Problem A: Where's Waldorf?

Input file: wal.in
Output file: wal.out

Given an *m* by *n* grid of letters and a list of words, find the location in the grid at which the word can be found. A word matches a straight, uninterrupted line of letters in the grid, regardless of case. The matching can be done in any of the eight directions either horizontally, vertically or diagonally through the grid.

Input specification

The input begins with a number on a line by itself, indicating the number of test cases that follow. Each test case consists of a pair of integers, *m* followed by *n*, 1 <= *m* <= 50 and 1 <= n <= 50 in decimal notation on a single line. The next *m* lines contain *n* letters each; this is the grid of letters in which the words of the list must be found. The letters in the grid are in upper or lower case. Following the grid of letters, another integer *k* appears on a line by itself (1 <= *k* <= 20). The next *k* lines of input contain the list of words to search for, one word per line. These words contain upper and lower case letters only (no spaces, hyphens or other non-alphabetic characters). All words have positive length.

Output specification

For each word in the word list of one test case, your program must output a pair of integers representing the location of the corresponding word in the grid. The integers must be separated by a single space. The first integer is the line in the grid where the first letter of the given word can be found (1 represents the topmost line in the grid, and *m* represents the bottommost line). The second integer is the column in the grid where the first letter of the given word can be found (1 represents the leftmost column in the grid, and *n* represents the rightmost column in the grid). If a word can be found more than once in the grid, then the location which is output should correspond to the uppermost occurrence of the *first* letter of the word (*i.e.* the occurrence which places the first letter of the word closest to the top of the grid). If two or more words are uppermost, the output should correspond to the leftmost of these occurrences. All words can be found at least once in the grid. Outputs of different test cases are separated by a blank line.

Sample input

```
2
2 2
by
re
2
be
ab
8 11
abcDEFGhigg
hEbkWalDork
```

```
FtyAwaldORm
FtsimrLqsrc
byoArBeDeyv
Klcbqwikomk
strEBGadhrb
yUiqlxcnBjf
4
Waldorf
Bambi
Betty
Dagbert
```

Sample output

```
1 1
2 2

2 5
2 3
1 2
7 8
```

Problem B: All Roads Lead Where?

Input file: `rom.in`
Output file: `rom.out`

There is an ancient saying that "All Roads Lead to Rome". If this were true, then there is a simple algorithm for finding a path between any two cities. To go from city A to city B, a traveller could take a road from A to Rome, then from Rome to B. Of course, a shorter route may exist.

The network of roads in the Roman Empire had a simple structure: beginning at Rome, a number of roads extended to the nearby cities. From these cities, more roads extended to the next further cities, and so on. Thus, the cities could be thought of as existing in *levels* around Rome, with cities in the *i*th level only connected to cities in the *i-1*st and *i+1*st levels (Rome was considered to be at level 0). No loops existed in the road network. Any city in level *i* was connected to a single city in level *i-1*, but was connected to zero or more cities in level *i+1*. Thus, to get to Rome from a given city in level *i*, a traveller could simply walk along the single road leading to the connected *i-1* level city, and repeat this process, with each step getting closer to Rome.

Given a network of roads and cities, your task is to find the shortest route between any two given cities, where distance is measured in the number of intervening cities.

Input specification

The first line of input contains two numbers in decimal notation separated by a single space. The first number (*m*) is the number of roads in the road network to be considered. The second number (*n*) represents the number of queries to follow later in the file.

The next *m* lines in the input each contain the names of a pair of cities separated by a single space. A city name consists of at most ten letters, the first of which is in uppercase. No two cities begin with the same letter. The name `Rome` always appears at least once in this section of input; this city is considered to be at level 0, the lowest-numbered level. The pairs of names indicate that a road connects the two named cities. The first city named on a line exists in a lower level than the second named city. The road structure obeys the rules described above. No two lines of input in this section are repeated.

The next *n* lines in the input each contain the names of a pair of cities separated by a single space. City names are as described above. These pairs of cities are the *query pairs*. Your task for each query pair is to find the shortest route from the first named city to the second. Each of the cities in a query pair is guaranteed to have appeared somewhere in the previous input section describing the road structure.

Output specification

For each of the *n* query pairs, output a sequence of uppercase letters indicating the shortest route between the two query pair cities. The sequence must be output as

consecutive letters, without intervening whitespace, on a single line. The first output line corresponds to the first query pair, the second output line corresponds to the second query pair, and so on. The letters in each sequence indicate the first letter of the cities on the desired route between the query pair cities, including the query pair cities themselves. A city will never be paired with itself in a query.

Sample input

```
7 3
Rome Turin
Turin Venice
Turin Genoa
Rome Pisa
Pisa Florence
Venice Athens
Turin Milan
Turin Pisa
Milan Florence
Athens Genoa
```

Sample output

```
TRP
MTRPF
AVTG
```

Problem C: Hoppers

Input file: hop.in
Output file: hop.out

Hoppers are people on a jump stick who can jump from one square to the other, without touching the squares in between (a bit like a knight in chess). They can pick up speed and make bigger hops, but their acceleration per move is limited, and they also have a maximum speed. The game of Hoppers is played on a rectangular grid, where each square on the grid is either empty or occupied. While hoppers can fly over any square, they can only land on empty squares. At any point in time, a hopper has a velocity (x,y), where x and y are the speed (in squares) parallel to the grid. Thus, a speed of (2,1) corresponds to a knight jump, (as does (-2,1) and 6 other speeds).

To determine the hops a hopper can make, we need to know how much speed a hopper can pick up or lose: either -1, 0, or 1 square in either or both directions. Thus, while having speed (2,1), the hopper can change to speeds (1,0), (1,1), (1,2), (2,0), (2,1), (2,2), (3,0), (3,1) and (3,2). It is impossible for the hopper to obtain a velocity of 4 in either direction, so the x and y component will stay between -3 and 3 inclusive.

The goal of Hoppers is to get from start to finish as quickly as possible (i.e. in the least number of hops), without landing on occupied squares. You are to write a program which, given a rectangular grid, a start point S, and a finish point F, determines the least number of hops in which you can get from S to F. A hopper starts with initial speed (0,0) and does not care about the speed when arriving at F.

Input specification

The first line contains the number of test cases (N) your program has to process. Each test case consists of a first line containing the width X (1 <= X <= 16) and height Y (1 <= Y <= 16) of the grid. Next is a line containing four integers separated by blanks, of which the first two indicate the start point (x1,y1) and the last two indicate the end point (x2,y2), where the start and end point are valid points on the grid (that is, 0 <= x1 < X, 0 <= x2 < X, 0 <= y1 < Y, and 0 <= y2 < Y). The third line of each test case contains an integer P indicating the number of obstacles in the grid. Finally, the test case consists of P lines, each specifying an obstacle. Each obstacle consists of four integers: x1, x2, y1 and y2, (0 <= x1 <= x2 < X, 0 <= y1 <= y2 < Y), meaning that all squares (x,y) with x1 <= x <= x2 and y1 <= y <= y2 are occupied. Neither the start point nor the finish point will ever be occupied.

Output specification

The string 'No solution.' if there is no way the hopper can reach the finish point from the start point without hopping on an occupied square. Otherwise, the text 'Optimal solution takes *N* hops.', where *N* is the number of hops needed to get from start to finish point.

Sample input

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

Sample output

```
Optimal solution takes 7 hops.
No solution.
```

Illustration of sample input

The example input can be illustrated as follows. Let `#` denote an occupied square, `S` denote the start point, `F` denote the finish point, and `*` denote the boundary of the grid. Then the sample input can also be given as follows.

```
2
5 5
*******
*    F*
* ####*
* ####*
*     *
*    S*
*******
3 3
*****
* #F*
*###*
*S# *
*****
```