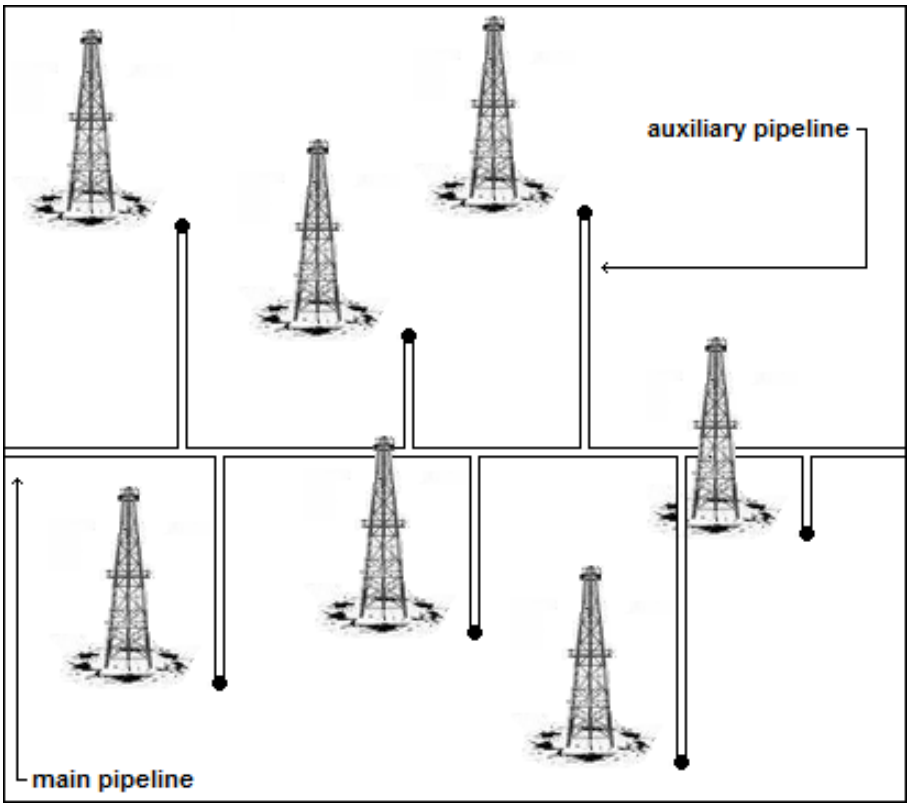
Output Example

```
3
```

Problem C

Oil Derricks

Professor Jones is consulting for an oil company, which is planning a large horizontal pipeline running east to west through an oil field of N wells. From each well, an auxiliary pipeline is to be connected directly to the main pipeline along the shortest path (either north or south, check the image bellow for clarity).



Problem Description

Given the coordinates *(x, y)* of the wells, how should the professor pick the location of the main pipeline in order to minimize the sum of the lengths of the auxiliary pipelines? Note that the main pipeline may be placed at any y coordinate, even if it passes over an oil derrick point.

Input Description

The first line of the input contains a single integer N, the number of oil derricks. Then N lines follow, each with two integers x_i y_i separated by a single space, representing the coordinates of the i-th oil derrick.

You may assume that no two oil derricks are at the same location.

Output Description

A single line with the minimum length of the sum of the auxiliary pipelines.

Constraints

1. $1 \leq N \leq 10^5$
2. $0 \leq x_i, y_i \leq 1000$

Input Example 1

```
3
2 2
4 6
8 4
```

Output Example 1

```
4
```

Input Example 2

```
4
1 5
1 1
1 4
1 6
```

Output Example 2

```
6
```

Input Example 3

```
1
1 1
```

Output Example 3

```
0
```

Problem D

Super Knapsack

You and your friends are planning on going camping. Any good camper knows that it is really annoying to travel with a lot of stuff. On the other hand, one needs to have all the necessary equipment so it is important to find a good balance. You have a knapsack and a list of items you want to take to the camp. After making your selection you are left with N objects. Now comes the hard task: how to fit the most in the knapsack? The knapsack has a capacity C and each object occupies c_i units of that capacity. When you try to pack you realize that if you sort the items by increasing size they satisfy $c_i > c_1 + ... + c_{i-1}$, for each $i > 1$.

Problem Description

Write a program that decides if it is possible to fit the knapsack exactly and if so, what items to take.

Input Description

The first line of the input contains two integers C and N separated by a single space, denoting the capacity of the knapsack and the number of objects, respectively. Then N lines follow, each with one integer c_i and a string (a single word on the alphabet {a, ..., z}) separated by a single space, that represent the size of the i-th object and its name.

Output Description

If it is possible to exactly fit the knapsack, print the object names in alphabetical order. If it is not possible print "Impossible" (with no "").

Constraints

1. $1 \leq N \leq 63$
2. $0 \leq c_i \leq 2^{64}$

Input Example 1

```
24 5
2 watch
3 knife
7 lantern
15 food
31 tent
```

Output Example 1

```
food
lantern
watch
```

Input Example 2

```
21 5
4 plates
1 compass
8 blanket
16 water
2 clothes
```

Output Example 2

```
compass
plates
water
```

Input Example 3

```
20 4
5 juice
4 chips
21 books
10 shoes
```

Output Example 3

```
Impossible
```

Problem E

Mastering Bureaucracy

Bureaucracy is the administrative structure of any large organization, characterized by hierarchical authority relations functioning under impersonal, uniform rules and processes. Its goal is to be rational, efficient, and professional. So, whenever a decision needs to be taken, a dedicated employee, the solicitor, elaborates a document exposing the problem, submits it to the approval of his boss, and waits patiently for a verdict. In his turn, the boss may have the authority to decide on the matter and signs the document, if the decision lies in its scope of jurisdiction; otherwise, he, himself, puts in writing his considerations in yet another document and submits it to his boss for approval, together with additional information he finds relevant. Thereafter, the boss of the boss of the dedicated employee analyses all the documents and may or may not decide by himself, and so on. I guess you all know what I am talking about.

Problem Description

We want to put in order this bureaucratic process, not by finding a better alternative to it, but by controlling if the information that is submitted for approval corresponds to that the decider is expecting, if the decider exists at all, etc. For that, we put together a small language for describing bureaucratic processes that goes like the following:

1. **NewDoc** d, which creates a new document with contents abstracted by d in the authority scope of the current decider. In our abstraction, the document content is irrelevant, only the identifier matters;
2. **Sign** a, that produces a signature a from a decider;
3. **Available** $l\ s_1\ \ldots\ s_n$, that makes available n sources of information identified by $s_1\ \ldots\ s_n$ at link l. These sources can be identifiers of document, signatures, or any other links introduced before in the current scope of authority. The idea is that information per se is not sent around. Instead, links are made available and people can refer to them to obtain the actual information. A link acts as a binder for grouping information together;
4. **Access** $l\ n\ s$, which allows a decider to access the information available at position n of link l and bind it to source s for further use in the decider;
5. **RequestApp** $d\ a\ l_1\ \ldots\ l_n$, that requests decider d a signature a (a brand new identifier) for approving the matters backed up by n sources available at links $l_1\ \ldots\ l_n$; and finally
6. **Decider** $d\ a\ l_1\ \ldots\ l_n$, which defines a decider block identified by d that will eventually produce a signature a and that requires information sources $l_1\ \ldots\ l_n$ to make his decision.

In particular, we want to enforce the following restrictions:

(a) in 1), identifier d is brand new in the current decider's scope of jurisdiction (abbreviate as scope, in what follows);

(b) in 2), identifier a corresponds to a requested signature for the current decider (introduced by the decider's definition (refer to 6. above));

(c) in 3), identifier l is brand new in the current scope and identifiers $s_1\ \ldots\ s_n$ correspond to previously introduced documents or links in scope (either by **NewDoc**, **Available**, **Access**, or **Decider**);

(d) in 4), identifier l was introduced before in scope (as in c.), n is a valid position to access information available at l, and s is a completely new identifier for the current scope that will hold the information at that location;

(e) in 5), identifier d refers to an existing decider defined by a **Decider** instruction, a represents the signature that will be produced for this request (and therefore is a newly introduced identifier) and the n sources of information identifiers $l_1 \ldots l_n$ must all be existing links in the current scope and conform to the information requested by the decider; finally,

(f) in 6), we must enforce that there is one (unique) decider identified by d in the document. A decider definition makes available identifiers a, $l_1 \ldots l_n$ in a new scope, private to the decider, during the execution of its decision process. In any circumstance a decider can be used as a source.

Input Description

The input file contains two or more blocks. Each block begins with a decider definition (**Decider**) and ends with signature (**Sign**) followed by an empty line (the file always ends with an empty line as well). The body of the block is composed by the instructions **NewDoc**, **Available**, **Access**, or **RequestApp** (as described in the Problem's section). Each instruction is written in a single line. The first block is always named *solicitor* with a single signature parameter. The remaining blocks begin with a **Decider** definition and proceed with the deciders algorithm described by instructions (as in the first block).

Output Description

A single line containing either a *yes* or a *no* whether the program conforms to the restrictions described above (from *(a)* to *(f)*).

Constraints

Concerning the input file you may rely on two facts:

1. The file is always written according to the syntax, so you do not need to account for syntactic errors.
2. Decisions in a hierarchy in a bureaucracy follow a pyramid like scheme forming a directed acyclic graph, so you may assume that the deciders in the file are arranged in such a way (a topological sort) that a **RequestApp** will always precede the **Decider** definition (except for the 1st block, the one that defines *solicitor*).

Also, be aware that each introduced identifier is visible until the end of the current block.

Example 1

In this example the *solicitor* requests an authorization to *boss1*, providing him a link to document *doc* that he created. In its turn, *boss1* makes his decision after collecting the approvals from *boss2* and *boss3*. Notice that *boss1* puts together a new document that has the same name, *doc*, as the one created by *solicitor*, but the names do not clash, since they are introduced in different decider

scopes. Also, observe that *boss3* gains access to *solicitor*'s document by drilling down on *docSource* using **Access** twice.

Input Example 1

```
Decider solicitor signature
NewDoc doc
Available docLink doc
RequestApp boss1 signBoss1 docLink
Sign signature

Decider boss1 sign1 docSource1
NewDoc doc
Available docLinkA doc docSource1
RequestApp boss2 signBoss2 docLinkA
Available signBoss2Link signBoss2
RequestApp boss3 signBoss3 docLinkA signBoss2Link
Sign sign1

Decider boss3 sign docSource signOtherBoss
Access docSource 2 initialDocLink
Access initialDocLink 1 doc
Sign sign

Decider boss2 sign docSource
Sign sign
```

Output Example 1

```
yes
```

Example 2

In this example the *solicitor* is trying to request an authorization to *boss1*. Nevertheless, the example has several errors, namely *solicitor* (a) declares a document with the name *docLink*, already introduced (by **Available**); (b) requests *boss1* authorization providing *docLink*, but the definition of *boss1* (in **Decider**) demands for two sources (*docSource1* and *docSource2*); *boss1* (c) tries to access *docSource1* in a position that is not defined; (d) requests for the authorization of a decider, *boss2*, that is not defined; and (e) signs the wrong identifier *docSource1*, instead of *sign1*.

Input Example 2

```
Decider solicitor signature
NewDoc doc
Available docLink doc
NewDoc docLink
RequestApp boss1 assBoss1 docLink
Sign signature
```

Decider boss1 sign1 docSource1 docSource2
NewDoc doc
Access docSource1 3 doc
RequestApp boss2 signBoss2 docLinkA
Sign docSource1

Output Example 2

```
no
```