2016 CCSC Eastern Conference Programming Competition

October 29th, 2016

Frostburg State University, Frostburg, Maryland

This page is intentionally left blank.

Question 1 — And Chips

For a Splotvian twist on any dish, just add chips! Yes, Splotvians love their chips, and will eat them with just about anything.

In order to help restaurants update their menus to cater to fans of Splotvian cuisine, you decide to write a program that will take the name of (almost) any dish and add "`and chips`" to it. An exception occurs for dishes whose names start with "q": Splotvians will never eat chips with these.

Program input: Each line of input is the name of a menu item. As a special case, if a line consists of just the text "`done`", that signals the end of input, and the program should exit immediately without any further output.

Example input:

```
pie
biryani
steak tartare
chicken cordon bleu
quinoa with wild mushrooms
jellied eels
baked alaska
cherries jubilee
done
```

Program output: For each input line (other than "`done`"), the program should print the same line, but with "`and chips`" added (with a space before "`and`"), unless the name of the dish starts with "q", in which case the name of the dish should be printed verbatim.

Example output (corresponding to the example input shown above):

```
pie and chips
biryani and chips
steak tartare and chips
chicken cordon bleu and chips
quinoa with wild mushrooms
jellied eels and chips
baked alaska and chips
cherries jubilee and chips
```

This page is intentionally left blank.

Question 2 — Know When to Hold 'Em

Splotvian poker has only four kinds of hands: four of a kind, three of a kind, pair, and high card:

- A *four of a kind* has four cards with the same rank (two, three, etc.) — this is the best type of hand
- A *three of a kind* has three cards with the same rank
- A *pair* has two cards with the same rank
- A *high card* is any hand that is not one of the hands listed above — this is the worst type of hand

Each card is represented by a rank (from lowest to highest, one of `23456789TJQKA`) and a suit (from lowest to highest, one of `CDHS`). A hand consists of five cards.

As one of the world's top Splotvian poker players, you decide to write a program to compare Splotvian poker hands. Hands are compared according to the following rules:

- A better type of hand always beats a worse type of hand; e.g., four of a kind always beats three of a kind, two of a kind always beats high card, etc.
- When comparing two high card hands, the hand with the better high card beats the hand with the worse high card (see below)
- When comparing two three or four of a kind hands, or two pair hands, the set of same-rank cards with the higher rank wins (e.g., three tens beats three nines)

A single card is "better" than another card if its rank is higher, or (if its rank is the same as the other card) its suit is higher. For example, nine of clubs (`9C`) is better than eight of spades (`8S`), and eight of spades (`8S`) is better than eight of hearts (`8H`). The high card in a hand is the one that is better than all the others.

As a special case, a hand with two pairs of cards with the same rank is considered to be a pair, and for purposes of comparing the hand with another hand, the pair of cards with the higher rank is the one that should be used. (E.g., if a hand has two fours and two sixes, it should be considered to be a pair of sixes.)

As another special case, if two pair hands are compared, and both pairs have the same rank, the hands should be considered as equal to each other.

Program input: The program should read pairs of lines, each line specifying a hand. Each hand is five cards, separated by spaces. Each card is a rank followed by a suit. As a special case, if a line is the text "`done`", that signals the end of the input, and the program should exit without further output. Example input:

```
QD 7D 3S 8C 7S
6C TD AS 2S 9H
2H 4H AS 2C 5D
7S AC JC QH 7C
```

```
AS AH 2C 3C 8D
5H 6S AC 6D 6C
KD 5H 6D KC KH
8H 8C JD 8D 8S
7S 9H 9D TC AC
8D AD QS 8H 3C
7C 4D AC JS 9C
2D 8C TD 7S KS
done
```

Program output: For each pair of hands in the input, the program should print a single line: `1` if the first hand is better than the second, `-1` if the first hand is worse than the second, and `0` if they are equal. Example output (corresponding to the example input shown above):

```
1
-1
-1
-1
1
1
```

Question 3 — Clock Wise

Your ClockMaster 3000™ alarm clock has a peculiar feature. It allows you to set an increment value n, where n is a positive integer. Once the increment value is set, the clock will only update its display every n minutes, until its display shows the same hour and minute values as when it entered increment mode. For example, if the clock enters increment mode at 1:57, it will remain in increment mode until the next time it displays 1:57. (Note that this could be *either* 1:57 AM *or* 1:57 PM.)

Due to the ClockMaster 3000™'s poorly-implemented voice recognition feature, it is very easy to enable increment mode by accident (for example, by sneezing). The good news is that while in increment mode, the ClockMaster 3000™ displays the current increment value, so at least you know what it's doing.

Since you are determined to make the best of the situation, you decide to write a program to determine how many minutes you will have to wait until the clock goes back to normal after you accidentally enable increment mode.

Program input: The program reads lines of text from standard input. Each normal line of input has one integer value, n, which is a positive value indicating the increment. (Note that it doesn't matter what time is displayed when the clock enters increment mode.) As a special case, a line with a value of -1 indicates that there are no more input lines and the program should terminate immediately (without any further output).

Example input:

```
7
26
61
27
-1
```

Program output: For each positive increment value, the program should print a line with a single integer, representing the total number of minutes until the clock resumes normal mode.

Example output (corresponding to the example input shown above)

```
5040
9360
43920
2160
```

This page is intentionally left blank.

Question 4 — Mini Golf

Splotvian mini golf is the latest craze! The rules are pretty simple:

- It's played on a rectangular grid
- Each grid cell is turf (`.`), cup (`%`), or wall (`#`)
- There are walls (`#`) on all sides
- There is one ball (`*`), which you can assume is placed on top of turf
- At each time step, the position of the ball changes by dy units vertically and dx units horizontally
- At any time step, the possible values for dx and dy are 1 and -1: 1 means one unit to the right (dx) or down (dy), -1 means one unit left (dx) or up (dy)
- The ball stops either when it falls in the cup, or after it has moved a specified maximum number of time steps without falling in the cup

The most interesting feature of Splotvian mini golf is a *collision*. A *collision* occurs when the ball's current direction (dx and dy) would cause it to intersect a wall (`#`) in the next time step. A collision will cause a change to the ball's dx value, dy value, or both, in order to deflect it away from the wall, according to the following rules (which are considered in order):

1. If flipping the sign of dy directs the ball to a non-wall cell, then the sign of dy is flipped
2. If flipping the sign of dx directs the ball to a non-wall cell, then the sign of dx is flipped
3. If flipping the sign of both dx and dy directs the ball to a non-wall cell, then the signs of both dx and dy are flipped

Here are before and after illustrations of each type of collision, each of which assumes that the ball (`*`) is initially traveling up and to the right:

```
.....    .....      ...#.    ...#.      .....    .....
#####    #####      ...#.    ...#.      ####.    ####.
.*...    .....      ...#.    .*.#.      ..*#.    ...#.
.....    ..*..      ..*#.    ...#.      ...#.    .*.#.
.....    .....      ...#.    ...#.      ...#.    ...#.
```

Collision type 1 Collision type 2 Collision type 3

Program input: The input is a series of courses. A course is a single specification line, followed by some number of lines specifying the layout of the course. The specification line has five integer values: h, w, dx, dy, and t. The h and w values are the height and width of the course. The dx and dy values indicate the initial direction of the ball. The t value

indicates the maximum number of time steps the ball will move. The lines specifying the course layout describe each row of the course, using the characters `#.%*` to represent wall, turf, cup, and the ball. As a special case, if the specification line consists of a single -1 value, that indicates the end of input, and the program should exit without further output. Example input:

```
6 12 1 1 30
############
#..........#
#......%%..#
#.*....%%..#
#..........#
############
7 25 1 -1 21
#########################
#..........#............#
#..........#......%%....#
#.....#....#......%%....#
#..*..#.................#
#.....#.................#
#########################
-1
```

Program output: For each input course, the program should determine the movement of the ball at each time step until it either falls in the cup or stops moving because it has reached the maximum number of steps. Each time a collision occurs, it should be represented as `x`c`,`r`@`t, where c is the column where the collision occurred, r is the row, and t is the time step. Rows and columns are numbered starting at 0 (top row/left column.) If the ball reaches the cup, it should be represented as `p`c`,`r`@`t, where c and r are column and row, and t is the time step. Time step 0 is the start of the simulation, where the ball is in its initial position. If the ball stops before reaching the cup, the program should print `miss`. The information for each event (collision, reaching the cup, or miss) should be printed on a single line, each event separated by a single space.

Example output (corresponding to the example input shown above):

```
x3,4@1 x6,1@4 p7,2@5
x6,1@3 x10,5@7 x14,1@11 x18,5@15 x22,1@19 x23,2@20 miss
```

Hints and specifications:

- You can assume each course will have at most 50 rows and 50 columns
- You can assume that the edges of a course will have walls, so the ball cannot go off the edge of the course
- There can be multiple cup (`%`) locations in a course, and the ball is considered to be in the cup if it reaches any of them

Question 5 — A "Maze"-Ing

The Queen of Splotvia has banished you to her fiendish labyrinth. Will you wander endlessly, or can you find the door to freedom?

Lucky for you, there aren't really any dangers in the labyrinth (other than boredom). The labyrinth is a grid, where each square of the grid is a wall (`#`), floor (`.`), entry point (`>`), or door to freedom (`$`). Your task is to find the shortest path from the entry point to the door to freedom. You can move one square at a time, and you can only move up, down, left, or right. Also, you can't move through a wall.

Program input: The input to the program is zero or more mazes. Each maze is specified as follows. The first line of input of the maze has two integer values, h and w, where h is the height of the maze grid and w is the width of the maze grid. The next h lines of input specify one row of the maze grid.

As a special case, if the first line of input for a maze is a single integer -1, then there is no more input and the program should terminate immediately without any further output.

Example input:

```
5 11
###########
>...#...#.#
#.###.#.#.#
#.....#...$
###########
9 24
########################
#.......#....##.....#..$
#.#.#######.....###.##.#
#.#.........###.#......#
#.#####.###.###.#.######
#.#.......#.###.#......#
>...###.#.#...#.######.#
#.#.....#...#..........#
########################
5 14
##############
#...#.....#..$
#.#.#.##.###.#
#.#...#....#.#
#>############
-1
```

Program output: For each input maze, the program should find the shortest path from the entry point to the door to freedom, if any such path exists. If there is a path, the program should print a line with the number of moves in the shorest path, followed by a printed representation of the maze (similar to the input representation) with percent (`%`) characters indicating the shortest path between the entry point and door to freedom. (Note that only the floor (`.`) should be changed to caret (`%`).) If there is no path, the program should print a single line of text, "`No escape`".

Example output (corresponding to the example input shown above):

```
16
###########
>%..#%%%#.#
#%#######%#.#
#%%%%%%%%%$
###########
32
########################
#.......#....#######.%$
#.#.########%%%%%%%%%%#
#.#....%%%%%%##.#..%%%%#
#.######%##.###.#.######
#.#%%%%..#.###.#......#
>%%%##.#.#...#.######.#
#.#.....#...#.........#
########################
No escape
```

Hints and specifications:

- Each maze will have exactly one entry point (`>`) and exactly one door to freedom (`$`)
- Path length is the number of moves (up, down, left, or right) required to travel from the entry point to the door to freedom
- You should not make any assumptions about the locations of the entry point and door to freedom: both can be anywhere in the maze
- You can assume that if a maze has a solution, there will be a unique shortest-path solution
- You can assume each maze will have at most 50 rows and 50 columns

Question 6 — Counting Pig Roll Sequences

The game of Pig is a very simple jeopardy dice game in which two players race to reach 100 points. Each turn, a player repeatedly rolls a six-sided die until either a 1 (called a "pig" is rolled or the player holds and scores the sum of the rolls (i.e. the *turn total*). At any time during a player's turn, the player must choose between two possibilities:

roll The player rolls the die. If the player rolls a 1 (a "pig"), the player scores nothing for the turn and it becomes the opponent's turn. If the player rolls 2–6, the number is added to the player's turn total and the player's turn continues.

hold The turn total is added to the player's score and it becomes the opponent's turn.

The Gambler's Fallacy: Some middle school teaching materials that use Pig to teach probability imply or explicitly state that it is useful to know how many non-pig rolls (> 1) have occurred in order to decide when to hold. This is an example of the Gambler's Fallacy where the next roll is thought to be influenced by previous rolls. As is often said, "Dice have no memory." In Pig, you could have a turn total of 6 from a single roll of 6 or three rolls of 2, but the next roll is independent of how many rolls have gone before.

For this problem, you will repeatedly compute how many different possible roll sequences result in a turn total of k, where $0 \leq k \leq 50$. For example, there are 5 possible roll sequences that result in a turn total of 6: 2-2-2, 2-4, 3-3, 4-2, and 6. Note that 2-4 and 4-2 count as distinct roll sequences; you are not counting roll sets. Note also that the empty sequence leads to a turn total of 0 at the beginning of your turn.

Program input: The input will consist of n integers, with $n > 0$. The first $n-1$ integers will be turn totals in the range [0, 50]. The last integer will be a -1, indicating the end of input. Example input:

```
2
6
10
-1
```

Program output: For each non-negative input k, you will output a single line with the integer number of possible roll sequences that result in a turn total of k. Example output (corresponding to the example input shown above):

```
1
5
29
```

This page is intentionally left blank.

Question 7 — Reorderings

The SplotMatic Gargantuabrain 9000™ is one of the first computers ever developed in the Republic of Splotvia. Its hardware is rather limited. It has three registers, `A`, `X`, and `Y`, each of which can hold a signed integer value. It supports the following hardware instructions:

- `MOV` *dreg val*: copy *val* into the register *dreg*
- `ADD` *val1 val2*: add *val1* and *val2* and store the sum in the `A` register
- `SUB` *val1 val2*: subtract *val2* from *val1* and store the difference in the `A` register
- `MUL` *val1 val2*: multiply *val1* and *val2* and store the product in the `A` register
- `DIV` *val1 val2*: divide *val1* by *val2* and store the whole part of the quotient in the `A` register, discarding the remainder (if any)

A value (*val*, *val1*, or *val2*) can be either a register (`A`, `X`, or `Y`), or an integer value. When a program starts running, all of the registers contain the value 0 (zero). Instructions are executed in order.

As part of the compiler team at SplotCo™, one of your jobs is to test new compiler optimizations. The performance of the SplotMatic Gargantuabrain 9000™ is highly sensitive to the precise order in which instructions are executed. Finding a way to reorder instructions without changing the result of the program can result in huge performance gains. Two programs are considered to have the same result if at the end of executing both programs, the values of the registers (`A`, `X`, and `Y`) are the same. For example, the following programs have the same result:

— *First program*	— *Second program*
`MOV X 4`	`MOV Y 5`
`MOV Y 5`	`MOV X 4`
`ADD X Y`	`ADD X Y`

At the end of both programs, `X`=4, `Y`=5, and `A`=9.

Your task is to write a program which can analyze a program and determine how many equivalent programs can be created by reordering the instructions of the original program.

Program input: The input is a series of *instruction sequences*. Each instruction sequence begins with a line specifying how many instructions the instruction sequence has. This is followed by lines with the instructions, in order. As a special case, if the number of instructions is -1, that is a signal that the input has ended and the program should exit without further output. Example input:

```
3
MOV X 4
MOV Y 5
```

```
ADD X Y
5
MOV A 100
MOV X 5
DIV A X
MOV Y 3
DIV A Y
4
MOV X 17
MOV Y -9
SUB X Y
MOV Y 44
-1
```

Program output: For each instruction sequence, the program should print two lines of output. The first line has the form

A=*aval* X=*xval* Y=*yval*

where *aval*, *xval*, and *yval* are the final values of the A, X, and Y registers. The second line is a single integer value indicating the number of permutations of the instruction sequence which have the same result as the original. Note that the original ordering is considered to be one of the permutations, so this number will be at least 1. Example output (corresponding to the example input shown above):

```
A=9 X=4 Y=5
2
A=6 X=5 Y=3
16
A=26 X=17 Y=44
2
```

Hints and specifications:

- An attempt to divide by 0 should cause a 0 to be stored in the A register
- You can assume that the instructions in each instruction sequence will be distinct from each other: e.g., the instruction `MOV X 7` could not appear twice in the same instruction sequence