Thirteenth Annual Ohio Wesleyan University Programming Contest

April 6, 2019

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++, Python2 and 3, 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: Be a Good Bee

Note that this question has a time limit of 20 seconds, instead of the usual 10.

Two bees (Bee A and Bee B) are looking at a row of flowers, dripping with sweet nectar to be harvested. The bees want to work together to harvest the flowers. The trouble is that each bee has different nectar harvesting skills, and so each bee takes a different amount of time at each flower. The bees are wondering what would be the right way to divide up the harvesting work so that they can both be done in the shortest amount of time. Can you help them decide what that time would bee (er.. be)?

Input: Each input instance begins with a positive integer n (<=30), the number of flowers. A value of n=0 denotes end of input.

There will then follow n pairs of integers, one pair per line, corresponding to each flower. The first number of each pair is how long Bee A would take harvesting nectar from that flower, the second number is how long Bee B would take harvesting nectar from that flower. Both times will be <= 10 for each flower.

Output: For each input instance I, output:

```
Case i: t
```

..where t is the smallest amount of time it can take for both bees to harvest the flowers. A flower can only be harvested by one bee, each bee can only harvest one flower at a time (but both bees can work on different flowers at the same time), and the overall finish time is finish time of the later of the finish times of the two bees. You can assume travel between flowers is instantaneous.

Sample Input:

```
3
1 2
2 3
4 1
3
1 2
1 2
1 2
3
5 4
6 8
2 9
0
```

Sample Output:

```
Case 1: 3
Case 2: 2
Case 3: 8
```

Problem B: Escape the Maze of Doom!

Boy, last night was intense. You actually don't remember what you were doing, and how you got…

"WELCOME TO THE MAZE OF DOOM!!!"

...huh?

"THE MAZE OF DOOOOOOOMMM!!!"

Well, that's different.

"DOOOOOOOOOOOOMMMMMMMMMM!!!!"

…and annoying. You look around and see that you're staring outside some kind of rectilinear maze. As you walk around and inspect it, you realize that most rooms of the maze are built to force you to exit in a specific direction based on where you came in from (So the exits are one-way and you usually won't be able to re-enter the room you came from). You need to reach the exit room to escape. You'd like to do it as quickly as possible because your head is throbbing, and you could use a glass of water. And the loud voice yelling "DOOOM!!" over and over isn't helping.

Input: Each input instance will begin with two integers, r and c (r,c <= 20), the number of rows and columns in the grid. A value of r=c=0 denotes end of input.

There will then be r rows of c columns of characters. Each character will be either:

- '!' This is the goal room. This will occur exactly once in each maze.
- 'L' 'R' 'S' 'U': This means that the exit of the room is made by turning 'L'eft, 'R'ight, moving 'S'traight, or doing a 'U'-turn. Notice that since these are based on the direction you enter the room, coming at the same room from different directions will exit you into different rooms.
- '*': This means you can exit the room in any direction.
- 'X': These are the rooms where you fall into the acid pit of DOOOMMM!!!, which contains laser acid-sharks of DOOOMMMM!!! You should avoid those rooms, since you'll probably meet your DOOOOMMMM!!!

Output:

For each input instance I, output:

```
Case i: n
```

..where n is the smallest number of moves needed to reach the goal room. You can enter the maze from the perimeter at any location (walking straight in from the outside). You then must follow the exit rules for the room you just entered. This means, for example, that if your first move enters a 'U' square, you will immediately turn around and exit the maze (and fail).

If no paths exist to the goal, output:

```
Case i: Impossible
```

Sample Input:

```
4 3
U U U
R R U
S ! U
R L U
4 6
X X X X X X
X S U S ! X
U * * * L X
X X X * * X
3 3
S U S
U ! U
S U S
0 0
```

Sample Output:

```
Case 1: 2
Case 2: 4
Case 3: Impossible
```

Problem C: Fighting Monsters Game

Little Joey likes to play with toy monsters. He has invented a game for how they fight:

- Each monster has a "defense" value. The attacking monster needs to roll that number or higher on a die to hit his opponent.
- If (and only if) the first roll was successful, both players roll two dice. The attacker's roll is their "damage." The defender's roll is their "block." The amount by which the "damage" exceeds the "block" is subtracted from the defending monster's life total. (If the block is greater than or equal to the damage, the attack is blocked and there is no change to the life total).

Little Joey's dad has noticed that when he plays the Fighting Monsters Game with Joey, it can take an awfully long time to kill a monster. One way of estimating the time it will take is by computing the "expected damage" of each attack.

For example (which may or may not be taken from a real life experience), if the first roll against the defense needed to be a 13 or higher on a 20-sided die, and then both damage and block rolls were two 4-sided dice, there is a 40% chance to hit, and when a hit happens, there are 4^4 = 256 ways for the 4 dice to come out. Out of those, 40 of them do 1 damage (there are two ways for the attacker to roll a 3 vs the defender's 2, there are 6 ways for the attacker to roll a 4 vs the defender's 3, and so on), 31 of them do 2 damage, 20 of them do 3 damage, 10 of them do 4 damage, 4 do 5 damage and just 1 possibility (attacker rolls 8, defender rolls 2) does 6 damage. The remaining 144 do 0 damage. So, if there is a hit, we expect to do:

$0 * \frac{144}{256} + 1 * \frac{40}{256} + 2 * \frac{31}{256} + 3 * \frac{20}{256} + 4 * \frac{10}{256} + 5 * \frac{4}{256} + 6 * \frac{1}{256} \approx .89$ damage per hit. Since we only hit 40% of the time, that means on average we reduce the defending monster's life by ≈.36 life per turn. If a monster starts with 100 life, that would mean an average of ≈ 280.7 (so round down to 280) turns. Ouch.

Little Joey's dad would like to know how much time he's in for, given a setup of dice and monster life. Can you help him?

Input:

Each input instance will begin with three integers: A V D and L. (A <= 50, V <= A, D <= 50, L <= 500). Values of A=V=D=L=0 denote end of input. Otherwise, A is the number of sides on the attacking die. V is the value the attacking die needs to reach to hit. D is the number of sides on the each of the pairs of damage and block dice, and L is the monster's life.

Output:

For each input case I, output:

```
Case i: T
```

Where T is the number of turns the game is expected to take, rounding down any decimal parts.

Sample Input:

```
20 13 4 100
20 13 6 100
20 1 10 100
0 0 0 0
```

Sample Output:

```
Case 1: 280
Case 2: 182
Case 3: 43
```

Problem D: Happy and Sad Numbers

Suppose we take a number like 19, and square each digit, and add them. $1^2+9^2 = 82$. Then we repeat the process: $8^2 + 2^2 = 68$, then $6^2 + 8^2 = 100$, and finally $1^2 + 0^2 + 0^2 = 1$. If a sequence of operations gets us to 1, we call the original number a *happy* number. It turns out that if a number is not happy, the process will always take us into an infinite loop where we repeat a number on the process multiple times. For example, if we start with 85, we get the numbers 89,145,42,20,4,16,37,58, and then 89 again, forming a loop. Numbers that cause kind of loop are called *sad* numbers.

Your job is to find out whether a number is happy or sad.

Input:

Each input case will begin with a positive integer n (<= 1,000,000,000). A value of n=0 denotes end of input.

Output:

For each input case I, output either:

```
Case i: Happy
```

Or

```
Case i: Sad
```

..as appropriate.

Sample Input:

```
19
84
123
1000
0
```

Sample Output:

```
Case 1: Happy
Case 2: Sad
Case 3: Sad
Case 4: Happy
```

Problem E: Not a Dumb Waste of Time

The Very Important & Not a Dumb Waste of Time Committee (The "VI&NDWTC") of your school is meeting today. The participants are all seated around a round table. The positions are numbered starting from 0 and increase going clockwise. (Deciding to start the numbering from 0 instead of 1 was the purpose of last week's VI&NDWTC meeting). As it turns out, each participant of the meeting has a preferred seat at the table, but not everyone was able to get their preferred seat at the start of the meeting. But, over time, various committee members get called out of the room (to take phone calls, meet with students, and somehow find other things to do with their time that inexplicably are more important than the VI&NDWTC meeting). When they come back into the room, they again try to sit in their preferred seat. If that seat is taken, they will move around the table clockwise until they find an open seat and sit there. People that are *in* the room never change seats while they are in the room. That would be weird.

Your job is to sit on the side (not at the table) as an observer for this meeting. With all of these comings and goings, it's hard to keep track of who is sitting where after a while. Perhaps a program can help?

Input:

There will be several input instances. Each instance will begin with two integers n (the number of people at the committee) and m (the number of seats at the table) (n <= 26, m <= 30, n <= m). A value of n=m=0 denotes end of input. Seats at the table are numbered clockwise in circular order, starting from 0. The seat numbered clockwise after seat m -1 is seat 0.

The second line will be a string of m characters denoting the initial seating positions of everyone on the committee. Committee members will be represented by the first n capital letters, and empty spaces will be represented by dots ('.')

The third line will be a string of n integers, separated by spaces, representing each person's preferred seating position. The first integer will refer to member 'A', the second to member 'B', and so on.

The fourth line will be an integer e (<= 50), the number of arrivals and departures from the room. On the next line will be e pairs of characters, representing the sequence of entrances and exits. These represent the order in which people enter and exit the meeting room.

The first character of each pair will be either a '+' (someone is entering the room) or '-' (someone is leaving the room). The second character of the pair will be the letter of a committee member. Each movement will always refer to a letter in the initial seating string. Every '+' move will refer to someone out of the room, and every '-' move will refer to someone currently in the room.

Output:

For each input instance I, output:

```
Case i: <seating>
```

Where <seating> is the representation of where everyone currently in the room is sitting at the end of the sequence of moves, starting from 0, using the '.' character to represent empty seats.

Sample Input:

```
4 5
AB.CD
1 1 1 1
5
-A −B −C +C +A
5 5
ABCDE
0 0 0 0 0
12
-A −B −C −D −E +B +A +E +D −B +C +B
0 0
```

Sample Output:

```
Case 1: .CA.D
Case 2: CAEDB
```

Problem F: Redundant Prerequisite

Congratulations! The VI&NDWTC from Problem E has elected you chair of your department! Your first task is to look at the courses in the major that your department offers and think about their prerequisite structures. It turns out as courses and requirements have changed over the years, some prerequisites have been made redundant.

For example, suppose you have three courses: CS300, CS200, and CS100. CS200 requires CS100 as a prerequisite. CS300 requires both CS200 and CS100 as prerequisites. The prerequisite of CS100 for CS300 is redundant, because anyone who wants to take CS300 must also have taken CS200, and to take CS200, you need to have taken CS100.

More formally, a prerequisite link is redundant if and only if any legal sequence of courses a student can take with the link is also a legal sequence of courses a student can take without the link.

The first step is getting a handle on the scope of the problem by figuring out just how many redundant links there are.

Input:

Each case will begin with two integers c (the number of course) and p (the number of prerequisite links) on the first line (c <= 50, p <= 1225) A value of c=p=0 denotes end of input. The second line will consist of c strings (the names of the courses). There will then follow p lines, each with two strings. The first of the strings is the prerequisite, the second is the course. These courses will always have been listed in the second line of the problem description, and there will never be a circular chain of prerequisite links.

Output:

For each input instance I, output:

```
Case I: L
```

..where L is the number of redundant links. A redundant link should be considered in isolation from all other redundant links (in other words, we're not thinking about combinations of link removals- we're counting the number of single links we can separately remove from the original collection of links).

Sample Input:

```
3 3
CS100 CS200 CS300
CS100 CS200
CS100 CS300
CS200 CS300
5 8
Phil100 Phil200 Phil250 Phil300 Phil499
Phil100 Phil200
Phil100 Phil250
Phil100 Phil499
Phil200 Phil300
Phil250 Phil300
```

```
Phil250 Phil499
Phil300 Phil499
Phil100 Phil300
0 0
```

Sample Output:

```
Case 1: 1
Case 2: 3
```