Problem A

(Program filename: A.CPP or A.PAS or A.DPR or A.java)

Parencodings

Let $S = s_1\ s_2\ \ldots\ s_{2n}$ be a well-formed string of parentheses. S can be encoded in two different ways:

- By an integer sequence $P = p_1\ p_2\ \ldots\ p_n$ where p_i is the number of left parentheses before the ith right parenthesis in S (P-sequence).
- By an integer sequence $W = w_1\ w_2\ \ldots\ w_n$ where for each right parenthesis, say a in S, we associate an integer which is the number of right parentheses counting from the matched left parenthesis of a up to a. (W-sequence).

Following is an example of the above encodings:

S	(((()())))
P-sequence	4 5 6666
W-sequence	1 1 1456

Write a program to convert *P*-sequence of a well-formed string to the *W*-sequence of the same string.

Input (filename: A.IN)

The first line of the input file contains a single integer t ($1 \le t \le 10$), the number of test cases, followed by the input data for each test case. The first line of each test case is an integer n ($1 \le n \le 20$), and the second line is the *P*-sequence of a well-formed string. It contains n positive integers, separated with blanks, representing the *P*-sequence.

Output (filename: A.OUT)

The output file consists of exactly t lines corresponding to test cases. For each test case, the output line should contain n integers describing the *W*-sequence of the string corresponding to its given *P*-sequence.

Sample Input

```
2
6
4 5 6 6 6 6
9
4 6 6 6 6 8 9 9 9
```

Sample Output

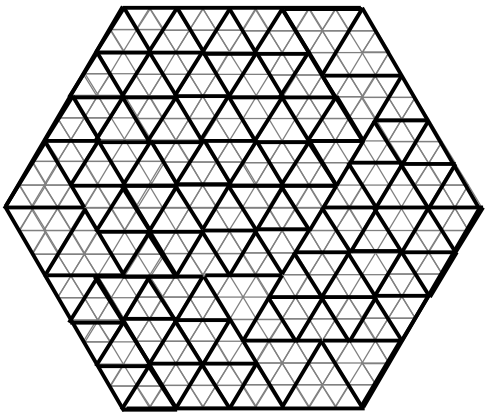
```
1 1 1 4 5 6
1 1 2 4 5 1 1 3 9
```

Problem B

(Program filename: B.CPP or B.PAS or B.DPR or B.java)

The Bermuda Triangle

People in the hidden region of the Bermuda Triangle make everything they need in triangular shapes. One day, someone decided to break the rule and bake a hexagonally shaped cake. But as usual, he has to serve the cake in triangular pieces. The pieces are equilateral triangles but in different sizes for different people. He can use as many triangles as needed to cut the cake into pieces, such that nothing remains from the cake. For example, the following figure shows one way that a hexagon with side 9 can be cut into triangles with side 2 and 3. (The cake is cut along the thick lines, thin lines are drawn to show the sizes).



Input is a hexagon and triangle types (specified by the length of their sides) and the goal is to decide if the hexagon can be completely divided by the given triangle types.

Input (filename: B.IN)

The first line of the input file contains a single integer t ($1 \le t \le 10$), the number of test cases, followed by the input data for each test case. Each test case consists of a single line, containing s ($1 \le s \le 25$), the length of the hexagon's side, followed by n, the number of triangle types ($1 \le n \le 10$), followed by n integers representing the length of each triangle type's side (between 1 and 25, inclusive).

Output (filename: B.OUT)

There should be one output line per test case containing either `YES` or `NO` depending on whether the hexagon can be completely divided by the given triangle types.

Sample Input

```
3
5 2 2 3
7 2 3 2
13 2 2 3
```

Sample Output

```
NO
NO
YES
```

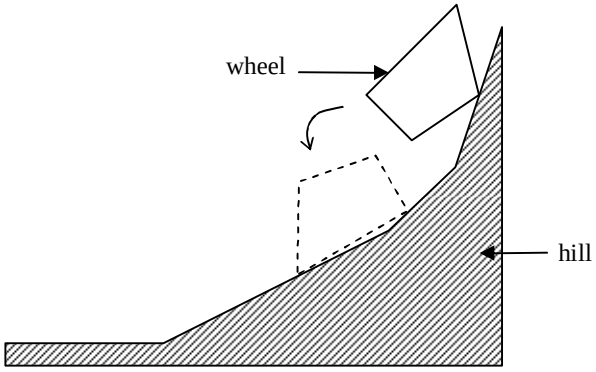
Problem C

(Program filename: C.CPP or C.PAS or C.DPR or C.java)

Deformed Wheel

The village's carpentry is located by a hill side. The carpenter's two little boys play with a piece of wood which looks like a deformed wheel with two identical convex polygon-shaped faces. One boy sets the wooden wheel on a slope at the hill top and let it roll down. The other boy is to quickly place himself at where he guesses the rolling wood would stop. Your program is to help him make the right guess.

More formally, we consider the wooden wheel as a simple convex polygon and we approximate the hill by a sequence of connected line segments with decreasing slopes. The slope of the last segment in the sequence is assumed to be zero, and the slope of the first segment is assumed to be a positive number. Initially, the wheel is placed on the hill such that there is at least one point of contact between the wheel and segments. For example in the following figure, the wheel in its initial position is drawn in solid lines, while the final position is drawn in dashed lines.



At any instant, the wheel rotates around one of its vertices, say P, if the y-coordinate of its center of gravity is decreased (note that this condition is necessary at *any* instant during the motion). It can be easily shown that at any instant, there is at most one such vertex. Rotation around P is stopped when the wheel touches a segment. The motion continues until no vertex can be found such that the wheel can rotate around it. At any instant, assume that changing the position of the center of gravity in any direction for at most 10^{-5} units, does not affect the stability of the wheel. Also assume that the friction between the wheel and the surface of the hill is so high that the wheel never slides on the surface.

Input (filename: C.IN)

The first line of the input file contains a single integer t ($1 \le t \le 10$), the number of test cases, followed by the input data for each test case. In the first line of each test case there is an integer n ($1 \le n \le 10$), that indicates the number of the wheel vertices. In each of the next n lines, there is a pair of numbers which are x and y coordinates of the initial position of a vertex. After this, there is a single line containing the initial x and y coordinates of the center of gravity of the wheel. You can assume that the center of gravity is inside or on the boundary of the polygon (note that the given center of gravity is not necessarily computable from wheel's geometric shape). Next lines of the test data will describe the shape of the hill. The surface of the hill is approximated with a series of line segments with decreasing slopes ending with a horizontal line segment. For each segment, there is a line containing length and slope of a segment (both of them are real numbers). The lines are ordered in decreasing slope (The last line of this part of the input has slope zero). You can assume that the last (horizontal) line is long enough that the wheel would not pass its end. In the last line of the test case, there is a line containing the x and y coordinates of the right end-point of the first segment. All coordinates and slopes are real numbers.

Output (filename: C.OUT)

For each test case, there should be a single line in the output file, containing two numbers which are *x* and *y* coordinates of the wheel's center of gravity. Round the numbers in the output to 3 digits after decimal point.

Sample Input

```
1
4
40 30
30 37
24 30
30 26
27 29
30 1
100 0
40 30
```

Sample Output

```
28.854 20.031
```

Problem D

(Program filename: D.CPP or D.PAS or D.DPR or D.java)

Illusive Chase

Tom the robocat is presented in a Robotics Exhibition for an enthusiastic audience of youngsters, placed around an $m \times n$ field. Tom which is turned off initially is placed in some arbitrary point in the field by a volunteer from the audience. At time zero of the show, Tom is turned on by a remote control. Poor Tom is shown a holographic illusion of Jerry in a short distance such that a direct path between them is either vertical or horizontal. There may be obstacles in the field, but the illusion is always placed such that in the direct path between Tom and the illusion, there would be no obstacles. Tom tries to reach Jerry, but as soon as he gets there, the illusion changes its place and the chase goes on. Let's call each chase in one direction (up, down, left, and right), a *chase trip*. Each trip starts from where the last illusion was deemed and ends where the next illusion is deemed out. After a number of chase trips, the holographic illusion no more shows up, and poor Tom wonders what to do next. At this time, he is signaled that for sure, if he returns to where he started the chase, a real Jerry is sleeping and he can catch it.

To simplify the problem, we can consider the field as a grid of squares. Some of the squares are occupied with obstacles. At any instant, Tom is in some unoccupied square of the grid and so is Jerry, such that the direct path between them is either horizontal or vertical. It's assumed that each time Tom is shown an illusion; he can reach it by moving only in one of the four directions, without bumping into an obstacle. Tom moves into an adjacent square of the grid by taking one and only one step.

The problem is that Tom's logging mechanism is a bit fuzzy, thus the number of steps he has taken in each chase trip is logged as an interval of integers, e.g. 2 to 5 steps to the left. Now is your turn to send a program to Tom's memory to help him go back. But to ease your task in this contest, your program should only count all possible places that he might have started the chase from.

Input (filename: D.IN)

The first line of the input file contains a single integer t ($1 \le t \le 10$), the number of test cases, followed by the input data for each test case. The first line of each test case contains two integers m and n, which are the number of rows and columns of the grid respectively ($1 \le m, n \le 100$). Next, there are m lines, each containing n integers which are either 0 or 1, indicating whether the corresponding cell of the grid is empty (0) or occupied by an obstacle (1). After description of the field, there is a sequence of lines, each corresponding to a chase trip of Tom (in order). Each line contains two positive integers which together specify the range of steps Tom has taken (inclusive), followed by a single upper-case character indicating the direction of the chase trip, which is one of the four cases of R (for right), L (for left), U (for up), and D (for down). (Note that these directions are relative to the field and are not directions to which Tom turns). This part of the test case is terminated by a line containing exactly two zeros.

Output (filename: D.OUT)

For each test case, there should be a single line, containing an integer indicating the number of cells that Tom might have started the chase from.

Sample Input

```
2
6 6
0 0 0 0 0 0
0 0 0 1 1 0
0 1 0 0 0 0
0 0 0 1 0 0
0 0 0 1 0 1
0 0 0 0 0 1
1 2 R
1 2 D
1 1 R
0 0
3 4
0 0 0 0
0 0 0 0
0 0 0 0
1 2 R
3 7 U
0 0
```

Sample Output

```
10
0
```

Problem E

(Program filename: E.CPP or E.PAS or E.DPR or E.java)

Puzzle Out

The scientific committee members of the 26th ACM/ICPC, who design the contest problems, use the following encryption algorithm to communicate the problem drafts securely through the Internet. To encrypt a text, all occurrences of each letter is replaced with another letter (possibly itself), such that no two letters are encrypted to the same letter. Both original and encrypted texts consist of only upper-case letters and blanks. Blanks are not encrypted and are repeated exactly in the encrypted text. As an example, the string `GSRH RH GSV URIHG HZNKOV` is the encrypted form of `THIS IS THE FIRST SAMPLE` according to the encryption table (A $\rightarrow$ Z, B$\rightarrow$ Y, C $\rightarrow$ X, …, Z $\rightarrow$ A).

A recipient of a problem draft has lost the encryption table, but he has a dictionary which includes all the possible words appearing in the problems. You are to help him set up a decryption table to enable him restore the original problem draft from the encrypted one. Given a dictionary of the original words used in the text, and the encrypted text, we want to find the right encryption table such that after decrypting the given encrypted text back to the original one, all words can be found in the dictionary.

Input (filename: E.IN)

The first part of the input file is a dictionary of English words common to all test cases. The first line of the file is d ($1 \le d \le 50000$); the number of words in the dictionary, followed by d lines each containing a word in the dictionary. The words in the dictionary are sorted in alphabetical order and all are in uppercase. Each word has at most 20 characters, but you can assume that sum of the length of all words in the dictionary is no more than 350,000. The next line contains a single integer t ($1 \le t \le 10$), the number of test cases, followed by the input data for each test case. Each test case, which is preceded by a single blank line, consists of multiple lines in the input file forming the encrypted text. Each line has a string containing only uppercase letters and blank. You may assume that no line break is occurred in the middle of a word and there may be arbitrary number of blank characters at the end of each line. Maximum length of input lines is 80.

Output (filename: E.OUT)

The output file contains exactly t lines, each corresponding to a test case. Each line should contain a single string of 26 characters which is the encryption of the string `ABCDEFGHIJKLMNOPQRSTUVWXYZ` according to the encryption table used in the test case. Letters in the output string should be in uppercase. It is possible that some letters do not appear in the encrypted text at all. In this case, put a * mark in place of those letters not appearing in the decrypted version of the input text. If the test case has no solution, the output line should contain `#No solution#.` If there is more than one possible encryption table for a test case, the output line should contain `#More than one solution#`.

Sample Input

```
14
BE
CHANGE
FIRST
IN
IS
MUST
SAMPLE
SEE
THE
THIS
TO
WISH
WORLD
YOU
4

GSRH RH GSV URIHG HZNKOV

IZM BMVU    SP UGP
RGTANP IZM KFVG UZ VPP
FA UGP KZWCQ

XYZ ABCDEFG

XZY ABD
```

Sample Output

```
Z***VU*SR**ON**K*IHG******
TSRQP*NGF**CBAZ**WVUM*K*I*
#No solution#
#More than one solution#
```

Problem F

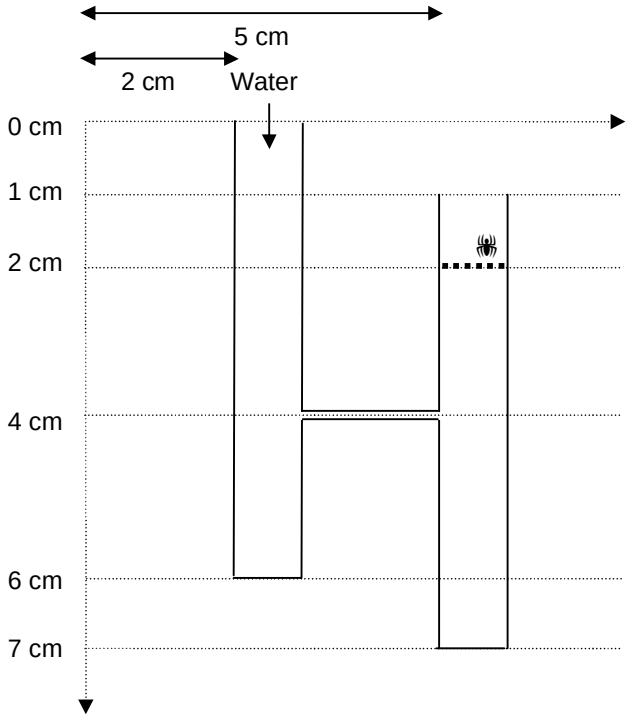
(Program filename: F.CPP or F.PAS or F.DPR or F.java)

The Willy Memorial Program

Willy the spider used to live in the chemistry laboratory of Dr. Petro. He used to wander about the lab pipes and sometimes inside empty ones. One night while he was in a pipe, he fell asleep. The next morning, Dr. Petro came to the lab. He didn't notice Willy while opening the valve to fill the pipes with hot water. Meanwhile, Stanley the gray mouse got what was going to happen. No time to lose! Stan ran hard to reach the valve before Willy gets drawn, but... Alas! He couldn't make it!

Poor Willy was boiled in hot water, but his memory is still in our hearts. Though Stan tried his best, we want to write a program, in the memory of Willy, to compute the time Stan had, to rescue Willy, assuming he started to run just when the doctor opened the valve.

To simplify the problem, assume the pipes are all vertical cylinders with diameter 1 cm. Every pipe is open from the top and closed at the bottom. Some of the pipes are connected through special horizontal pipes named *links*. The links have very high flow capacity, but are so tiny that at any given time, the volume of water inside them is negligible. The water enters from top of one of the pipes with a constant rate of 0.25π cm^3/sec and begins to fill the pipe from the bottom until the water reaches a link through which it flows horizontally and begins to fill the connected pipe. From elementary physics we know if two pipes are connected and the surface of the water is above the connecting link, the level of water in both pipes remains the same when we try to fill one of them. In this case the water fills each pipe with a rate equal to half of the rate of incoming water. As an example, consider the following configuration:



First, the lower 2 centimeters of the left pipe is filled with water at full rate, then, the lower 3 centimeters of the right pipe is filled, and after that, the upper part of the two pipes are filled in parallel at half rate. The input to your program is a configuration of pipes and links, and a target level in one of the pipes (the heavy dotted line in the above figure). The

program should report how long it takes for the level of water to reach the target level. For the above configuration, the output is 9 seconds.

It is assumed that the water falls very rapidly, such that the time required for the water to fall can be neglected. The target level is always assumed to be a bit higher than the specified level for it. As an example, if we set the target point to level 4 in the left pipe in the figure above, the elapsed time for water to reach that target is assumed to be 5 (not 2), Also note that if the water reaches to the top of a pipe (say in level x), it won't pour out outside the pipe until empty spaces in connected pipes below level x are filled (if can be filled, i.e. the level of water reaches the connecting links). (Note that there may be some links at level x, to which water is entered). After all such spaces are filled; the water level would not go up further.

Input (filename: F.IN)

To describe positions, we assume the coordinates are expressed as (x, y) and the origin lies in the top-left of all pipes and links. (Note that y coordinates are increased downwards). All coordinates are integer numbers between 0 and 100, inclusive.
The first line of the input file contains a single integer t ($1 \le t \le 10$), the number of test cases, followed by the input data for each test case. The first line of each test case is p ($1 \le p \le 20$), the number of pipes, followed by p lines, each describing a pipe. Each pipe description line consists of three numbers. The first two are (x, y) coordinates of the upper-left corner of the pipe and the third number is the height of the pipe (at least 1 cm and at most 20 cm). Note that diameter of each pipe is 1 cm.

After input data describing the pipes, there is a line containing a single integer l, which is the number of links ($0 \le l \le 50$). After it, there are l lines describing links. Each link description contains 3 integers. The first two are (x, y) coordinates of the left end-point of the link and the third is the length of the link (at least 1 cm and at most 20 cm). It is assumed that the width of the link is zero.

The last line for each test case contains two numbers. The first is the number of target pipe (starting from one, with the order appeared in test data). The second line is the desired y for the level of water in the target pipe (note that the specified level may be out of the pipe at all).

You can assume the following about the input:

- The water enters into the first pipe.
- No link crosses a pipe.
- No two links have the same y coordinates.
- No two pipes have the same upper-left x coordinates.
- Both endpoints of each link are connected to pipes.

Output (filename: F.OUT)

The output file should contain exactly t lines with no blank lines in between, each corresponding to one test case. Each output line should contain the time required for the water to reach the target level in the target pipe (an integer number). If in a specific test case, the water never reaches the target level, the line should contain `No Solution` string in it.

Sample Input

```
1
2
2 0 6
5 1 6
1
3 4 2
2 2
```

Sample Output

```
9
```

Problem G

(Program filename: G.CPP or G.PAS or G.DPR or G.java)

Parallel Expectations

We are to predict some facts about the behavior of a single processor designed for running two programs in parallel. Programs are sequences of commands according to the following grammar:

<Program> → <Command> *
<Command> → <Variable> := <Operand> <Operator> <Operand>
<Operator> → + | -
<Operand> → <Variable> | <Constant>

A <Variable> is a sequence of (at most 20) alphanumeric characters (A...Z, a...z, and 0...9) starting with a letter (not case sensitive). A <constant> is an unsigned integer number (less than 100). There may be arbitrary number of blank or tab characters between tokens.

Before execution, programs are translated into machine language. A statement of the form X := Y + Z is translated to the following set of machine instructions:

```
Mov R1, Y
Mov R2, Z
Add R1, R2
Mov X, R1
```

A `MOV` instruction copies the content of its second operand into its first operand. An `Add` (`Sub`) instruction, adds (subtracts) its second operand from its first operand and the result is stored in the first operand. Note that Y and Z denote either a variable or an integer constant. Instructions generated for the command X := Y - Z is similar to the above instructions, except that `Sub` command is used instead of `Add`.

The processor is given two machine language programs and starts executing them from the first instruction. In each step, it randomly selects one of the two programs and runs the next instruction from the selected program. This continues until one program reaches its end. In this situation, the remaining instructions from the other one are executed sequentially to the end and the processor stops. It is assumed that all variables are shared between two programs, but each program has a separate register set. The goal of this program is to compute the expected final value of all variables among all possible executions of the programs. More precisely, we want to consider every possible execution of the two programs and for each variable, calculate the average of its final value in different executions. It is assumed that the initial value of all variables is zero.

Input (filename: G.IN)

The first line of the input file contains a single integer t ($1 \le t \le 10$), the number of test cases, followed by the input data for each test case. The data for each test case consists of a pair of programs. Each program is written as a sequence of consecutive lines, each line containing exactly one command. Programs end with a line containing only the word `END`. You may assume that no variable in any program is named 'END'. There is no blank line between programs of one test case. There are at least one and at most 25 lines in each program. Total number of variables in two programs is no more than 10.

Output (filename: G.OUT)

For each test case, the output file should contain the expected final value of all variables in alphabetical order of variable names (digits precede letters in this order). Output for different test cases should be separated by exactly one blank line. Round the numbers in the output to 4 digits after decimal point. Do not omit trailing zeros after decimal point (e.g. write 1.2000 instead of 1.2).

Sample Input

```
1
S := 1 + 3
END
S := S+S
END
```

Sample Output

```
3.0000
```

Problem H

(Program filename: H.CPP or H.PAS or H.DPR or H.java)

University Entrance Examination

There is a fierce competition among high-school graduates in Iran to pass the centralized nationwide university entrance examination. Ministry of Science, Research, and Technology has set up the Education Evaluation Organization (EEO) to take care of all aspects of this big exam. This year the EEO managed to select some 150,000 students to enter universities out of 1.4 million high school graduates participated in a tough 4.5 hours multiple-choice exam. This annual event is usually preceded by a multi-billion Rial business offering preparatory courses to enthusiastic students. A few weeks after the big exam day, each participant receives a score sheet, and a list of Field-Department-University (FDU), displaying each field of study in the universities' departments (e.g., the Software Engineering field of Computer Engineering department at Sharif University of Technology) along with their capacity for that year. The eligible participants (those who have scored enough to be allowed to declare their FDU priorities) fill out a priority indication form, and declare the FDUs they like to enter, in the order of their preference. The EEO processes the forms, and considering the total score, the participant's FDU priority list, and some other selection rules, enters the accepted participants' names in the list of each FDU, until all capacities are exhausted. Those who are not entered in a list are considered failed and may try again next year. Each accepted participant's name may be entered in only one list.

One of the interesting selection rules is to persuade participants to enter universities in the vicinity of their home towns. This is to help reduce the number of requests for staying in the university dormitories.

The selection process is so complex and so sensitive to many, that EEO has decided to hire the very best programmers in Iran to design a new selection algorithm and write a completely new program for what they have been doing for years. ACM programming contest is where these programmers can be found.

There are N students S_1 to S_N, and M items F_1 to F_M, each representing one of the FDUs. There are also a number of geographic regions. For each participant, the total score, the geographic region where his/her high school diploma was awarded, and a priority list of his/her wanted FDUs are available. For each FDU, the geographic region where the corresponding university is located, and its capacity for that year is recorded.

Write a program to compute the list of accepted students with the FDU they can enter to, given the above list of input data. Your program must abide with the following rules:

1. (Local student selection rule) Suppose two students A and B have both selected F in their priority lists and F is in region R. Also suppose that score of A is greater than B's score. Then, if B is from region R (local) and A is from other regions (non-local), and B's score is greater than 70% of A's score, then B has priority over A to enter F. In *all* other cases A has priority over B to enter F.
2. (Fairness rule) Students should be treated according to their priority list of FDUs. That is, an accepted student will be accepted to the first possible FDU he/she can enter.

Note: We assume that scores are all different integer values.

Input (filename: H.IN)

The first line of the input file contains a single integer t ($1 \le t \le 10$), the number of test cases, followed by the input data for each test case. The first line of each test case contains N ($1 \le N \le 150$) and M ($1 \le M \le 50$) followed by N lines, each for one student. The format of these lines is R_i, M_i, K, F_{i1}, …, F_{iK} in this order. In this line, that is for student i, R_i is his/her region number, M_i is his/her score in the entrance exam, K is the number of FDUs in his/her priority list ($0 \le K \le M$), and his/her priority list containing the FDU numbers in order of interest. Then there are M lines, one for each FDU. Each line contains R_i, and C_i in that order, which respectively is region number of F_i (the ith FDU) and the capacity of F_i. Note that region numbers are arbitrary integers.

Output (filename: H.OUT)

Outputs for different test cases are separated by exactly one blank line. For each test case, you should write N lines, one for each of the N students. If student i has been accepted to FDU F_j, then ith line should contain j, and `not accepted`, if that student has not been accepted in any FDU of his/her interest.

Sample Input

```
1
9 2
1 100 2 1 2
2 80 2 2 1
1 90 1 1
2 40 1 2
2 50 1 1
1 60 1 2
2 75 1 1
1 95 1 1
2 30 1 2
1 3
2 4
```

Sample Output

```
1
2
1
2
not accepted
2
not accepted
1
2
```