Eleventh Annual Ohio Wesleyan University Programming Contest

April 1, 2017

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: Broadcast Relay

Suppose we have a network and we are trying to send a message from one node to all others. The nodes don't know the direct path to each other, so the source node broadcasts the message to all adjacent nodes. That node will broadcast to all adjacent nodes, and so on until all nodes get the message.

The problem is that some nodes may be isolated from the sender, and never receive the message. Your job is to analyze the network and determine the minimum number of connections needed to have a signal reach all nodes.

Input: Each input instance begins with an integer n (≤50), the number of nodes in the system. Nodes are numbered sequentially from 1 to n. A value of n=0 denotes end of input. Otherwise, there will then follow n lines of the form m a_1 .. $a_{m.}$ The first line states the number of neighbors of node 1, and then lists its direct neighbors by number. The second line will hold the neighbors of node 2, and so on.

The next line will be an integer s between 1 and n, the source of the message.

Connections are possibly, but not necessarily, bidirectional. That is, a being a neighbor of b does not necessarily mean that b is a neighbor of a.

Output: For each input case i, output the following:

```
Case i: c
```

Where c is the minimum number of connections needed to be added to the network before a message from s can be received by all nodes.

Sample Input:

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

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

Sample Output:

```
Case 1: 2
Case 2: 3
Case 3: 0
```

Problem B: Lost in the Woods

“Don’t go hiking alone!” they said. Bah- you have a compass, you’ll be fine!

“Don’t go off the trail!” they cried. Good advice, but then you saw this really beautiful bird, and followed it for a while. Now you’re not quite sure where the trail is. Now it’s getting late, and you’re starting to worry.

Then you remembered the advice of Bud, your Computer Science Buddy. Bud had a feeling that something like this might happen, and gave you an algorithm to follow for this exact situation:

- Mark your current location as the “starting point.” You’ll probably be seeing a lot of it.
- Imagine yourself at the center of a circle of radius d. Break the circle into k equally sized chunks.
- Face straight north, and start walking for distance d. If you find the trail, great. If not, go back to the starting point.
- Turn clockwise to face the next “break” in the circle, and walk the same fixed distance. If you find the trail, great, if not, go back to the starting point.
- Repeat this process until you either find the trail or have gone in a complete circle (meaning you have taken k trips from the start). If you have not found the trail after a complete circle, start over again (walking straight north) but going twice the distance as your previous circuit. If necessary, your third time around the circle will be distance 3d, your fourth time around the circle will be a distance 4d, and so on.

Notice that no matter how many pieces you break the circle into (as long as it’s at least 3), or how far your fixed distance is (as long as it’s more than 0), you will always find a straight line trail with this algorithm. The only question is how many trips it will take from the starting point- that’s the question you’re going to answer.

Input:

The input will begin with an integer n, denoting that there are n cases to test. Each case will begin with a line containing the coordinates of two points: x1 y1 x2 y2. These points define the line of the trail. The trail is an infinitely long straight line that goes through those two points. All coordinates will be integers between -1000 and 1000

The next line of input will contain two integers: k ($3 \le k \le 100$), the number of equally-spaced walks you would do in one trip around the circle, and d ($1 \le d \le 1000$), the distance from the starting point you will go on your first trip.

The starting point will always be the origin (0,0). The trail will never intersect the origin.

Output:

For each input case i, output the line:

```
Case i: t
```

Where “t” is the number of trips from your starting place it will take you to find the trail.

Sample Input:

```
3
-10 10 10 10
4 1
4 2 3 1
3 5
0 2 2 2
50 1
```

Sample Output:

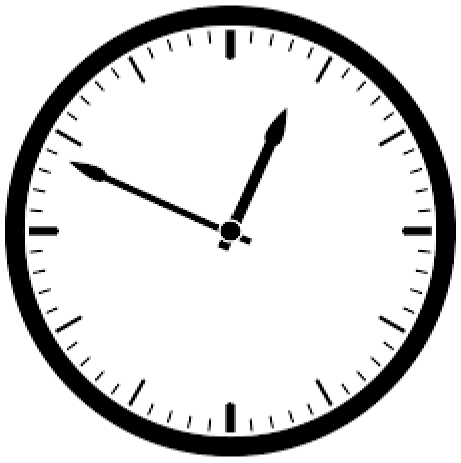
```
Case 1: 37
Case 2: 2
Case 3: 51
```

Problem C: Mirrored Clocks

Imagine an analog clock (the kind with hands). Most people are familiar with these kinds of clocks, and practiced at reading them- so much so, that many clocks don't even have numbers:

(11:11)

Now, imagine seeing the same clock through a mirror. The clock looks the same, but the hour and minute hands are flipped across the vertical line going from 12 to 6:



(12:49)

We won't worry about the second hand, but your task is to figure out what time an analog clock looks like if we look at it in a mirror.

Input:

The first line of input will be an integer n, the number of input cases. Each input case will be one line consisting of a time in the form h:mm (for hours 1-9) or hh:mm (for hours 10-12). All times will be legal times on a 12 hour clock (hours between 1 and 12, minutes between 0 and 59)

Output:

For each input case i, output

```
Case i: Time
```

Where “time” is a time (either in h:mm or hh:mm, as appropriate) corresponding to what the input time would look like if viewed through a mirror on an analog clock

Sample Input:

```
5
11:11
9:00
12:30
1:27
6:00
```

Sample Output:

```
Case 1: 12:49
Case 2: 3:00
Case 3: 11:30
Case 4: 10:33
Case 5: 6:00
```

Problem D: Match 3 Cascade

A common type of game that is often seen on smartphones is called “Match-3”. In this kind of game, players are presented with a grid of colored pieces. The player can swap two adjacent pieces (horizontally or vertically, not diagonally). If this results in one or more vertical or horizontal rows of 3 or more pieces of the same color, all pieces in all of those rows are removed simultaneously, and new pieces fall vertically into the empty spaces.

It is possible that when these new pieces fall, one or more new rows of 3 or more pieces of the same color will get created, and those pieces will be removed (again, simultaneously) as well. This process continues until the board results in a state where no rows of columns of 3 or more pieces of the same color exist. It is possible to create with one move a cascade that removes a large amount of pieces. Your task is to find the move that removes the most pieces in a given board.

In these games, as pieces are removed, the board is filled with random pieces (that come down from the top). For our purposes, we will assume that these new pieces will never match with each other or with any piece already on the board (so will never be removed in a cascade match).

Input:

Each input instance will begin with to integers m and n ($1 \leq m,n \leq 20$). A value of m=n=0 denotes end of input.

There will then follow an m rows of n strings containing capital letters. Each letter represents a different color. There will be no horizontal or vertical chains of 3 or more of the same letter, and there will be at least one pair of horizontally or vertically adjacent letters that when swapped will result in a chain of 3 or more adjacent letters.

Output:

For each input instance i, output the following:

```
Case i: k
```

Where k is the maximum pieces removed by any swap of pieces on the board.

Sample Input:

```
3 5
ABCDE
ABCDE
BAFGH
5 5
ABCDE
FGHGG
FAGCC
BFGDD
ABCDE
8 8
ROGOOPOY
BOOSYRYS
RGGRGYGR
PBRBOGSS
RSGGYBBY
RSYROYYG
GOSGYBOY
OYRPGBBR
0
```

Sample Output:

```
Case 1: 6
Case 2: 5
Case 3: 6
```

Problem E: One-pass War

As a child you may have played the card game "War". Its most memorable feature was that the games could go on indefinitely.

Apparently kids at school today are playing a streamlined version of the game. Here are the rules:

- Both players get exactly half of the cards in the deck
- Each round, players choose one card from their deck. The higher ranked card wins both cards, and places them in a "score pile". If the two cards are of equal rank, both cards are discarded and not scored for either player.
- The deck is only gone through once, after which all cards are in one of the two score piles, or discarded. At that point, the game is completed, and the cards in the score piles are counted, with the higher total winning.

Not only is this game faster than the "classic" version of War (where score piles are reclaimed to form a new deck and the game only ends when all of the cards for one player have run out), the option to choose a card instead of randomly picking one is interesting, and potentially adds a dimension of strategy to the game. Your task is to take two hands of cards and determine what the maximum possible scores for each player are.

Input:

Each input instance begins with an integer n: (≤ 100) the number of cards in both hands. A value of n=0 will denote end of input. There will then follow two lines of n positive integers, denoting the ranks of the cards. Higher numbers beat lower numbers, and all cards will be integers (cards like Jacks, Queens, and Kings will be represented as numbers).

Note that it is *not* necessarily the case that there will be 52 cards total, or that the distribution or values of cards correspond to a real deck. I've seen many instances of card games (especially with young children) where a "deck" of cards contains a random mishmash of cards from lots of different places.

Output:

For each input instance i, output:

```
Case i: a b
```

Where a is the highest possible score the first player can achieve, and b is the highest possible score the second player can achieve.

Sample Input:

```
5
1 2 3 4 5
5 3 2 1 4
6
12 8 10 4 6 2
11 9 7 5 3 1
```

```
7
100 90 80 70 60 50 40
10 9 8 7 6 5 4
0
```

Sample Output:

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

Problem F: Unique Donations

As the newest worker for your favorite local charity, they assigned you to work Unique Circle. The houses in Unique Circle are arranged in a circle (go figure), and are all on the same side of the street, but the people there pride themselves on their uniqueness so much that they don't want donate to you if they see one of their neighbors do so. Specifically, if the residents of one house donate money to your cause, neither of their neighbors will.

Luckily, you are armed with the latest in donation projections- you have an estimate of how much each house would potentially donate if you asked them to. Using this information, you can compute the maximum possible total donations you can receive, given the peculiar nature of the people on this block.

Input:

Each input instance will begin with an integer n(≤100), the number of houses on the block. A value of n=0 denotes end of input.

There will then follow a line with n positive integers: The donation projections of each of the n houses.

Since the block is a circle, the first and last houses in the list are considered adjacent.

Output:

For each input case I, output the line:

```
Case i: k
```

..where k is the maximum amount of donations you can gather from this block.

Sample Input:

```
5
5 1 2 3 4
6
5 4 1 4 1 4
10
9 1 2 3 4 5 6 7 1 8
0
```

Sample Output:

```
Case 1: 8
Case 2: 12
Case 3: 24
```