Twelfth Annual Ohio Wesleyan University Programming Contest

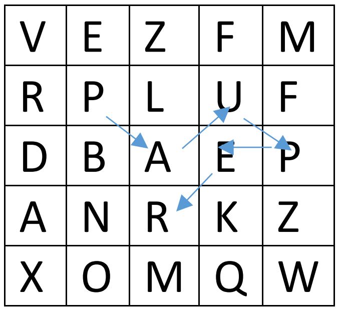
April 7, 2018

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: Boggle Verifier

The board game Boggle consists of a square board of letters. Players try to form words out of adjacent letters. Here is an example 5x5 board, with the letters used to form the word “PAUPER” given:



Words can start anywhere, and can be formed by connecting letters adjacent in any direction, including diagonally. Words can not use the same letter in the same location more than once. (So, for example, the word “FLUFF” is not able to be spelled in the above board, because you would need to use one of the two F’s in the board twice).

Your task is to take a sample board, and a set of sample words, and determine which ones can be spelled on the board or not.

Input: There will be several input instances. Each input instance will begin with a positive integer n (<= 8) denoting the size of the board. A value of n=0 denotes end of input. There will then follow N lines of N capital letters separated by spaces.

There will then be an integer m (m <= 100) denoting the number of words to verify on each board. There will follow m strings of, one per line. Each string will be 7 characters or less. These are the words to check on the board.

Output: For each input instance i, output:

```
Board i
```

Then for each word j on that board, output:

```
Word j: yes
```

… if it is possible to spell the word on the board, or

```
Word j: no
```

...if it is not.

Sample input:

```
5
V E Z F M
R P L U F
D B A E P
A N R K Z
X O M Q W
4
PAUPER
FLUFF
BRAKE
QUIZ
4
P O W A
E S I T
R S L U
E I N T
4
POSERS
SILENT
SPORE
SLIT
0
```

Sample Output:

```
Board 1
Word 1: yes
Word 2: no
Word 3: yes
Word 4: no
Board 2
Word 1: yes
Word 2: no
Word 3: no
Word 4: yes
```

Problem B: Fishing For Change

The other day, I was at the drive-through and was digging in my pockets trying to get exact change to pay for my food. It occurred to me that as time passed, I was much less likely to have certain kinds of coins (like dimes, which I will receive at most 2 of in any transaction) than others (such as pennies, which I can receive up to 4 in any transaction). This caused me to think about how to extend this idea to other systems of coins-- which coins would be most likely to accumulate if the denominations of the coins were different?

Input:

Each input instance will begin with an integer d-- the number of denominations of coins (1 <=d <= 100). There will then be d distinct integers in decreasing order. The last integer will always be a 1. A value of d=0 will denote end of input.

Otherwise, there will follow a single positive integer n (1 <=n <=10000). This denotes that we are considering change from 1 cent all the way up to (and including) n cents. For each amount of change, we expect the cashier to give coins by going through the denominations of coins in decreasing order, and for each denomination, giving as many coins as possible. This may lead to not giving not fewest possible coins for some denominations, but this is how cashiers work.

Output:

For each input case i, output:

```
Case i: d1 d2 d3..dd
```

Which represent the denominations of coins in decreasing order of frequency that they are returned across all possible change amounts. If there is a tie, output the smaller denomination first.

Sample Input:

```
4
25 10 5 1
100
3
5 3 1
20
2
3 1
8
0
```

Sample Output:

```
Case 1: 1 25 10 5
Case 2: 5 1 3
Case 3: 1 3
```

Problem C: Friendship Paradox

The Friendship Paradox is a phenomon that occurs in friendship networks. It can be expressed as follows: Most people have fewer friends than most of their friends do. This has been seen in real social media networks like Facebook (most people's friends have more friends than they do) or Twitter (most people have fewer followers than their followers have), and even in academic paper authorship (most people co-author fewer papers than their co-authors do). This tends to arise because most of these kinds of networks have many people with a few connections and a relatively small number of people with many connections

In this problem, you'll look at some sample networks and determine the extent to which this paradox holds.

Input:
There will be several input instances. Each instance will start with two integers n, the number of people to consider, and f, the number of friendship connections. (1<= n <= 100, 0<=f <= n*(n-1)/2). Values of n = f = 0 denote end of input. There will then follow f lines. Each line will consist of two different numbers between 1 and n, meaning that the people with those numbers are friends. The relationship is symmetric. That is, if person a is a friend of person b, then person b is a friend of person a.

Output:
For each input instance i, output:
```
Case i: k
```

Where k is the number of people who have at least one friend, and who have less friends than at least half of their friends (rounded down) do.

Sample Input:
```
6 11
3 1
3 2
3 4
3 5
3 6
1 2
1 5
1 4
2 6
4 6
5 6
5 4
1 2
```

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

Sample output:

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

Problem D: SSS

As the president of "Sexy Socks for Snakes," Sybil Hiss likes to sew long, straight "snake socks" made out of several connected pieces of fabric. Unfortunately, the fabric store sells fabric in preset lengths, which rarely are the same length that Miss Hiss needs for her sexy sock. So, Sybil has to form her sock by sewing together different pieces of different lengths together. Sewing two pieces together takes time, which Sybil never has enough of. In fact, she's willing to make some of her socks a bit too long (but never too short) if it means she can minimize the number of sewing passes she can make.

Input: There will be several input cases. Each case begins with a positive number n (n <= 10000), the length in inches of the completed sock. A value of n=0 denotes end of input. Otherwise, the input will continue with a positive integer m (m <= 100) denoting the number of fabric lengths available. The completed sock can use as many of each length as desired, but can only use available lengths. There will follow m integers, denoting the lengths of each fabric. The input will conclude with an integer L (0 <= L <= n), the number of inches the final sock can be longer than n.

Output: For each input instance i, output:

```
Case i: k
```

..where "k" is the smallest number of pieces of fabric used to create a finished sock that is at least n inches long, and at most n+L inches long. This will always be possible with some number of fabric pieces.

Sample Input:

```
10
2
4 3
1
12
3
5 6 2
2
15
3
11 5 1
0
0
```

Sample Output:

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

Problem E: Stuffing My Wallet

People claim that I have way too many cards and things in my wallet. I don't know what they're talking about. You never know when you'll go back to that fast food place in Arizona and need to stamp your loyalty card.

What is hard, though, is finding things in my wallet. Every so often I organize my cards, but then when I use the cards, I have to take them out and put them back; I probably don't put the card back in the correct place, and things get out of order. Lately, I have tried to be a bit more careful about putting the cards back in the wallet- each card gets replaced at either the front of the wallet (before all other cards), the back of the wallet (after all other cards), or the precise middle of the wallet (If the wallet has an odd number of cards, we replace the card after half of the cards rounded down). After a series of such removals and replacements, we can reconstruct the order of everything in my wallet. Will you do that for me?

Input: There will be several input instances. Each instance will begin with an integer n (n <= 100), the number of items in the wallet. A value of n=0 denotes end of input. Then there will follow n distinct strings, one per line, which are the contents of the wallet from front to back. Then there will be a number k (k <= 100). There will follow k lines. On each line will be a string s and a character c. s is the name of the item we are removing from the wallet (and will always be one of the n strings listed previously), and c will either be the letter 'F', 'B', or 'M', saying whether we put the item back in the Front, Back, or Middle of the wallet respectively.

Output: For each input instance I, output:

```
Wallet i:
```

Then the contents of the wallet after the k moves listed in the input, one item per line. Output a blank line after the last card in each wallet (including the last one)

Sample Input:

```
5
bookstore
creditcard
icecream
sandwich
school_id
3
creditcard F
icecream B
sandwich M
6
card_1
card_2
card_3
```

```
card_4
card_5
card_6
9
card_1 F
card_2 F
card_3 F
card_1 F
card_5 F
card_6 F
card_1 F
card_4 F
card_3 B
0
```

Sample_Output:

```
Wallet 1:
creditcard
bookstore
sandwich
school_id
icecream

Wallet 2:
card_4
card_1
card_6
card_5
card_2
card_3
```

Problem F: Territoriality

Farmer Bob and Farmer Wilson both think they own the same field. Over the course of years, they have tried to “stake their claim” on the field by building fences around it. As a result, some parts of the field are completely surrounded by one or the other sets of fences (or by the boundaries of the field), and some parts of the field are neutral (bounded by fences owned by both farmers).

As the land surveyor, you’ve been sent out to assess how much field is claimed by each farmer. Luckily, the field is broken into a grid of plots, making your life easier. An empty plot is claimed by a farmer if there is a rectilinear path through empty plots to at least one fence owned by that farmer, and no rectilinear paths through empty plots to fences owned by the other farmer.

Input:
There will be several input instances. Each instance begins with an integer r and c, the number of rows and columns (in plots) of the rectangular field. (1 <= r,c <= 20). A value of r = c = 0 denotes end of input. There will then follow r rows of c characters. Each character will either be ‘B’ (for a piece of fence placed by Farmer Bob), ‘W’ (for a piece of fence placed by Farmer Wilson), or ‘-‘ (for an empty space).

Output:
For each input instance I, output:
Case i: b w

Where b is the number of empty plots claimed by Farmer Bob, and w is the number of empty plots claimed by Farmer Wilson.

Sample Input:
```
6 6
- W - W - -
W W W W B B
- - - - B -
- - - - B B
- - - - B -
- - - B - -
7 7
W W W W - - -
- - - W - - -
W W W B B B -
- - - B - B -
W W W B B B -
- - - W - - -
W W W - - - -
```

```
7 8
W W W W - - - B
- - - W - - - B
W W W B B B - B
- - W B - B B B
W W W B B B - B
- - - W - - B W
W W W - - - W -

0 0
```

Sample output:

```
Case 1: 4 2
Case 2: 1 6
Case 3: 2 9
```