# KU ACM Intramural Programming Competition

April 19, 2014

INSTRUCTIONS: The KU ACM Intramural Programming Competition is an ACM-styled competition in which 10 problems, in ascending difficulty, will be administered in a period of five hours. You may complete the problems in any order you wish. The following languages are supported:

C
C++
C#
Clojure
Go
Haskell
Java
JavaScript
Lua
Objective-C
Perl
PHP
Prolog
Python2
Python3
Racket
Ruby
Scala

# 1 DATA ENTRY DEBACLE

DESCRIPTION: A phone book company attempted to digitize all of their listed numbers and had to hire teams to manually input listings from paper phone books. However, one worker's number pad had a faulty '1' key, causing all of the numbers entered by this worker to be missing a digit. Thankfully, when the work was divided, this person received only numbers with a '1' appearing in the fourth digit place of a standard 10 digit phone number.

INPUT: Read from stdin a list of complete and incomplete 10 digit phone numbers, each separated by newlines, from stdin. The first line of input will be the number of phone numbers to read. It is guaranteed that each line of input be at least 9 digits long and no greater than 10 digits in length. Numbers read from left to right, and are indexed starting at one.

OUTPUT: Write all the phone numbers back to stdout but with the incomplete numbers made correct by inserting a '1' digit in the appropriate place. Each number must be separated by a newline.

```
Sample Input:
4
7855524364
840525369
558347896
9886354789

Sample Output:
7855524364
8401525369
5581347896
9886354789
```

## 2 HAPPY MUSHROOMS

DESCRIPTION: Forsaking the sun, you decide the live out the rest of your years underground. In order to survive, you collect a wide array of luminous fungi on which you intend to subsist yourself. However, you know that for a subterranean mushroom to reach its full size, it needs space around it. After digging rectangular caves in which to grow your mushrooms, you discover that a mushroom will whither and die if it is ever placed next to a wall, or placed adjacent to another mushroom. You decide to write a program that will simulate the planting of various mushroom "farms."

INPUT: Read from stdin a character based representation of the layouts of potential farms. The first line will be an integer representing the number of farms to test. The subsequent lines will represent a collection of farms, each headed by two, space-delimited integers denoting the dimensions of the following farm, number of rows by number of columns. The cells of the farm will be denoted either as # if the cell is dirt, or as 'M' if the cell contains a mushroom. Walls are denoted by the bounds of the farm. Each line of input will be separated by a newline.

OUTPUT: For each farm, write to stdout an integer representing the number of mushrooms that would die if a given farm was allowed to grow. Mushrooms will die if placed next to walls or directly adjacent another mushroom. A mushroom is adjacent to another mushroom if and only if it is above, below, to the left, or to the right of the other mushroom in the grid. Each integer must be separated by a newline.

```
Sample Input:
2
5 5
#####
#M#M#
##M##
#M#M#
#####
5 4
M###
#M##
##M#
##M#
M###

Sample Output:
0
4
```

# 3 ROBOT TESTING FACTORY FACTORY

DESCRIPTION: You work for a company that wants to test a new line of walking robots en masse. Your project director has laid out a grid on which to place the robots, and has also told you which position to place them, and what direction they should be initially facing. You are also told the number of steps the robot should take in one test session. Each robot also reacts when it reaches one of the walls of the grid, turning a specified number of degrees before moving again. Your boss has tasked you to write a simulation program so that the actual movements of the robots can be tested against your theoretical expectation.

INPUT: Read from stdin a succession of tests parameters for your robot. The first line will be an integer $T$, that indicates the number of tests to perform. The subsequent $T$ lines will be your cases. Each test case has the format:

*M N X Y StartDir RotDir S*

where $M$ is the number of rows in the grid, $N$ is the number of columns in the grid, $X$ $Y$ is the coordinate that the robot will be initially placed with $X$ as the row coordinate, and $Y$ as the column coordinate. Note that the top left cell of the grid is indexed as *(0, 0)* and the bottom right cell is indexed as *(M-1, N-1)*. *StartDir* is the direction the robot will initially face in degrees, where 0 indicates the robot starts facing the right wall of the grid, 90 indicates it starts facing up, 180 indicates it starts facing left, and 270 indicates it starts facing down. *RotDir* indicates what direction the robot will turn when it encounters a wall. 90 indicates the robot will turn 90 degrees relative to the wall it hit, -90 indicates it will turn -90 degrees, and 180 indicates the robot will turn 180 degrees (i.e., it will turn about-face from the wall it hit). $S$ is the number of steps the robot will take in the simulation, that is, the number of cells it will traverse in the grid. Note, a rotation is not considered a step, upon hitting a wall the robot will turn immediately. All variables will be integers, and each member variable of a test case will be whitespace delimited. Each test case will be separated by a newline.

OUTPUT: Write to stdout the final $X$ $Y$ position of the robot at the end of each test case. Separate each case by a newline.

```
Sample Input:
3
5 5 0 0 90 90 5
10 10 3 3 0 180 7
3 7 2 2 180 -90 4

Sample Output:
1 4
8 3
0 0
```

# 4 CONSOLE CRISIS

DESCRIPTION: You just bought a new game console, and you and your $n$ friends (since you have a variable number of friends) are planning on assembling for a most glorious weekend of gaming. Unfortunately, you don't have enough controllers! Circumstances require you buy more.

Although your console (the newly released PlayBox InfiniJamz) only has four available ports, you can support an arbitrary number of additional controllers by use of a special 'splitter', a device that splits a single port into three ports. Each extra controller costs $30, and each extra splitter costs $15. You already have a certain number of controllers, but you may need to buy more controllers (and possibly some splitters) to accommodate your given $n$ friends. Note that you are not playing, only your friends -- you much prefer to watch.

INPUT: Read from stdin a number of tests cases representing differing numbers of friends and starting equipment. The first line will be an integer $c$, which will represent the number of test cases. The following $c$ lines will each consist of integers $n$ $m$ (delimited by whitespace) such that $0 \leq n \leq 1000000000$ and $0 \leq m \leq 1000000000$, where $n$ represents the number of friends and $m$ represents the number of controllers you already have. Assume that you have no splitters at the beginning of each test case (i.e., if you buy splitters in a previous test case, they do not "carry over" to subsequent test cases).

OUTPUT: For each test case, write to stdout an integer representing the total cost (without dollar signs) of the number of controllers and splitters you would need to buy. If you do not need to buy anything, simply output `0`.

```
Sample Input:
3
4 2
4 4
6 0

Sample Output:
60
0
195
```

# 5 GOBLIN MASTER

DESCRIPTION: Having felt a bit evil lately, you decide to throw your lot in with a horde of goblins as their devious advisor. Hoping to minimize casualties during goblin invasions, you decide to write a generalized program that determines if a goblin army would be victorious in battle, or if they would be defeated.

Thankfully, your airborne Goblin Dirigible Corps provide you excellent data on the battle readiness of your enemies, and you can also safely make several assumptions about how troops both on your side, and on opposing sides, attack:

- On each turn, the unit with the highest *offense* from each team will attack the enemy with the lowest *health*. If there is a tie among two enemy unit for lowest *health*, the unit with the highest *offense* will be the one that is attacked
- If the *offense* from one enemy is greater than the *health* of their opponent, the remaining *offense* points *(offense - enemy health)* will dealt to the next weakest enemy
- On each turn, whatever *offense* points were previously expended by a unit are replenished, but *health* of any previously damaged unit is not replenished
- A unit with whose *health* is reduced to zero is considered killed, i.e., it is considered removed from its army and can no longer attack
- You can also assume the goblins always attack first, because dwarves are slow and rarely take improved initiative as a combat feat

The goblins have varied types units, as do their mortal enemies, the dwarves, and each has different stats:

| Goblins | Health | Offense |
|---|---|---|
| Archers | 5 | 10 |
| Infantry | 10 | 7 |
| Trolls | 10 | 15 |

| Dwarves | Health | Offense |
|---|---|---|
| Pikeman | 15 | 5 |
| Swordsmen | 7 | 7 |
| Ballista | 5 | 20 |

INPUT: Read from stdin a manifest detailing the number of different units on both sides. Input will consist of 6 lines, each with a leading integer *n* representing the number of a type of unit, and the unit *type*, separated by whitespace.

OUTPUT: Write to stdout either the statement `Victory` if your goblins kill all the units of the dwarven army first, or `Defeat`, if the dwarves kill all the units in your goblin army first.

```
Sample Input:
3 Pikemen
2 Swordsmen
1 Ballista
1 Archers
4 Infantry
1 Trolls

Sample Output:
Victory
```

# 6 CHARPERM CODES

DESCRIPTION: A charperm code is defined to be a set of codewords, where each codeword consists of a fixed number of characters in some order. For example, consider a charperm code where each codeword contains exactly 3 occurrences of *x*, exactly 2 of *y*, and exactly 1 of *z*. Then 3 of the 60 possible codewords are:

```
xyxxyz
xyxxzy
xyxyxz
```

These codewords are in alphabetical order, and if all 60 codewords were listed in alphabetical order, then these 3 codewords would appear consecutively. Hence we define a successor of a codeword to be the codeword which follows it in alphabetical order (e.g., `xyxxzy` is the successor of `xyxxyz`). If a codeword is last alphabetically, it has no successor.

INPUT: Read from stdin a number of codewords to find the successor of. The first line will be an integer *n*, and the subsequent *n* lines will contain single codewords where each codeword is between 1 and 100 lowercase alphabet letters, each taken from different charperm codes. Each codeword will be separated by a single newline.

OUTPUT: Write to stdout the charperm successor of each codeword. If a given codeword has no successor, write `No successor`.

```
Sample input:
2
ghggih
onnmm

Sample output:
ghghgi
No successor
```

# 7 TILE TERROR

DESCRIPTION: Much to your chagrin, you are placed into a grid-shaped labyrinth where the floor consists of tiles that can move you either up, left, right, or down. The labyrinth's magic prevents you from moving voluntarily, leaving you at the mercy of the tiles' movement. You want to determine how quickly you can leave the labyrinth -- or if you will be stuck in an endless tile loop.

INPUT: Read from stdin a succession of labyrinths that you must escape. Each labyrinth is headed by four, white-space delimited numbers *r c x y*, followed by a newline, where *r* is the number of tile rows in the labyrinth, *c* is the number of tile columns, and *(x, y)* is the coordinate of the tile where you are first placed. Note that the upper-left tile is indexed *(0, 0)*, and the bottom right tile is indexed *(r-1, c-1)*. The following *r* lines will each consist of *c* characters from the set *{U, L, R, D}*, where *U* represents an up tile, *L* a left tile, *R* a right tile, and *D* a down tile. The end of input is signaled by a Labyrinth header consisting entirely of zeros, *0 0 0 0*.

OUTPUT: For each labyrinth, determine if the labyrinth can be exited (i.e., move you out of the bounds of the grid). If the labyrinth can be exited, calculate *s*, the number of steps needed to exit (moving from one tile to another, or moving from a tile to out-of-bounds, counts as 1 step), then write "s steps to exit" to stdout. If the tile places you in a loop, calculate *t*, the number of steps needed to reach the loop, and calculate *u*, the number of steps in the loop, then write "t steps before loop with u steps" to stdout. Do not worry about pluralization of "steps" if there is only 1 step.

```
Sample input:
4 5 1 0
RDRDL
RRUDL
RDLLR
LRRRR
3 6 0 0
DDLLLL
DRRRDU
RRRRRU
2 2 0 0
DR
UR
0 0 0 0

Sample output:
13 steps to exit
6 steps before loop with 12 steps
0 steps before loop with 2 steps
```

# 8 THE GREAT EQUATIONIZER

DESCRIPTION: You take equality very seriously -- so much that, when you see any group of numbers, you take the time to add arithmetic operators around them to create an equation. There is a catch: you are only allowed to use addition, subtraction, multiplication, parentheses, and the equals sign to achieve your goal. For example, given the numbers below:

```
4 2 1 1
```

you can create 5 distinct arithmetic equations with the above constraints:

```
4 = 2*(1+1)
4 = 2+1+1
4-2 = 1+1
4-(2+1) = 1
4-2-1 = 1
```

Note that parentheses are only used where omitting them would produce a different result. As an example, `4-(2+1) = 1`, but `4-2+1 = 3`. However, `4 = 2+1+1` and `4 = 2+(1+1)`, so we do not count `4 = 2+(1+1)` as a different equation.

INPUT: Read from stdin a collection of numbers with which you could write various expressions. Each set of numbers will begin with a nonnegative integer *n*. If *n > 0*, then use the following n integers (which are all nonnegative) to find arithmetic equations. A zero value of n indicates the end of input. Since your equations would need to be well-formed, it will always be the case that *n* is at least 2 except at the end of input, when it will be `0`. Each number in the test cases will be whitespace delimited, and each test case will be separated by a newline.

OUTPUT: For each set of *n* numbers, write to stdout an integer representing how many distinct arithmetic equations can be formed with the above constraints. Separate each case with a newline.

```
Sample input:
4 4 2 1 1
6 2 4 6 6 4 2
4 4 1 1 1
0

Sample output:
5
94
0
```

# 9 CROW-NIHILATOR-3000

DESCRIPTION: Scarecrows ℝ Us is developing a new kind of weaponized scarecrow that can detect nearby crows and vaporize them.* The scarecrow can see directly up, down, left, right, and diagonally. For example, the grid below represents a field that is 8 acres by 8 acres in size, the `X` represents the scarecrow, an s represents an acre it can see, and a # represents an acre it cannot see.

```
s##s##s#
#s#s#s##
##sss###
sssXssss
##sss###
#s#s#s##
s##s##s#
###s###s
```

While this new scarecrow technology is cutting-edge, it comes with one major drawback: scarecrows are unable to distinguish other scarecrows from crows. If two scarecrows were configured like this:

```
sssssssX
s#####ss
s####s#s
s###s##s
s##s###s
s#s####s
ss#####s
Xsssssss
```

Then the two scarecrows would vaporize each other, since they are in each other‘s diagonal line of sight. Gardeners want to be assured that mutual scarecrow self-destruction won‘t occur, so you are tasked with arranging 8 scarecrows on a 8 x 8 acre field such that the following conditions hold:

1. All acres of the field are either occupied by a scarecrow or in a line of sight of a scarecrow
2. No two scarecrows are in each other‘s line of sight.

*No crows were harmed in the testing of this problem.

When a gardener gives you a field to work on, you can assume that there is exactly one scarecrow per column, but there may be scarecrows in each other's horizontal or diagonal lines of sight. You must find the minimum number of scarecrow movements needed to achieve the above two conditions. One move consists of moving a single scarecrow to a new position in its column (i.e., you are only allowed to move scarecrows vertically).

INPUT: Read from stdin lines of integers representing arrangements of scarecrows in a field. Each line will contain a list of 8 integers, each in the range [1, 8] inclusive, and each integer whitespace delimited. The value of the *ith* integer in the list represents the row occupied by the scarecrow in its respective *ith* column. For example, in the list `1 2 3 4 5 6 7 8`, the '1' would represent the scarecrow of the first column occupying the cell in the first row (i.e., the scarecrow would be in the upper-left acre), and the '8' would represent the scarecrow of the 8th column occupying the cell in the 8th row (i.e., the scarecrow would be in the bottom-left acre). Each line will be separated by a newline. If all the numbers in the list are 0, that indicates end of input.

OUTPUT: For each configuration of scarecrows, write to stdout an integer representing the minimum number of scarecrow movements needed achieve the two conditions stated above.

```
Sample input:
1 2 3 4 5 6 7 8
4 4 4 4 4 4 4 4
1 3 5 7 2 4 6 8
6 4 7 1 8 2 5 3
0 0 0 0 0 0 0 0

Sample output:
7
7
3
0
```

## 10 RIGHT IN THE JUG-ULAR

DESCRIPTION: You just inherited a farm from your crazy uncle (we all have one). As it turns out, your uncle owns a lot of cows and has to milk them regularly, so he also owns a large collection of milk jugs. Meanwhile, the curious townsfolk have started to notice you, and like good neighbors, they want to buy your cows' milk.

You receive an unusual request one day from a pair of customers. They each want an unspecified amount of milk, but they stipulate that each of them should receive the exact same amount, and that the containers of milk they receive should be filled to capacity. Unfortunately, your crazy uncle's milk jugs come in many different sizes, so it's up to you to divide up your jugs in a certain way in order to appease your customers' order.

For example, suppose you have one 1-liter jug, two 5-liter jugs, and one 11-liter jug. You can meet the customers' demands by giving one of them the 11-liter jug and the other three jugs to the other. Note that you cannot fill up only the two 5-liter jugs and give one to each customer -- they know how many jugs you have, and they'll feel cheated if you don't fill up all of them. Sometimes, though, you simply won't be able to divide the jugs among the two (e.g., if you had one 1-liter jug, one 3-liter jug, and one 5-liter jug), so be ready to break the bad news if needed.

INPUT: Read from stdin several collections of integers representing the sizes of various milk jugs in a given session to divide. Each line will begin with a nonnegative integer $n$, where $0 \leq n \leq 2,000$. If $n = 0$, that indicates the end of input. If $n > 0$, then $n$ indicates the number of whitespace-delimited integers (each between 1 and 2,000) that follow it on that line, where each of the $n$ following integers will represent a milk jug size.

OUTPUT: For each set of jugs, determine if it is possible to split up all of the milk jugs (filled to capacity) among the two customers such that each customer receives the exact same amount of milk. If it is possible, write `1` to stdout. Otherwise, write `0`.

```
Sample input:
4 1 5 5 11
3 1 3 5
0

Sample output:
1
0
```