ACM@KU
Intramural Programming Competition

April 23, 2016

INSTRUCTIONS: The KU ACM Intramural Programming Competition is an ACM-styled competition in which 10 problems of assorted 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 (gcc 4.8.2)
C++ (g++ 4.8.2)
Haskell (ghc 7.6.3)
Java (Oracle Java 1.8.0)
JavaScript (Node 0.10.25)
Lua (5.2.3)
PHP 5
Python2.7
Python3.4
Ruby 1.9.3

1 Secret of the Sorting Hat

DESCRIPTION: Taking an unsolicited and opinionated moment to compare J. K. Rowling's *Harry Potter* to Terry Pratchett's *Discworld*, you may note that Hogwarts has no department for computing, whereas the Unseen University has Hex, a computer built and overseen by the wizard, Ponder Stibbons. Hogwarts' lack of computational expertise is glaringly obvious should you look at its student roster database. Their database does not actually relate a student to a house, rather, it stores a letter code next to the student's name. Even worse, this list is not sorted [1]. Your task will be to go through the database and sort out the students, then print them in a sensible manner.

INPUT: Read from `stdin` an integer n representing the number of lines of input. Each of the subsequent n lines will contain a House Indicator and then a student's full name (first then last). The House Indicators are `G` for `Gryffindor`, `H` for `Hufflepuff`, `R` for `Ravenclaw`, and `S` for `Slytherin`.

OUTPUT: Write to `stdout` each House in alphabetical order, and their list of students also in alphabetical order, sorted by the first name. Each listing of students by House should begin with the full name of the House (as given above), followed by a colon `:` and a newline, then by the sorted listing of newline separated students.

```
Sample Input:
6
G Harry Potter
S Draco Malfoy
G Hermione Granger
H Newton Scamander
S Tom Riddle
R Luna Lovegood

Sample Output:
Gryffindor:
Harry Potter
Hermione Granger
Hufflepuff:
Newton Scamander
Ravenclaw:
Luna Lovegood
Slytherin:
Draco Malfoy
Tom Riddle
```

[1] As an embarrassing aside, legend has it that the Sorting Hat actually employs Bogosort, essentially 'making it up as it goes along' with a complexity of O((n+1)!)

2 Perfect IPC Progressions

Description: We got so excited for IPC that we started seeing its name everywhere. We noticed certain English words had the “target” letters `I P C` in order, though sometimes with other letters in between. We decided we wanted to know what the full list of such perfect IPC progressions were in the English language, and we also want to know how many additional letters separate the target letters in the given word. Write a program to determine if the letters `I P C` appear in order in a given word, and state how many additional letters appear between `I P C`.

Input: Read from `stdin` an integer *n* representing the number of lines of input. Each of the subsequent *n* lines will represent a word to check for our target letters, `I P C`, and for which to count the number of letters in between.

Output: For each word, write to `stdout` the number of letters that appear between the first occurrence of `I` and the first following `P` (non-inclusive), and then between the first following `P` and the first following `C`. You should print the number of additional letters if and only if the letters `I P C` can be found in the string in order (though they need not necessarily be consecutive). If the letters `I P C` cannot be found in the word in their proper order, print `INVALID` instead.

```
Sample Input:
6
RIPCORD
DREAM
IMPERFECT
SCEPTIC
IMPACT
SUPERCALIFRAGILISTICEXPIALIDOCIUOS

Sample Output:
0
INVALID
5
INVALID
2
19
```

3 Complex Complex Complex

Description: Computing professionals have not been known for their skills in creative writing. As an incredibly lazy ENGL 101 student, you want to write a program that will take your halfhearted essays and swap out simple words for those fancy 5-dollar words using a given thesaurus.

Input: Read from `stdin` an integer *n* representing the number of new-line separated entries in your thesaurus. Each subsequent line will be a thesaurus entry, that is, a pair of space-separated words. The first word will represent a word to be replaced, and the second word will be the word you replace with. In addition to the first *n* lines, there will be an additional line of text representing the body of text into which you will perform your substitutions. You will be guaranteed that the body of text has no punctuation.

Output: Write to `stdout` the transformed version of the body of text, where each word that also appears in the thesaurus has been swapped out for its corresponding entry.

```
Sample Input:
3
building complex
problem complex
obtuse complex
I told my psychologist I have a deep fear of complicated structures
    and they told me I had a obtuse building problem

Sample Output:
I told my psychologist I have a deep fear of complicated structures
    and they told me I had a complex complex complex
```

Note: Although the body text in the sample above appears on two lines, the sample text from input will always be in one line, and output should always be written to a single line.

4 Glittering Mazes

Description: You’ve become trapped in an enormous hedge maze. Luckily, a talking crow has perched on your shoulder and promises to lead you out. The only complication is that the crow insists on picking up colorful stones that happen to be along the path, and insists on doing so in a particular order: alternating between rubies and emeralds.

As you navigate the maze, if you encounter an emerald in the path, you can only move past it if you have most recently moved past a ruby. If you encounter a ruby, you can only move past it if you have most recently moved past an emerald. When you first enter the maze however, you can traverse either a ruby or an emerald. But after that traversal you must maintain the proper order. Write a program to determine the shortest path that the crow will take you down.

Input: Read from `stdin` a representation of a maze. The first line will be a pair of space-separated integers, h w, representing the dimensions of the maze, rows by columns. The next line, x y, will be the coordinate of the entry point to the maze (coordinates are index from zero, and from the top left). The next h lines will represent the maze. Each line will be of length w, and will consist of the characters `#`, `S`, `G`, `R`, and `E`, where `#` represents a wall, `S` represents the start, `G` represents the exit, `R` represents a ruby lying in the maze path, and `E` represents an emerald lying in the maze path. Space characters represent a space through which you can move freely.

Output: Write to `stdout` the shortest solution to the maze, represented by the `X` character. The start point `S`, the exit `G`, each ruby `R` and emerald `E`, and every space should be replaced with an `X` provided a path to escape the maze exists. If no possible path exists, print `NO SOLUTION` instead.

```
Sample Input:
7 11
1 1
###########
#S###G   E#
# ### ### #
#R   R   R#
# ### ### #
#E   R   E#
###########

Sample Output:
###########
#X###XXXXX#
#X### ###X#
#X   R   X#
#X### ###X#
#XXXXXXXXX#
###########

Sample Input:
7 11
1 1
###########
#S###G   R#
# ### ### #
#R   R   R#
# ### ### #
#E   R   E#
###########

Sample Output:
NO SOLUTION
```

5 Cardus Belli

Description: The classic card game 'War' has simple rules: each player begins with half a deck of shuffled playing cards, and every round of play, the players compare the top cards of their decks. This comparison of two cards is referred to as a *Battle*. Whichever player has the highest card wins the round, and places both of the shown cards on the bottom of their deck. If however, there is a tie, referred to as a *War*, a new *War Chest* is created. This chest starts with the two revealed cards, and then each player puts their next three top cards onto the chest. The players then reveal their next top card. If player one wins they put the two revealed cards and the full war chest onto the bottom of their deck. If player two wins, they get the cards and the chest. If they tie again, a new war begins, with the two revealed cards added to the war chest. Whichever player collects all the cards wins. Given a set of cards, you will simulate a game between two players.

Conventions: Variants of the game exist based on the order that cards are laid. Follow the conventions here to simulate our game. First, imagine there is no difference between face-up or face-down with the cards, that is, the cards have their values printed on both sides. When a player wins a Battle, they always place their card on the bottom of their deck first, followed by the losing player's card.

When a War begins, the war chest is always initialized with player one's card on top of player two's. When the players take their top three cards, the cards are taken as one collection, that is, the top card of a player's deck is the top card of the collection of three, and the third card is the bottom card of the collection of three. When they place their collections of three on top of the war chest, player one's three cards go on top of player two's three cards, and these two piles go on top of the current chest. When a player wins a War, they again place their card on the bottom of their deck first, followed by the losing card, and then the war chest. If another tie occurs in a War, the new war chest consists of player one's revealed card on top, then player two's card, then the rest of the war chest.

If one player does not have at least 4 cards at the beginning of a War, they lose by default, and the winning player gets their remaining cards and the war chest added to their deck. Below we have an example on a subset of cards.

```
Sample Game:
P1: T 6 K 1 2 3 9
P2: 5 J K 4 5 T A

P1: 6 K 1 2 3 9 T 5
P2: J K 4 5 T A

P1: K 1 2 3 9 T 5
P2: K 4 5 T A J 6
```

```
P1: 9 T 5        WAR CHEST: 1 2 3 4 5 T K K
P2: A J 6

P1: T 5
P2: J 6 A 9 1 2 3 4 5 T K K

P1: 5
P2: 6 A 9 1 2 3 4 5 T K K J T

P1:
P2: A 9 1 2 3 4 5 T K K J T 6 5
```

INPUT: Read from `stdin` two lines each consisting of 26 characters, that will consist of the the values 2 through 9, and the additional cards `T` for 10, `J` for the Jack, `Q` for the Queen, `K` for the King, and `A` for the Ace. Suit is irrelevant for our game, and Aces count as the highest card.

OUTPUT: Write to `stdout` which player wins, `PLAYER 1` or `PLAYER 2`.

```
Sample Input:
K A 3 6 2 9 J 3 Q 7 3 5 8 K 6 7 4 Q T 9 9 8 A 2 5 T
K 3 4 6 2 9 Q J K Q 4 5 7 8 7 4 5 J J A T T 2 A 6 8

Sample Output:
PLAYER 2
```

6 Hydrotopia

Description: You live in Hydrotopia, where everything is powered by hydroelectric dams. As Hydrotopia's chief engineer, you are tasked with plotting the dynamics of the moving water. Water is collected in large, vertical tanks, and flows through generators, where it then settles in the next tank. You want to write a program that determines how many pumps your generators will perform:

- A generator will make a pump when a water block flows through it.
- Water flowing through a generator cannot move to a position higher than the generator itself.
- All water flows through generators left to right.
- All water must completely settle before it can flow through any generators again. That is, if given:

  ```
  X X
  XW1
  XXX 2 X
  XXX X X
  XXX 3 X
  XXXXXXX
  ```

  Generator 2 would not be the one to pump water flowing through 1, it must settle to the bottom of the tank first, so generator 3 would pump it to the rightmost tank.
- Water must be contained within the walls of the tank; it cannot flow over the walls of a tank into another section.

Input: Read from `stdin` a pair of space-separated integers representing the dimensions of the tank complex, *h w*, the height followed by the width. The next *h* lines will all have length *w*, and will consist of the characters `X`, `G`, `W`, and the space character. `X` denotes the walls of the tanks, `G` denotes a generator, `W` denotes a block of water, and the space character (when within the walls of a tank) denotes space that a water block can flow into.

Output: Write to `stdout` the total number of total pumps your generators will make.

```
Sample Input:
7 7
XWX
XWX
XWX
XWG X X
XWG X X
XXX G X
XXXXXXX

Sample Output:
5
```

7 Sneaky Images

DESCRIPTION: Steganography is the practice of hiding information in images or video. The IPC administrators have devised a clever way to hide text inside of a string of RGB values. In a stream of RGB values, we have encoded individual digits of length-8 binary numbers into the last digit of 3-digit RGB values. These length-8 binary numbers correspond to ASCII values for individual characters[2], remember in ASCII, the characters `A` through `Z` start from decimal 65 and end with 90, and `a` through `z` start from decimal 97 and end with 122. The space character has a binary value of `00100000` with a decimal value of 32. These binary digits are always contiguous in the stream, that is, when decoding a single character, you should be able to read off a consecutive sequence of eight `0`'s and `1`'s without any other non-binary digit in between. Scan through the input, find the binary numbers, and convert them to their ASCII character equivalents to print the message.

INPUT: Read from `stdin` a single line of space separated, 3-digit RGB values.

OUTPUT: Write to `stdout` the decoded binary message. You are guaranteed that only lowercase `a-z`, uppercase `A-Z`, and the space character have been encoded. Note that in the example below, the first contiguous string of binary digits starts at `070` and runs through to `121` inclusively. Although this sample is shown on two lines, input will always be on a single line.

```
Sample input:
250 113 074 190 011 129 130 014 148 070 191 170 100 050 230 160 121
    240 031 140 160 200 120 051 081 060 091 140 230 241 201 070 171

Sample output:
ACM
```

[2]In case you need to remember your binary/decimal ASCII values, we've linked them here `http://sticksandstones.kstrom.com/appen.html`

8 The Towers of Euclid

Description: Your Discrete Math professor has two great loves: prime numbers[3] and the Towers of Hanoi. They decided they would create one great problem from the two: The Towers of Euclid. In this problem, you are given three stacks, arranged side by side. The first two stacks contain stacked integers. The problem is to rearrange the integers among the three stacks such that when reading horizontally, you read prime numbers. Consider the initial and final arrangement of stacks below:

```
               Before:            After:
Row 4    | |   | |   | |      | |   | |   | |
Row 3    | |   | |   | |      | |   | |   | |
Row 2    |7|   |3|   | |      | |   | |   | |
Row 1    |4|   |6|   | |      |4|   |6|   |3|
Row 0    |9|   |4|   | |      |9|   |4|   |7|

Stack     0     1     2        0     1     2
```

A valid move in the Towers of Euclid is to take an integer from the top of one stack, and move it to the top of another. So in the example above, we reach the final configuration by first moving the 7 from stack 0 to stack 2, and then moving the 3 from stack 1 to stack 2. Lastly, reading the stacks horizontally, 947 and 463 are both prime numbers, so the arrangement is valid. However, given two initial stacks, certain prime arrangements can be arrived to using minimal valid moves. Write a program that when given two initial stacks of integers finds the final arrangement yielding a prime reading using the minimal number of valid moves.

Note: Although technically there could be several arrangements that could be arrived to in an equal number of minimal moves, you are guaranteed for the given input only one arrangement is optimal.

Input: Read from `stdin` two lines of space-separated integers. Each integer will have a value between 1 and 9. The first line of input will represent the first stack, read from the bottom up, that is, the first integer of the line represents the bottom of the stack, the last integer of the line represents the top of the stack. The second line of input represents the second stack, read off in the same manner.

Output: Write to `stdout` the final ordering of the stacks such that a horizontal reading produces only prime numbers, and such that this ordering was arrived to using a minimal number of valid moves. Write your output a stack at a time, separating each stack with a newline, with the bottom-most integer of the stack first.

[3] The sequence of primes is all positive integers that are only divisible by 1 and the number itself, starting from 2 and continuing infinitely

```
Sample Input:
9 4 7
4 6 3

Sample Output:
9 4
4 6
7 3
```

9 N-PRINCESSES

DESCRIPTION: Programmatically, the frontier of chess was conquered long ago by IBM's Deep Blue computer. Most recently, the ancient Chinese game of Go was been cracked by Google's AlphaGo. Humans though can still amuse themselves with variants on the classics. The N-Queens problem from Computer Science describes arrangements of the Queen[4] on a standard 8 by 8 chess board such that no Queen on the board threatens[5] another.

One alternative Chess piece is the Princess. This piece can move in the L-shape of a Knight, and any distance down diagonals. For this problem, you will be given an n by n board, and you must calculate the total number of valid arrangements such that when the n princesses are placed, none of them threatens any of the others. As examples, single valid arrangements when n is 5 and 8 are given below:

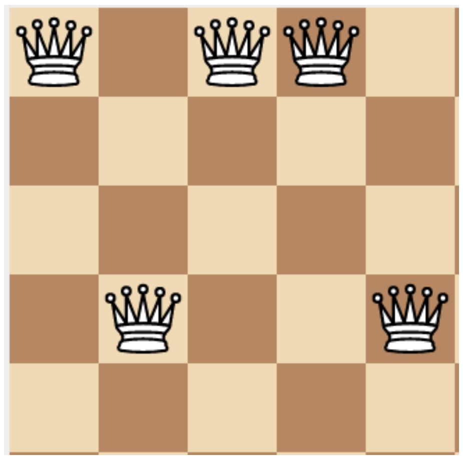

Figure 9.1: When n is five

[4]A Queen is a piece that can move any number of un-blocked spaces horizontally, vertically, and diagonally
[5]In Chess, one piece threatens another if the piece can make a valid move to the cell occupied by another piece

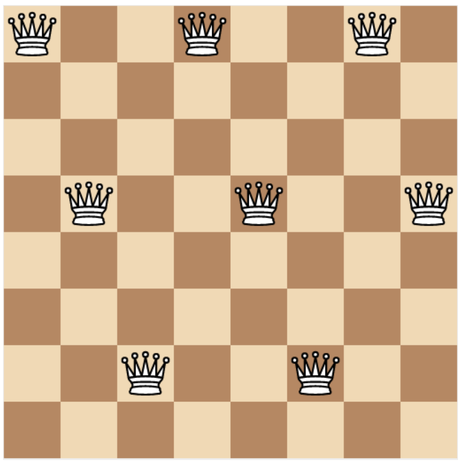

Figure 9.2: When n is eight

INPUT: Read from `stdin` a single integer$n$ representing the length and width of the chess board, and the number of princesses to place.

OUTPUT: Write to `stdout` a single integer representing the number of valid arrangements of the n princesses.

```
Sample Input:
2

Sample Output:
4
```

10 Rebar Barons

DESCRIPTION: A collection of warring states has rendered an enormous plain one giant irradiated quagmire. The problem, however, is that this has left pairs of allied city states separated by the no-man's land. Each of the city states wants to build elevated highways over the no-man's land such that they are connected to their allied city-state. The catch is that none of these elevated highways can cross another, else the construction of highways devolve into resuming hostilities. As a good war opportunist who happens to own the largest concrete mixing company in the region, you want to determine what is the maximum number of elevated highways that can be built such that none overlap. See the example below:

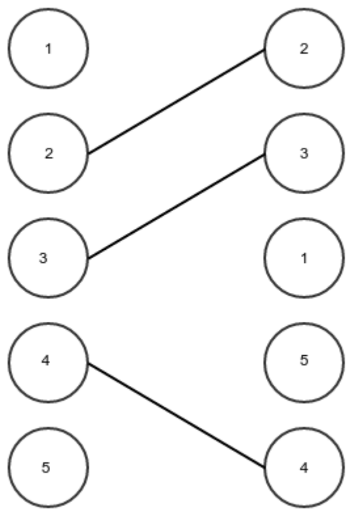

Figure 10.1: A valid arrangement of highways

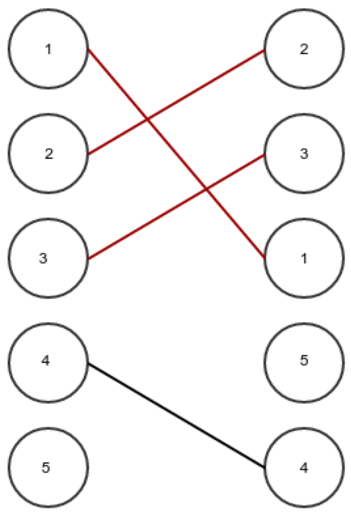

Figure 10.2: An invalid arrangement of highways, see that the top three cross

INPUT: Read from `stdin` two lines of space-separated integers. The first line of integers represents one side of the no-man's land, where the cities are arranged in ascending vertical order. The next line of integers represents the opposite side of the no-man's land, where the numbers represent the ally of the city-state with the same number. These cities are also arranged vertically, but in an unsorted ordering.

OUTPUT: Write to `stdout` the maximum number of elevated highways that can be constructed to connect pairs of allied city-states such that no highway overlaps another.

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

Sample Output:
3
```