The University of Akron Programming Competition

Presented by
The University of Akron Computer Science Department
Association for Computing Machinery Student Chapter

April 22, 2010

Rules:

1. There are **seven** questions to be completed in **four hours**.

2. C, C++ and Java are the only languages available.

3. Data is read from Standard Input and output is sent to Standard Output. Do not prompt for input values in the code you submit to be judged.

4. All programs are submitted as source code only. Submitting compiled information (binary, .class) will result in a Compilation Error penalty.

5. Non-standard libraries cannot be used in your solutions. The Standard Template Library (STL) and C++ string libraries are allowed. The standard Java API is available, except for those packages that are deemed dangerous by contestant officials (e.g., that might generate a security violation).

6. The input to all problems will consist of multiple test cases unless otherwise noted. The input listed on the problem description is not intended to be a comprehensive input set. The input is guaranteed to adhere to the descriptions in each problem; you do not need to check for invalid input.

7. Use of personal electronics during the competition is cause for disqualification. Cell phones must be turned all the way off (not silent mode, not airplane mode, etc.). You may use any written notes, books, or reference materials you bring with you.

8. Programming style is not considered in this contest. You can code in any style or with any level of documentation your team requires.

9. All communication with the judges will be handled through PC^2. All judges' decisions are final.

Problem A: Circular Permutations

Given a portion of the lower-case English alphabet beginning with 'a', in the usual order, and a list of fixed letters, perform a single circular permutation of the non-fixed letters.

For example, given the portion of length 7 ("`abcdefg`"), fixed entries of 0, 2, 3, and 5 mean that letters 'a', 'c', 'd' and 'f' are fixed, and we cycle the remaining letters one position to the left, giving the permutation "`aecdgfb`".

Details of the Input
The input will begin with two integers, l and f, on a single line. The first integer, l ($1 \le l \le 26$), indicates the length of the alphabet used. The second integer, f ($0 \le f \le l$), is the number of letters whose position will be fixed. The following line will contain a list of f numbers ($0 \le f_i < l$) indicating the letters to be fixed in position.

Input values of $l = f = 0$ indicate the end of input and should not be processed.

Details of the Output
For each case, print the line

```
Alphabet <case number>: <permuted alphabet>
```

where the `case number` corresponds to the input case beginning with 1, and the `permuted alphabet` is the resulting string after applying the above algorithm.

Sample Input

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

Sample Output

```
Alphabet 1: aecdgfb
Alphabet 2: cbad
```

Problem B: *N*-Way Rock Paper Scissors

Rock Paper Scissors is typically a two-player game where each player picks one of the three objects (rock, paper, or scissors), and the players both reveal their choice at the same time, usually by counting to three and then revealing. The act of revealing your choice is sometimes called a "throw." If both players throw the same object, it is a tie and they play again until they throw different objects. The winner is then determined by these rules:

1. Rock beats scissors: The rock can smash the scissors.
2. Scissors beat paper: The scissors can cut the paper.
3. Paper beats rock: We have no idea why. Really.

More than two people can play Rock Paper Scissors at the same time, but things get slightly more complicated. Each player picks an object and they all throw at the same time (just like in two-player). Players may be eliminated on each throw, and the *remaining players* continue this process until only one remains. The rules for determining which players are eliminated on each throw depend on how many of the three objects are represented on that throw:

1. If only one of the three objects is represented then nobody is eliminated, just like a tie in the two-player version.
2. If exactly two of the three objects are represented, then one of those objects is stronger than the other object (according to the 3 rules above). All players throwing the *weaker* option are eliminated, regardless of which object had more people throwing it. For example, if there are 4 throwers and their throws are {rock, rock, rock, paper}, then the players throwing rock are eliminated.
3. If all three of the choices are represented and each has the same number of people throwing them, then nobody is eliminated (this is considered a tie). If all three choices are represented *unevenly*, the group or groups with the most throwers move forward. For example, if there are 5 throwers and their throws are {rock, rock, rock, paper, scissors}, then the players throwing paper and scissors are eliminated. If there are 5 throwers and their throws are {rock, rock, paper, paper, scissors}, then only the player throwing scissors is eliminated.

Play continues amongst the players who have not been eliminated until only one player remains. Your job is to determine the winner of several *n*-way Rock Paper Scissors games.

Details of the Input

Each test case will begin with a line containing positive integers $2 \le n \le 20$ and $1 \le m \le 50$. The value in n represents the number of players. We guarantee that a winner will be determined in at most m throws (but perhaps less). The following n lines contain the following information for each player:

$$\texttt{<name>}\ \texttt{<throw}_1\texttt{>}\ \texttt{<throw}_2\texttt{>}\ \ldots\ \texttt{<throw}_m\texttt{>}$$

The player's `name` is a non-empty string of at most 20 alphabetic characters, and each of the $\texttt{throw}_i$ values is a string from the set {"`rock`", "`paper`", "`scissors`"}. The list of m $\texttt{throw}_i$ values represent the order of objects that will be thrown by that player (even though they may be eliminated before the m^{th} throw).

Input values of $n = m = 0$ indicate the end of input and should not be processed.

Details of the Output
For each test case, output one line:

```
Round <case number> goes to <winner>!
```

where `case number` reflects which input case this output corresponds to, and `winner` is the name of the winner (the one player who is never eliminated according to the rules).

Sample Input

```
4 4
Lisa rock scissors paper rock
Johnny scissors scissors scissors scissors
Dave rock scissors paper paper
Tim rock scissors scissors paper
2 3
Dave rock paper rock
Tim rock paper scissors
0 0
```

Sample Output

```
Round 1 goes to Tim!
Round 2 goes to Dave!
```

Problem C: Collatz Conjecture

Consider the following function: $C(n) = \begin{cases} 3n+1, \text{ if } n \text{ is odd} \\ n/2, \text{ if } n \text{ is even} \end{cases}$

In 1937, Lothar Collatz conjectured that the sequence formed by repeatedly applying this function (taking the output of one step and using it as input in the next step) would eventually reach 1 for all positive starting values of *n*. Here are two examples of the sequence using 7 and 53 as input values:

7 → 22 → 11 → 34 → 17 → 52 → 26 → 13 → 40 → 20 → 10 → 5 → 16 → 8 → 4 → 2 → 1

53 → 160 → 80 → 40 → 20 → 10 → 5 → 16 → 8 → 4 → 2 → 1

The Collatz Conjecture has never been proven to reach 1 for all positive input, but it has been shown to reach 1 for numbers as large as 2,362,292,408,970,609,843,065 (much larger than the largest number that can be stored in a 32-bit integer – if you're interested in helping the distributed computing project dedicated to testing large numbers against the Collatz Conjecture, visit http://boinc.thesonntags.com/collatz/).

Given two input integers *m* and *n*, we want to find the sequences generated by repeatedly applying C(*m*) and C(*n*) and then find the First Common Value To Appear In Both Collatz Sequences. We call this the FCVTAIBCS(*m*, *n*). For the case above, FCVTAIBCS(7, 53) = 40 (as 40 is the First Value To Appear In Both Collatz Sequences).

Details of the Input
Each line of the input will contain two integers *m* and *n* representing the values to be used for creating the Collatz sequences that will be used for the FCVTAIBCS function. Both of these values will be positive integers, and no value in either Collatz sequence will exceed the maximum value of a signed 32-bit integer. Input values of *m* = *n* = 0 indicate the end of input and should not be processed.

Details of the Output
For each case, output

```
FCVTAIBCS(<m>,<n>)=<result>
```

where `m` and `n` are the values given in the input, and `result` is the calculated value of FCVTAIBCS(`m`, `n`). There are no spaces in the output.

Sample Input

```
7 53
0 0
```

Sample Output

```
FCVTAIBCS(7,53)=40
```

Problem D: Software Watermarking

Software watermarking is a technique for embedding identification information into a program that can later be extracted, usually to prove code has been stolen. As a simple example, we could embed the following into our code:

```
String watermark = "Copyright 2010 University of Akron";
```

The problem with this example is that it's easily found and altered by hackers, so we've come up with a sneakier approach. We'll construct our software watermarks by encoding the watermark message as a series of arithmetic operations. Consider the table:

	+	-	*	/	%
1	a	b	c	d	E
2	f	g	h	i	J
3	k	l	m	n	O
4	p	q	r	s	T
5	u	v	w	x	Y
6	z	1	2	3	4
7	5	6	7	8	9
8	0	_	.	-	&

For an example watermark of "test", 't', 'e', 's', and 't' translate to "4%", "1%", "4/" and "4%" respectively. Hence, we add the following to our source code:

```
int w = (4%(1%(4/(4%seed))));
```

The "seed" variable in the most deeply nested operation is simply any integer variable that's already defined in the current scope. By using a seed value instead of a constant int to fill in the last spot, we prevent the operations from being precomputed by an optimizing compiler. We'll also just hope that this stream of calculations never causes a divide or mod by 0 error. This is just a proof of concept implementation, after all!

For this problem, you'll be producing the watermark source code line using the table above. You will be given a watermark string to encode and a variable name to use as the seed.

Details of the Input

The input will begin with a positive integer *n* indicating the number of cases to follow. Each case appears on one line in the format:

```
<watermark> <seed variable name>
```

There is a space between the watermark and the seed variable name. The watermark will consist only of characters in the table above, and the seed variable name will consist of some combination of uppercase letters, lowercase letters and numbers (but will not begin with a number). Both the watermark and seed variable name will be between 1 and 50 characters long, inclusive.

Details of the Output

Each case will produce one line of output of the form

```
int w = <encoded watermark>;
```

There is one space between "`int`", "`w`", "`=`", and the encoded watermark. The encoded watermark must contain no spaces, and must have one left parenthesis character, '(', immediately preceding each integer literal in the expression, with the appropriate number of right parentheses characters, ')', at the end. There are no spaces between the final right parenthesis and the semicolon.

Sample Input

```
2
test seed
copyright_2010 contest
```

Sample Output

```
int w = (4%(1%(4/(4%seed))));
int w = (1*(3%(4+(5%(4*(2/(2-(2*(4%(8-(6*(8+(6-(8+contest))))))))))))));
```

Problem E: Wacky World

Wacky World sure is a crazy place! Just ask one of its residents, Walter Winters (his friends call him Wally). You see, Wacky World is a two dimensional world. In Wacky World, there are many different zones. Each zone can be drawn on a map as a grid of squares, with each square being indicated by a coordinate composed of a letter and a number, like A9 or B3 (see example below). The letter indicates the particular column and the number of the particular row that the visitor is in. Wally knows, and you should too, that Wacky World laws only allow the movement of an individual from one square to another in either a horizontal manner or a vertical manner. No diagonal moves are allowed!

But, there is one little wrinkle to Wacky World's geography. You see, when Wally gets to the edge of a zone, he can instantly teleport to the exact opposite side of the zone without moving an inch! Furthermore, if Wally reaches a corner of a zone, Wally can instantly teleport to any other corner in the zone – again, not moving an inch!

You see, our friend Wally is a bit out of shape and wants to get from one location to another in Wacky World by traveling as little distance as possible. To help him, just figure out for Wally the shortest distance (in squares) that he has to travel to get from his starting location to his ending location. Movement from one location on the Wacky World grid to another adjacent location counts as having moved one square. But remember, it is possible to teleport to opposite sides of Wacky World once you reach the edge of the map without having traveled any squares.

Wally knows this is all so confusing, so he's figured out a couple of examples ahead of time for you to consider. Let's say that Wally is in a zone of Wacky World that has dimensions 6 x 6 (that is, six rows and six columns). Then a map of that zone would look like this (notice the way the grid is labeled):

	A	B	C	D	E	F
1						
2		Y				
3						
4					X	
5						
6						

Now let's say that Wally wants to get from location X (square E4) to location Y (square B2). The shortest distance Wally has to travel to get from X (E4) to Y (B2) is four squares. How so? Note that from E4, Wally first can travel to F4. That's one square. Now, since he is at the edge of the zone, he can teleport to the opposite side of the zone, to A4. Remember, at this point he has still only traveled one square. Wally travels no distance during the teleportation process. Next, Wally moves to B4. That brings our total to two squares. Finally, Wally moves to B3 and then to B2, bringing the total distance traveled to four squares. Man, this place really is Wacky!

In this example, you may notice that there are several ways to travel from X to Y by traveling only four squares. Your job is merely to report what is the least amount distance that Wally has to travel to get from one location

to another. Ready?

Details of the Input
There will be multiple problem instances, each corresponding to a distinct zone that Wally is traveling in. Each input set will begin with a line of two integers *r* and *c* ($1 \le r \le 26$, $1 \le c \le 26$), indicating the size of the zone where *r* is the number of rows and *c* is the number of columns. Values of $r = 0$ and $c = 0$ indicate the end of input. The next line will be an integer *n* ($n \le 100$) indicating the number of test cases for the particular zone. Finally, there will be *n* lines of coordinates of the form *start* and *finish*, where *start* is Wally's starting location and *finish* is Wally's final destination. See the formatting of *start* and *finish* in the Sample Input.

Details of the Output
The results for each zone should be printed in the output as follows:

```
Results for Wacky World – Zone #<case number>
```

where `case number` is the number of the problem instance starting with 1. Following this title should be *n* lines indicating the results for each test case of the form:

```
The shortest distance between <start> and <finish> is <x> squares.
```

where `start` and `finish` are the starting and ending coordinates, respectively, and where `x` is the shortest distance between those two locations.

Finally, there should be an extra blank line after each output set.

Sample Input

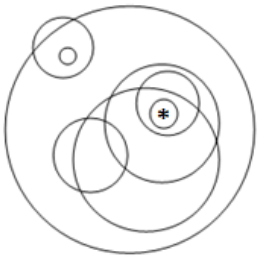
```
7 7
7
G1 A7
B2 B3
A1 G7
G1 G7
A2 F3
C6 D4
E3 A2
26 26
4
E3 F25
A1 G26
G7 E13
Q3 E17
0 0
```

Sample Output

```
Results for Wacky World – Zone #1
The shortest distance between G1 and A7 is 0 squares.
The shortest distance between B2 and B3 is 1 squares.
The shortest distance between A1 and G7 is 0 squares.
The shortest distance between G1 and G7 is 0 squares.
The shortest distance between A2 and F3 is 2 squares.
The shortest distance between C6 and D4 is 3 squares.
The shortest distance between E3 and A2 is 3 squares.

Results for Wacky World – Zone #2
The shortest distance between E3 and F25 is 4 squares.
The shortest distance between A1 and G26 is 6 squares.
The shortest distance between G7 and E13 is 8 squares.
The shortest distance between Q3 and E17 is 23 squares.
```

Problem F: Nested Circles



A circle *M* is considered "inside" another circle *N* if every point of *M* is in the interior of *N* (note that the interior of *N* does not include the points of *N* itself). Given a set of circles, we define the "nesting level" of a circle as follows:

- If the circle is not "inside" of any other circle, then it is at nesting level 1.
- Otherwise, the nesting level is 1 more than the largest nesting level of all circles it is "inside" of.

For example, the smallest circle containing the asterisk in the above image has nesting level 4, because that circle is "inside" circles with nesting levels 1, 2, 2, and 3 (each of which was calculated via the same process), and one more than the largest of those values is 4.

Given a set of circles, find the largest nesting level of any circle.

Details of the Input

Each input case will begin with a positive integer $n \leq 1000$ indicating the number of circles in the plane. Each of the following *n* lines will contain three integers:

```
x y r
```

These indicate the center of a circle (*x*, *y*) and the radius *r*. The absolute value of *x* and *y* will be less than or equal to 1000. The value of *r* will be between 1 and 1000, inclusive. There will be no duplicate circles in a particular input case. A value of $n = 0$ indicates the end of input and should not be processed.

Details of the Output

For each case, output

```
Case <i>: <nesting level>
```

where `<i>` is the case number (beginning at 1 for the first case), and `<nesting level>` is the largest nesting level of any circle in that test case.

Sample Input

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

Sample Output

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

Problem G: PhotoEdit

You are one of the developers working for PhotoEdit, the newest photo editing tool on the mobile phone market. Your job is to implement a flexible "Super Fill" tool. The user activates the Fill by clicking a pixel in the image, and then every pixel reached by the Fill algorithm (described below) is changed to some new color. In a Basic Fill, if pixels adjacent (horizontally or vertically only – no diagonals) to the chosen pixel are the same exact color as the chosen pixel, then the Fill recursively expands to those pixels and this continues until there are no more contiguous pixels of the same color to fill in with the new color.

For this problem, color values are just simple integers (None of this red-green-blue nonsense – if two "colors" are the same integer, they are the same. If the two colors are one integer apart, then the delta between them is one.).

Your "Super Fill" tool will be more flexible than the Basic Fill described above. The user will be able to specify two types of tolerance. The first we'll call Local Tolerance. The Super Fill is allowed to expand into pixels that are within the Local Tolerance of the current pixel value. Consider a case where we want to visit the following pixel colors in order: <13, 14, 15, 13, 13>. If the Local Tolerance value is 1, then the Super Fill can expand from the first pixel of color 13 to the 14 and then to 15, because both of those jumps are only 1 unit apart. This instance of the Super Fill cannot then expand from that 15 to the next pixel with color 13, because that is a delta of 2, which is larger than the Local Tolerance (even though we started at 13). Keep in mind that if there exists some *other* path of pixels that reaches that final color 13 pixel without ever jumping more than the Local Tolerance between pixels, then it is filled in by the Super Fill. If the Local Tolerance value is 3, then we can jump from a pixel of color 5 to a pixel of any of colors 2, 3, 4, 5, 6, 7, or 8. More clearly, a pixel *X* can only be included in the fill if there exists some path *P* (using horizontal/vertical movement) from the starting pixel to pixel *X* where each consecutive pair of pixels in *P* is no greater than the Local Tolerance apart in color value.

The second type of tolerance is Path Tolerance. The Path Tolerance adds the constraint that the sum of the absolute values of the deltas between each consecutive pair of pixels in *P* must be less than or equal to the Path Tolerance. For example, consider a case with Path Tolerance value of 3 with a Local Tolerance value of 1, using the path *P* with values <13, 14, 15, 14, 14, 14, 13>. The starting pixel is the first in the list. The path uses up one unit of Path Tolerance on the first jump to get to the 14, another to get to the 15, and yet a third to get to the next 14 (even though *P* has already gone through a pixel of value 14). This path is now out of Path Tolerance units, so when it eventually reaches the final 13 value, the Super Fill cannot expand into that pixel.

Your job is to execute the Super Fill given an input image, Path Tolerance, Local Tolerance and the starting pixel.

Details of the Input

Each test case will begin with one line containing six integers:

```
<height> <width> <PT> <LT> <X> <Y>
```

`Height` and `width` describe the dimensions of the input image and will both be between 1 and 1000, inclusive. `PT` and `LT` represent the Path Tolerance and Local Tolerance, respectively. The Local Tolerance will be between

0 and 65535, inclusive. The Path Tolerance will be between 0 and 1000000, inclusive. X and Y represent the coordinates (X, Y) of the starting pixel as described in the example below.

Each input case is then followed by `height` lines, each containing `width` integers. These integers describe the input image. Each pixel value will be between 0 and 65535, inclusive.

Consider an example image with `height` = 4 and `width` = 4:

If the starting position is specified as X = 1, Y = 3, then that maps to the value 2 in the table.

	X = 1	X = 2	X = 3	X = 4
Y = 1	1	4	5	8
Y = 2	8	4	6	8
Y = 3	**2**	4	5	8
Y = 4	7	4	6	8

Details of the Output

For each case, output

```
Case <i>:
<new image>
```

where `i` is the input case number starting at 1 and `new image` is the image in the same format it appeared in the input (`height` lines of `width` integers), but every pixel that was filled by Super Fill is printed simply as 'F'.

Sample Input

```
1 5 100 1 1 1
13 14 15 13 13
1 7 3 1 1 1
13 14 15 14 14 14 13
3 3 3 3 2 2
1 2 1
4 5 8
1 9 1
4 4 3 2 1 3
1 4 5 8
8 4 6 8
2 4 5 8
7 4 6 8
0 0
```

Sample Output

```
Case 1:
F F F 13 13
Case 2:
F F F F F F 13
Case 3:
1 F 1
F F F
1 9 1
Case 4:
1 F F 8
8 F 6 8
F F F 8
7 F 6 8
```