Tenth Annual Ohio Wesleyan University Programming Contest

April 2, 2016

Rules:

1. There are six questions to be completed in four hours.
2. All questions require you to read the test data from standard input and write results to standard output. **Do not** use files for input or output. **Do not** include any prompts, other debugging information, or any other output except for exactly what is specified by the problem.
3. The allowed programming languages are C, C++ and Java.
4. All programs will be re-compiled prior to testing with the judges' data.
5. Non-standard libraries cannot be used in your solutions. The Standard Template Library (STL) and C++ string libraries are allowed.
6. Programming style is not considered in this contest. You are free to code in whatever style you prefer. Documentation is not required.
7. All output cases will end in a newline character. So after your program finishes output, the cursor should be on the first column of the line immediately below your last case's output.
8. All communication with the judges will be handled in the PC^2 environment.
9. Judges' decisions are final. No cheating will be tolerated.

Problem A: Workin' On the Railroad.

It's 1877, and railways are being built to connect the country together. An enterprising rail baron back east, J.P von Smoot III, has decided to enter the game. He's bought several defunct rail companies and would like to use them to build a large railway connecting far-flung cities.

Mr. von Smoot has commissioned a survey of the wilderness for you and provided you with a chart of the terrain. The potential area has been divided into rectangular cells, and each cell marked with its predominant terrain feature:

- Clear spaces, which cost $1 to connect to a rail line (really, it's probably "$1 thousand" or "$1 million", but Mr. von Smoot doesn't like excess zeros)
- Forest spaces, which cost $2 to connect to a rail line
- Mountain spaces, which cost $3 to connect to a rail line
- Track spaces (places that have pre-laid track Mr. Van Smoot has acquired), which cost $0 to connect to a rail line.

Two cells will also have cities. The start city is assumed to already have track, the ending city is assumed to be clear. All connections must be made in rectilinear directions (so no diagonals).

Your task is to find a way to connect two candidate cities into one rail network (possibly, but not necessarily, using pre-existing rails) minimizing the total cost to Mr. von Smoot.

Input:

There will be several input instances. Each will begin with two numbers, r and c (1 ≤ r, c ≤ 50), indicating the number of rows and columns in the terrain grid. A value of r=c=0 will denote end of input.

There will then follow two pairs of rows and columns denoting the rows and columns of the two cities to connect. Then will follow a number f, the number of forest cells. Then there will be f pairs of rows and columns denoting the locations of forest cells. Then will follow a number m, the number of mountain cells, followed by m pairs of rows and columns denoting the locations of mountain cells. Then there will follow a number t, the number of track spaces, followed by t pairs of rows and columns denoting the locations of track cells.

All terrain locations (tracks, cities, forests, or mountains) will be on the grid, and assume that row 0, column 0 is the upper-left corner of the grid and row r-1 and column c-1 is the bottom right.

No space will have more than one terrain feature (forest, mountain, track, or city). A space that has no terrain feature is assumed to be clear.

Output:

For each input case i, output:

```
Case i: $X
```

Where "X" is the cost of the minimum path connecting the three cities, following the rules above.

Sample Input:

```
5 5
0 0
4 4
3
2 1
2 4
3 4
2
0 1
1 1
2
2 2
2 3
3 8
1 0
1 7
1
1 1
1
1 6
4
1 2
1 3
1 4
1 5
0 0
```

Sample Output:

```
Case 1: $7
Case 2: $6
```

Problem B: Mix and Max Operations

Given a sequence of digits from 0-9 (for example: 1,2,3,4,5,6,7,8,9}, we can combine the numbers in that sequence into numbers (perhaps 12,345,6,and 789) and place operators + and * in between them (perhaps 12+345*6+789), and then evaluate that equation using the normal order of operation rules.

What's interesting is that when doing this there are sometimes multiple ways to reach the same number. For example, given the sequence {1,2,3,4,5,6,7,8,9} we can make 100 by 1+2+3+4+5+6+7+8*9 or 12+34+5*6+7+8+9 (and several other ways). Now, aren't you curious how many ways we can combine numbers like this to get the same result? I knew it!

Input:

There will be several input instances. Each will begin with a number n ($1 \le n \le 10$), the number of digits in the sequence. A value of n=0 denotes end of input. Otherwise, the n digits of the sequence will follow on the next line. The last line will be the positive integer target value t (≤ 2,000,000,000). The sequence of numbers may repeat or omit numbers between 0 and 9, and may be in any order. The order of numbers in the sequence cannot be changed.

Output:

For each input instance i, output the line:

```
Case i: k
```

..where "k" is the number of ways we can reach t using all of the numbers in the sequence in the correct order, by concatenating digits, or inserting + or * operations between numbers.

Two "ways" are different if they result in different-looking equations. Two groupings that are the "same" by commutativity but involve different groupings are different. For example, sample input 3 has two ways to make 24: 22+2 and 2+22

Sample Input:

```
9
1 2 3 4 5 6 7 8 9
100
2
2 2
4
3
2 2 2
24
4
2 3 4 5
120
4
2 3 4 5
119
3
0 0 7
7
0
```

Sample Output:

```
Case 1: 7
Case 2: 2
Case 3: 2
Case 4: 1
Case 5: 0
Case 6: 5
```

Problem C: Which One-Time Pad?

A *one-time pad* is an encryption mechanism consisting of long sequence of characters (one character for each symbol in the message to be encrypted). Each character in the message is added to the corresponding character in the pad to get an encrypted message. Adding 'A' keeps the letter the same, adding 'B' increases the letter by 1, and so on. Additions that go beyond 'Z' wrap around.

So, if your plaintext message is "JELLY" and your pad is "ABCDE", you would get:

Plaintext Letter	J	E	L	L	Y
Pad Letter	A	B	C	D	E
Ciphertext	J	F	N	O	C

..for a final ciphertext of "JFNOC".

Decryption, of course, "subtracts" the letter, instead of adding it.

Since a one-time pad exists to encrypt a given plaintext message into *any* ciphertext message (and vice versa), this method of encryption, while tedious, is 100% secure.

That is, assuming that the sender and receiver have the same pads, and don't lose them.

You just came on shift at the NSA one-time-pad decryption office, and whoever was there before you has really made a mess of the place. A message has just come in needing to be decrypted, and the one-time pads you're supposed to use are all over the place -on the floor, in the trash, stuck in the ceiling (how did <u>that</u> happen?), just everywhere. After working at cleaning up the office for a bit, you think you think you have a handle on the situation: You're pretty sure that the pad that decrypts the message is one of two possible candidates. You're also pretty sure that since the messages coming in consist of government communications, you can guess that certain words appear someplace in the plaintext of the message you received, though you don't know exactly where. Now the only question is: can you determine which pad is correct?

Input

There will be several input instances. Each will begin with a number n (< 100), the number of characters in each pad (and in the message). A value of n = 0 denotes end of input. Then there will be 3 lines of n characters each, the two one-time pads, and then the ciphertext message.

Next will be a number m (≤n) and the next line will contain an m character message that you think (hope?) will appear somewhere in the decrypted ciphertext.

Output:

For each input case i, output either:

```
Case i: Pad 1
Case i: Pad 2
Case i: Both
```
Or
```
Case i: Neither
```
..depending on which one-time pad is a possible candidate for the message.

Sample Input:

```
5
ABCDE
ZZYZY
JFNOC
3
LLY
10
ABCDEFGHIJ
DALKHBCOUD
SDFKLJDSHJ
3
XYZ
6
XXXXXX
YYYYYY
DEFCDE
3
FGH
0
```

Sample Output:

```
Case 1: Pad 1
Case 2: Neither
Case 3: Both
```

Problem D: Bouncing the Evil Eye

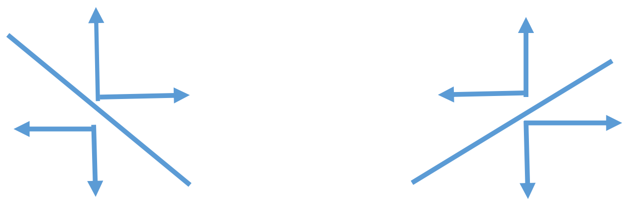
Working as a minion isn't all it's cracked up to be. Every so often a hero comes into your territory, and you hear the booming voice of the Overlord - "MINION! MOVE THE MIRRORS!" And then you have to trudge out and move the mirrors so that the Overload's Death Laser can shoot from his Evil Eye, and bounce around your land and incinerate the hero.

But last night you were up late playing with some ring that fell into your lair, and when you hear that voice ("MINION! MOVE THE MIRRORS!"), you're just not feeling it. You don't want to be incinerated yourself (the Overlord has a temper), so you've decided that you'll move just one mirror, in one direction. If that's not enough, well, I guess those plucky heroes might survive today.

Input:

There will be several input instances. Each instance will begin with two positive integers, r and c (≤20), the dimensions of your territory (in rows and columns). A value of r=c=0 will denote end of input. Otherwise, there will be r rows of c characters. Each character will either be:

`'–'`: an empty space
`'*'`: a wall. Walls block lasers, and mirrors can only move through clear spaces.
`'I'`: The evil eye (get it?). Exactly one character will be the evil eye. If the laser somehow bounces back to the evil eye, it will be blocked.
`'/'` or `'\'`: Mirrors, oriented in one of two possible directions. Mirrors cannot be reoriented (but they can be moved)
'`H`': The hero. Exactly one character will be a hero.
A laser can be shot from the evil eye in any of the four cardinal directions. The laser will bounce off mirrors in 90 degree directions, based on the incoming direction and the orientation of the mirror:



Output:

For each input instance i, output

```
Case i: Yes
```

.. if by moving at most one mirror in a straight line in one of the four cardinal directions we can shoot a laser from the evil eye in any of the four cardinal directions to hit the hero. Lasers can only go through empty space or bounce off of mirrors.

Otherwise, output:

```
Case i: No
```

Sample Input:

```
10 10
------H--\
----------
----I***--
----------
--\------/
----------
----------
----------
----------
----------
5 7
-------
-------
I-----H
*******
/\/\/\/
8 8
--------
-******H
-I\---*-
------\-
--------
--------
--------
--------
0 0
```

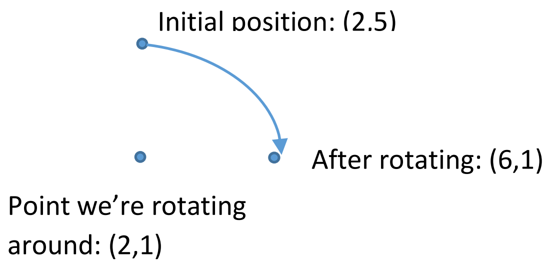
Sample Output

```
Case 1: Yes
Case 2: Yes
Case 3: No
```

Problem E: The Incredible Moving Point

In high school math, you probably learned about transformations of points and shapes in the plane. The kinds of transformations we'll consider are:

- Translation, which moves a point a fixed distance in both the x and y directions
- Reflection, where a point is "flipped" across the x axis, y axis, the line y=x or the line y=-x
- Rotation, where a point is turned clockwise 90 degrees relative to a point. For our purposes, the point we rotate around will always be directly below the point we're moving. So, for example:



In this problem, you're given a starting point and a sequence of transformations, and your job is to figure out where the point will end up. Notice that if the point begins at integer coordinates, each of the above transformations will keep the point at integer coordinates.

Input:

The input will begin with a positive integer n (the number of input instances). Each input instance will begin with a line containing two integers, x and y (-10000 ≤ x,y ≤10000). This will be the starting position of the point. Then there will be a positive integer k (≤100), which denotes the number of transformations. Then there will be k lines. Each line will be either:

- `T dx dy` (A translation by integer dx and dy. -10000 ≤dx, dy, ≤ 10000)
- `Ref t` (Reflection. T is either 1,2,3, or 4. 1 means flipping across the x axis, 2 means flipping across the y axis, 3 means flipping across y=x, 4 means flipping across y=-x)
- `Rot dy` (Rotate 90 degrees clockwise around a point an integer dy below the point. 1≤ dy ≤ 10000)

Output:

For each input case i, output:

```
Case i: (fx,fy)
```

..where (fx,fy) are the final x and y coordinates of the point after going through the series of transformations given above.

Sample Input:

```
2
0 0
5
T 2 2
Ref 2
Ref 4
Rot 5
T -3 4
99 -90
9
Rot 5
Rot 5
Rot 5
Rot 5
Rot 5
Ref 1
Ref 2
Ref 3
Ref 4
```

Sample Output:

```
Case 1: (0,1)
Case 2: (124,-115)
```

Problem F: There are more than 10 kinds of people

You've may have heard the "awesome" joke: "There are 10 kinds of people-- those who know binary and those who don't." The joke depends on the idea that the sequence of digits "10" has different meanings depending on whether you interpret it in base ten (as "ten") or base two (as "two").

An interesting question is whether you can pull this trick in other bases. For example, the number 88 in base 10 will show up as a bunch of different digit combinations in a bunch of different bases:

- 1011000 in base 2
- 10021 in base 3
- 1120 in base 4
- 323 in base 5

  … and so on

But, some numbers are not equivalent, no matter what base you use. For example, 40 in base 10 will not be equal to 6 in any base.

There are two special cases to consider:

- Unary (base 1) doesn't follow the standard "positions in the number are increasing powers of the base" rule, so we'll ignore it.
- Bases larger than 10 use symbols to denote a value larger than 10 in a single position. Obviously, any number in these bases can only use digits 0-9 to match a base 10 number.

Input:

There will be several input instances. Each will begin with two numbers n m. (≤10,000). A value of n=m=0 will denote end of input. Otherwise, n is to be interpreted as a base 10 number, and m is a number we'd like to be equivalent to n.

Output:

For each input case i, output either:

```
Case i: b
```

.. where b is the smallest base (of at least 2) where n in base 10 is equivalent to m in base b.

If no such base exists, output:

```
Case i: Not possible
```

Sample Input:

```
88 323
2 10
40 6
291 123
99 99
0 0
```

Sample Output:

```
Case 1: 5
Case 2: 2
Case 3: Not possible
Case 4: 16
Case 5: 10
```