Fourth Annual Ohio Wesleyan University Programming Contest

March 27, 2010

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. You cannot 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 communication with the judges will be handled in the PC^2 environment.
8. Judges' decisions are final. No cheating will be tolerated.

Problem A: This game is Bonkers!

The classic board game Bonkers involves players rolling dice and moving along a board with spaces. The trick in this game is that the spaces can be "tagged" with cards, which instruct the player to move an additional number of spaces, forwards or backwards. If, after following the instructions on the card the player ends on a space with a new card, that card's instructions are followed. This process is repeated until the player ends up on a space with no cards attached to it.

One situation that can arise is that the cards give instructions that send the player into an infinite loop (where a player keeps going to the same spaces forever). There is actually a rule in the game that says if this happens, the player caught in a loop gains a point.

Your job is to write a program that, given a board configuration, determines if an infinite loop exists.

Input:
There will be several input instances. Each input will begin with an integer, N (≤100), denoting the size of the board (in spaces). A value of N=0 will signify the end of the input.

If N > 0, there will then follow N integers on the next line. Each integer will represent the value of the card attached to the space- a positive number means to move forward, a negative number will mean to move backwards, and a 0 will mean that there is no card on the space. The board is circular, so it is possible for a card to send a player beyond the end of the board, either forwards or backwards (having the result of wrapping around the end of the board). Each card value will be in the range between –N and N (not inclusive)

Output:
For each input instance i, output the location of the first space on the board that is part of an infinite loop, or a message that no loop exists. Board positions start at 1.

Sample Input:
```
10
1 2 0 3 0 0 3 0 2 -1
8
0 0 3 0 0 -3 0 0
5
-4 -4 -4 -4 0
0
```
Sample Output:
```
Board 1: Loop begins at space 1.
Board 2: Loop begins at space 3.
Board 3: No loop found.
```

Problem B: Airline Seating

While many airlines assign you a seat when you get your ticket or boarding pass, some do "open seating" where people get onto the plane and sit where they want. An interesting social behavior pattern emerges on these flights- seats at the window and aisle fill up first, from the front of the plane, going towards the back, and the middle seats (between the window and aisle) fill up last (starting from the front). The exception to this is people who fly in groups, who will want to sit next to each other in a row.

Imagine that you're in line for an airplane. Looking at the people ahead of you in line, you want to know whether you'll be forced to sit in the middle or not.

The plane we're looking at has 3 seats in a row. Each row has a window seat, and aisle seat, and a middle seat. Each person that gets on the plane will have a preference:
"A"isle people will sit in an aisle seat in the closest row to the front of the plane. If all aisle seats are taken, they'll take the window seat in the closest row to the front of the plane. If all aisle and window seats are taken, they'll take a middle seat, as close to the front of the plane as possible.
"W"indow people will sit in a window seat in the closest row to the front of the plane. If all window seats are taken, they'll take the aisle seat in the closest row to the front of the plane. If all aisle and window seats are taken, they'll take a middle seat, as close to the front of the plane as possible.
"G"roup people will try to sit with their entire group in a contiguous line. They find the first row with the largest number of seats (not necessarily 3, if the group is late to get on the plane), and start filling that row, and consecutive rows behind it with their group members. Rows are filled window first, then middle, then aisle. You can assume that all groups will fit on the plane following this approach, and that all groups have at least 2 people.

Input:
There will be several input instances. Each case will begin with two integers: *r*, indicating the number of rows in the plane, and *n*, indicating the number of people ahead of you in line (you can assume $r \leq 20$, and $n < 3*r$). If $r = 0$, then we have reached the end of the input. Otherwise, there will then follow *n* characters denoting the passengers ahead of you. Each character will be either 'A', 'W', or 'G', denoting the passenger's preference. Any 'G' people will be in the same group as all 'G' people near them in line until the group is broken by a non-'G' character, or an end of the line.

Output:
For each input case, output either:
Non-middle seat found in row *k*.
or
Forced into middle seat in row *k*.
Either way, *k* represents the row that is closest to the front of the plane that you'll be sitting in (starting from 1).

Sample Input:

```
15 17
A W A W A W A W A W A W A W G G G
10 20
A A A A A A A A A A A A A A A A A A A A
9 20
G G G G G G G G G G G G G G G G G G G G
0 0
```

Sample output:

```
Non-middle seat found in row 9.
Forced into middle seat in row 1.
Non-middle seat found in row 7.
```

Problem C: Random Baby Crawling

Imagine a baby crawling in the XY plane. The plane is filled with toys, and each toy has a different "interest value" to the baby. For simplicity, we'll assume that the baby can see in all directions (including behind itself), but only out to a certain distance. The baby will start crawling to the most interesting toy within its range. It is possible that during the crawl, a different, more interesting toy enters the baby's visual range. If this happens, the baby will change course and go towards this more interesting toy. The baby will stop and look around after moving one foot (or after reaching the toy it is going towards), and if a more interesting toy is seen, change direction, going towards the new toy.

Eventually the baby will stop moving, having reached the most interesting toy within its range.

Input:

The input will start with an integer *c* denoting the number of test cases. Each of the *c* test cases will begin with an integer, *r*, denoting the baby's visual range, and an integer, *t* ≤20), denoting the number of toys in the test case. There will then follow *t* lines of values, with 2 integers on each line, signifying the toy's X and Y position (in feet, relative to the origin). Toys will be listed in increasing order of interest, and also non-decreasing order of Y direction. At least one toy will be within the baby's visual range at the start. The baby will start at the origin (position (0,0)).

Output:

For each test case *i*, output the line:

Case *i*: The baby is playing with toy *j*.

..where *j* is the number of the toy the baby ends up with (starting from 1)

Sample Input:

```
2
10 5
-1 4
1 5
-2 12
2 13
3 20
5 5
-1 3
1 4
5 8
-4 12
0 19
```

Sample Output:

Case 1: The baby is playing with toy 5.

Case 2: The baby is playing with toy 2.

Problem D: Oh, Craps.

The casino game craps is played by rolling two dice. On the initial roll:

- A roll of 7 or 11 automatically wins
- A roll of 2, 3, or 12 automatically loses
- Any other number becomes the "point number". There will now be one or more subsequent rolls. If the "point number" is rolled a second time before a 7 is rolled, the player wins. If a 7 is rolled first, the player loses.

One casino strategy to play craps is to bet on every die roll. If a roll makes a point (and not an instant win or loss), then a new bet is placed, and that new bet starts a new sequence, that runs simultaneously with the old one. It's possible to have several bets on the table at the same time, but eventually a 7 will be rolled, and remove all of them (any points waiting to be resolved are lost, and the "new" bet on that roll will win).

Suppose for this problem that all bets are made in the amount of $1. So, each win adds $1 to the player's bankroll, and each loss subtracts $1 from the player's bankroll. Given a sequence of rolls using this strategy, determine the change to the player's bankroll for the sequence. You can assume that the player never runs out of money.

Input:
There will be several input instances. The sequence will begin with an integer, *n*, denoting the number of sequences. There will then be *n* sequences, one per line. Each line will begin with an integer, *r*, (r ≤50) denoting the number of rolls. Each of the *r* rolls that follow will be an integer in the range 2-12. There may be more than one 7 in a sequence.

Output:
For each sequence ,i, (starting from 1), output either:
```
Sequence i: Win of $X.
Sequence i: Loss of $X.
```
or
```
Sequence i: Break even.
```
..as appropriate.

Sample Input:
```
3
9 7 7 11 2 3 6 4 6 7
12 2 3 4 5 6 7 8 9 10 11 12 7
8 4 5 6 5 5 5 2 7
```

Sample Output:
```
Sequence 1: Win of $2.
Sequence 2: Loss of $6.
Sequence 3: Break even.
```

Problem E: Run the Numbers

It's pretty complicated to evaluate random number generators. It's not enough to see if you get a good range of numbers, we also need to see if the numbers are spread out randomly- a random number generator that chooses numbers between 1 and 10, but generates the numbers from 1 to 10 in sequential order wouldn't satisfy our definition of "random".

One way to check if the spread of a sequence of numbers is random is to use what the statisticians call a "runs test". We're going to use a variation of the "Wald-Wolfowitz runs test" that works as follows:

Given a sequence of K integers that were all equally probable to be drawn (for example, from a die rolling simulator), let n be the number of integers greater than the median, and let m be the number of integers less than the median. (We'll assume that the size of our range is even, so there will be no integers exactly equal to the median). We then compute the number of runs of high and low values in the sequence. A run is a sequence of consecutive high (or low) values of maximal size. Each integer in the sequence is a member of exactly one run. We then compute the upper and lower bounds on what the number of runs "should" be, using the following formula:

$$Bound = \frac{2mn}{K} + 1 \pm 1.96 * \sqrt{\frac{2mn * (2mn - K)}{K^2(K-1)}}$$

(For the statistically minded, the 1.96 value allows 95% of normally distributed sequences to fall within our bounds).

Note that the use of the ± gives up both an upper and a lower bound. If the number of runs in our sequence falls between the two bounds, we conclude that the sequence is random (at least, according to this test). If it doesn't, we conclude that the sequence isn't random.

Input:
There will be several input sequences. Each sequence will begin with an integer k (≤100), representing the length of the sequence, and an integer r (≤100), representing the size of the range. The value of r will always be even. A value of k <= 0 will signify the end of the input.

If k > 0, there will then follow k integers within the range 1-r (inclusive) which form the sequence to be evaluated.

Output:
For each sequence I, output either:

```
Sequence i: Random
```

or

```
Sequence i: Not Random
```

..as appropriate.

Sample Input:

```
10 10 10 1 9 2 8 3 7 4 6 5
15 8 8 2 7 3 4 1 8 6 2 7 7 4 1 8 3
12 6 1 2 3 4 5 6 1 2 3 4 5 6
0
```

Sample Output:

```
Sequence 1: Not Random
Sequence 2: Random
Sequence 3: Not Random
```

Problem F: Wandering Around at Knight

You're probably familiar with word-search puzzles, where you're presented with a grid of letters and need to find a word by choosing adjacent letters. A variation on this is the *Knight-move* word search puzzle, where you're presented with a grid of letters and need to find a word in a configuration where adjacent letters in the word are reachable by moving like a knight in chess. (A knight in chess moves from its current spot by either two rows and one column, or two columns and one row, in any direction).

So, the knight at position "K" in the table below, can move to any of the spaces labeled "X"

	X		X	
X				X
		K		
X				X
	X		X	

Your job is to write a program that tries to find a word in a grid of letters where each letter is connected to the next by a knight move. Some specific rules for the puzzles:

- You can never "wrap around" any edge of the grid
- You can never use the same letter twice.

Input:

There will be several input instances. Each input will begin with values *r* and *c* (both ≤ 8), denoting the dimensions of the grid. Values of *r* or *c* of 0 signify the end of input. Otherwise, there will follow *r* rows of *c* characters, one row per line. Each character on the line will be separated by a space, and each character will be an upper-case letter.

After the grid, there will be a string of ≤ 8 upper-case letters, which is the word to look for in the grid.

Output:

For each input instance *i*, output either:

Grid i: Word found

or

Grid i: Word not found

..as appropriate.

Sample Input:

```
5 5
A B C D E
F T H I J
K O E N O
S Q R S T
U T N X Y
CONTEST
```

```
5 6
W A B W C E
F G O H I W
O P A C R X
J R K L M N
D W O A B Q
WORD
0 0
```

Sample Output:

```
Grid 1: Word found
Grid 2: Word not found
```