CSE 109 — Competitive Programming

Notes taken by Nolan Chai

Spring 2023

Contents

Preface

These are a collection of notes personally taken by me, specifically for readings and allotted content for UCSD's CSE 109 Competitive Programming taken in Spring 2023. These notes are planned to extend beyond CSE 109 as I hope to get better at competitive programming. Additionally, these are not endorsed by the lecturers nor staff, and I have modified them (often significantly) over random periods of time. They may become nowhere near accurate representations of what was actually lectured, or written in the books, and are simply to aid in my own understanding. In particular, all errors are almost surely mine.

These notes differ from others in that these are not lecture based, but rather, competition and approach based. I'm personally new to competitive programming, and much of these notes will contain logical errors / fallacies that I will try my best to fix.

My other notes are available **here**.

1 Problem Set 0

1.1 POJ 1004: Financial Management

Description: Larry graduated this year and finally has a job. He's making a lot of money, but somehow never seems to have enough. Larry has decided that he needs to grab hold of his financial portfolio and solve his financing problems. The first step is to figure out what's been going on with his money. Larry has his bank account statements and wants to see how much money he has. Help Larry by writing a program to take his closing balance from each of the past twelve months and calculate his average account balance.

Input: The input will be twelve lines. Each line will contain the closing balance of his bank account for a particular month. Each number will be positive and displayed to the penny. No dollar sign will be included.

Output: The output will be a single number, the average (mean) of the closing balances for the twelve months. It will be rounded to the nearest penny, preceded immediately by a dollar sign, and followed by the end-of-line. There will be no other spaces or characters in the output.

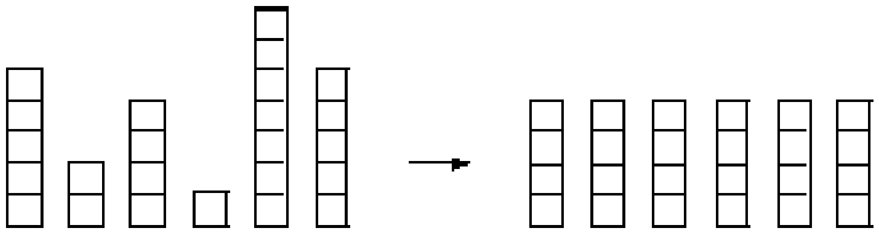
Approach: At a high level, I approach this by first seeing that we can simply loop over the input twelve times, each time adding to the sum and dividing by twelve. I know magic numbers are bad, but this specific case seems alright.

```
Set variables .
For i from 0 to 12:
    Input month
    sum += month
Return string with sum/12
```

We can see that the runtime is trivially $\Theta(n)$, or, more accurately: $\Theta(n + 1)$ assuming that initializing our variables do not take any time.

1.2 POJ 1477: Box of Bricks ☆

Description: Little Bob likes playing with his box of bricks. He puts the bricks one upon another and builds stacks of different height. ”Look, I’ve built a wall!”, he tells his older sister Alice. ”Nah, you should make all stacks the same height. Then you would have a real wall.”, she retorts. After a little con- sideration, Bob sees that she is right. So he sets out to rearrange the bricks, one by one, such that all stacks are the same height afterwards. But since Bob is lazy he wants to do this with the minimum number of bricks moved. Can you help?



Input: The input consists of several data sets. Each set begins with a line containing the number n of stacks Bob has built. The next line contains n numbers, the heights hi of the n stacks. You may assume $1 \leq n \leq 50$ and $1 \leq hi \leq 100$.

The total number of bricks will be divisible by the number of stacks. Thus, it is always possible to rearrange the bricks such that all stacks have the same height.

The input is terminated by a set starting with n = 0. This set should not be processed.

Output: For each set, first print the number of the set, as shown in the sample output. Then print the line ”The minimum number of moves is k.”, where k is the minimum number of bricks that have to be moved in order to make all the stacks the same height.
Output a blank line after each set.

Sample input:

```
6
5 2 4 1 7 5
0
```

Sample output:

```
5
```

Approach: Firstly, we need to know what exactly the algorithm is doing. To make all the stacks the same height, we essentially just find the average value per stack over the number of stacks. In our sample, we have 4 to be the height of each stack. To get each stack to the intended height, we either remove or add such that it reaches the average, being $\mid height - average \mid$, as order does not matter. Then, we divide the sum of these values by two, as our actions are repeated twice (we move blocks from one to another).

Therefore, we have the following algorithm:

```
Set variables .
For each set:
    set++
    if (input = 0)
        break and return
    else {
        moves = 0
        average = 0
        for each brick in stack:
            average += brick
        average = brick / stack_length
        for each brick in stack:
            moves += abs(brick − average)
        moves = moves / 2
    }
    Print out set and moves
```

We can see from the above that we have an approximate runtime of $\Theta(n)$, or more specifically, $\Theta(2n + 3)$ assuming that our variables are initialized correctly.

1.3 POJ 3030: Nasty Hacks

Description: You are the CEO of Nasty Hacks Inc., a company that creates small pieces of malicious software which teenagers may use to fool their friends. The company has just finished their first product and it is time to sell it. You want to make as much money as possible and consider advertising in order to increase sales. You get an analyst to predict the expected revenue, both with and without advertising. You now want to make a decision as to whether you should advertise or not, given the expected revenues.

Input: The input consists of n cases, and the first line consists of one positive integer giving n. The next n lines each contain 3 integers, r, e and c. The first, r, is the expected revenue if you do not advertise, the second, e, is the expected revenue if you do advertise, and the third, c, is the cost of advertising. You can assume that the input will follow these restrictions: $-106 \leq r, e \leq 106$ and $0 \leq c \leq 106$.

Output: Output one line for each test case: "advertise", "do not advertise" or "does not matter", presenting whether it is most profitable to advertise or not, or whether it does not make any difference.

Sample Input:

```
3
0 100 70
100 130 30
−100 −70 40
```

Sample Output:

```
advertise
does not matter
do not advertise
```

Approach: First, we make note that the algorithm takes in r, e, c for a total of n times. Then, we can make a mathematical examination of what advertise, does not advertise, and do not advertise really mean. We are essentially comparing two values: r and $e - c$. Therefore, this problem can be trivially solved by looping over this comparison per line like so:

```
Set variables .
For each line :
    If (r = (e − c))
        Print "does not matter"
    Elif (r > (e − c))
        Print "do not advertise"
    Else
        Print " advertise"
```

As seen above, we have a runtime of $\Theta(n)$. However, **do keep in mind that issues may arise when trying to run loops this way along with the way inputs are read in C++.** In C++, we will need an additional while loop to continue while inputs are being passed, which will result in the runtime being $\Theta(n^2)$.

1.4 POJ 3096: Surprising Strings ☆

Description: The D-*pairs* of a string of letters are the ordered pairs of letters that are distance D from each other. A string is D-*unique* if all of its D-pairs are different. A string is *surprising* if it is D-unique for every possible distance D.

Consider the string ZGBG. Its 0-pairs are ZG, GB, and BG. Since these three pairs are all different, ZGBG is 0-unique. Similarly, the 1-pairs of ZGBG are ZB and GG, and since these two pairs are different, ZGBG is 1-unique. Finally, the only 2-pair of ZGBG is ZG, so ZGBG is 2-unique. Thus ZGBG is surprising. (Note that the fact that ZG is both a 0-pair and a 2-pair of ZGBG is irrelevant, because 0 and 2 are different distances.)

Input: The input consists of one or more nonempty strings of at most 79 uppercase letters, each string on a line by itself, followed by a line containing only an asterisk that signals the end of the input.

Sample input:

```
ZGBG
X
EE
AAB
AABA
AABB
BCBABCC
*
```

Sample output:

```
ZGBG is surprising.
X is surprising.
EE is surprising.
AAB is surprising.
AABA is surprising.
AABB is NOT surprising.
BCBABCC is NOT surprising.
```

Approach: At a high level, the obvious solution would be to place the string into a character array and linearly search each time. However, the issue here is how we search in our implementation. My algorithm loops over each string in the input, and for each character in the string, will loop per pair value to store all possible pairs in an array and check to see if they are all unique using a boolean 2D array. We increment the pair value per instance. Since the maximum distance of a pair is the length of the string minus two, we compare and then either return surprising or not surprising.

Our algorithm would be as follows:

```
For string in input:
    If string = *
        Terminate
    pair = 0
    While loop:
        array = []
        pair++
        For character in string :
            if (pair + char index) <= len(string)
                string_pair = char[index] + char[pair+index]
        Initialize a boolean array for array
        For each pair in array:
            Mark true in bool_array if unique
            If already explored, then break
    If pair = (len( string - 2))
        Print surprising
    Else
        Print not surprising
```

As we can see above, the runtime of our algorithm is $\Theta(n^3)$. This is a naive approach, and we can probably get this down to $\Theta(n^2)$.

1.5 POJ 3852: String LD ☆

Description: Stringld (left delete) is a function that gets a string and deletes its leftmost character (for instance Stringld("acm") returns "cm").

You are given a list of distinct words, and at each step, we apply stringld on every word in the list. Write a program that determines the number of steps that can be applied until at least one of the conditions become true:

1. A word becomes empty string, or
2. a duplicate word is generated.

For example, having the list of words aab, abac, and caac, applying the function on the input for the first time results in ab, bac, and aac. For the second time, we get b, ac, and ac. Since in the second step, we have two ac strings, the condition 2 is true, and the output of your program should be 1. Note that we do not count the last step that has resulted in duplicate string. More examples are found in the sample input and output section.

Input: There are multiple test cases in the input. The first line of each test case is $n(1 \leq n \leq 100)$, the number of words.Each of the next n lines contains a string of at most 100 lower case characters.The input terminates with a line containing 0.

Output: For each test case, write a single line containing the maximum number of stringld we can call.

Sample Input:

```
4
aaba
aaca
baabcd
dcba
3
aaa
bbbb
ccccc
0
```

Sample Output:

```
1
2
```