UCF “Practice” Local Contest — Aug 25, 2018

Fold the Paper Nicely

filename: `fold`
(*Difficulty Level:* `Easy`)

Dr. Orooji has a daily calendar (365 pages) on his desk. Every morning, he tears off one page and, as he is reading the notes on the next page, he folds (out of habit) the sheet in his hand. Dr. O noticed that he always folds the sheet (a rectangular paper) along the longer side, e.g., if one side is 80 and the other side is 60, he would fold along 80; this will make the paper of size 40 and 60; if he folds it again, he would fold along 60 since that’s the longer side now.

The Problem:

Given a rectangular piece of paper and how many times Dr. O folds it, you are to determine the final sizes. When folding a side with an odd length, the fraction is ignored after folding, e.g., if a side is 7, it will become 3 after folding.

The Input:

The input consists of one line, containing three positive integers (each ≤ 10000), the first two providing the rectangle sides and the third providing the number of folds.

The Output:

Print the final values for the rectangle (larger side of the final values first).

Sample Input	**Sample Output**
`60 51 4`	`15 12`
`3 2 50`	`0 0`
`3 2 1`	`2 1`
`5 800 2`	`200 5`

Explanation for the first Sample Input/Output: the rectangle starts with sides `{60, 51}` and successively becomes `{30, 51}, {30, 25}, {15, 25}, {15, 12}`.

Explanation for the last Sample Input/Output: the rectangle starts with sides `{5, 800}` and successively becomes `{5, 400}, {5, 200}`.

Call Me Maybe

filename: `callme`

(*Difficulty Level:* `Medium`)

The popular song, “Call Me Maybe” by Carly Rae Jepson, has been spoofed with great comedic success on the internet recently. In particular, one of the YouTube videos splices words from different speeches by President Obama to string together the lyrics of the song. You have decided that this idea is comic gold. Given a library of a particular person’s speeches as well as the lyrics to a song, you want to spoof them by writing a program to automatically check the text of all of the speeches to make the appropriate word substitutions from the speeches to the song lyrics. For example, the first line of “Call Me Maybe” is “I threw a wish in the well.” Perhaps “I” appears as the 10th word in speech number 4, “threw” appears as the 54th word in speech number 12, etc. Of course, if the speaker never said “wish” in any of his speeches, then the spoof is not possible. With your program, you can spoof almost any song with any celebrity and get rich!

The Problem:

Given a list of speeches by an individual, as well as the text of a song to spoof by pulling words from those speeches, determine if the spoof is possible and, if so, give a listing of which words in which speeches to use as substitutes.

In order to obtain a unique answer, you must cycle through each occurrence of a particular word in the set of speeches, in order by speech and word. Thus, if the word, “I” appears five times in the song and appears in the following four locations: (1) speech 2, word 7, (2) speech 4, word 12, (3) speech 4, word 50, and (4) speech 5, word 1, then the first substitution would come from speech 2, the second substitution would be the 12th word of speech 4, the third substitution would be the 50th word of speech 4, the fourth substitution would be from speech 5 and the last substitution would be from speech 2 again. As this example illustrates, a word does not have to appear in speeches as many times as it appears in a song. It does, however, have to appear at least once. Note also that a word may appear several times in a speech and these occurrences are used in order and before using the first occurrence of that word in the next speech.

The Input:

The input consists of several lines and provides a spoof to potentially create. The first input line contains an integer, *m* $(1 \le m \le 100)$, representing the number of speeches for which there is text. Each of the speeches follows. Each speech is stored on a single line. The first token on each of these lines will be an integer, *t* $(1 \le t \le 1000)$, representing the number of words in the speech. Each of these words will consist of 1-20 lowercase letters. Words are separated by spaces and there are no other characters on these lines.

The input speeches are followed by a single line storing the lyrics of the song to be spoofed. The first token of this line will be an integer, *l* $(1 \le l \le 50)$, indicating the number of words in the lyrics. Each of these words will consist of 1-20 lowercase letters. The words in the lyrics are separated by spaces and there are no other characters on this line.

The Output:

If no spoof is possible, then simply output the string `NOT POSSIBLE`.

If a spoof is possible, output *w* lines, where *w* is the number of words in the song to spoof. Each line will contain two integers (separated by one space): the speech number and the word number, both of which are numbered, starting at 1.

Sample Input #1:

```
3
8 tom brady threw a pass in the pocket
5 i sung very well today
8 the genie wishes i had one wish left
7 i threw a wish in the well
```

Sample Output #1:

```
2 1
1 3
1 4
3 7
1 6
1 7
2 4
```

Sample Input #2:

```
1
15 let it be told to the future world that we came forth to meet it
8 do not ask me i will never tell
```

Sample Output #2:

```
NOT POSSIBLE
```

Rummy Score

filename: `rummy`

(*Difficulty Level:* `Medium`)

Anya (Arup’s younger daughter) plays Rummy with a modified deck of playing cards. The card values are 1 through 13 (1 does not serve the dual purpose of 1 and 14; it is only 1) and there are no suits. Also, a deck may contain any number of each card value (unlike a standard deck of cards which has exactly four of each).

The Problem:

Given the seven cards in a Rummy hand (held by Anya), you are to determine the points lost by the hand. In Rummy, you group three (or more) cards together if they have the same value or they have consecutive values. Examples of a group include {4, 4, 4}, {6, 6, 6, 6, 6}, {9, 10, 11}, and {1, 2, 3, 4}. The “points lost by a hand” is the sum of values for cards that are not in a group.

When computing the points lost by a hand, you group the cards to minimize the loss, i.e., you want the smallest total sum of values for the ungrouped cards. Note that a given card could possibly belong to more than one group; you need to pick the group that minimizes the total points lost. For example, given the seven cards {2, 2, 2, 3, 4, 8, 10}, the grouping {2, 3, 4} is better than the grouping {2, 2, 2}. Note also that a card can be used in only one group, e.g., from the Rummy hand {2, 2, 2, 3, 4, 8, 10}, we cannot create the two groups {2, 2, 2} and {2, 3, 4} since there are not a total of four cards with a value of 2 in the hand.

The Input:

The input consists of one line, containing seven integers, each integer between 1 and 13, inclusive (the values are separated by a single space).

The Output:

Print the points lost by the input hand.

(Sample Input/Output on the next page)

Sample Input **Sample Output**

5 2 2 6 2 10 4	10
11 11 11 11 1 2 3	0
11 2 3 4 5 13 1	24
4 5 6 6 6 6 13	13
2 2 2 3 4 8 10	22
12 13 1 4 4 4 4	26
10 10 10 8 9 8 9	10

Boots Exchange

filename: `boots`
(*Difficulty Level:* `Easy`)

A boot shop has received a shipment from the factory consisting of ***N*** left boots and ***N*** right boots. Each boot has some integer size, and a left and right boot will form a proper pair if they have equal sizes. Each boot can only belong to a single pair. The employees of the boot store want to create ***N*** proper pairs of boots. Fortunately, the factory has offered to exchange any number of boots in the shipment with new boots of different sizes.

The Problem:

Given the size of the left boots and right boots, your task is to determine the least number of boots that need to be exchanged.

The Input:

The first line of input contains the integer ***N***. Next line contains ***N*** positive integers separated by a single space. These are the size of the left boots. Next line contains ***N*** positive integers separated by a single space. These are the size of the right boots. ***N*** is between 1 and 100 inclusive, and each boot size is between 1 and 99,999 inclusive.

The Output:

Output the least number of boots that need to be exchanged.

Sample Input	**Sample Output**

`3`
`1 3 1`
`2 1 3`
	`1`

`2`
`1 3`
`2 2`
	`2`

`4`
`1 2 3 4`
`2 4 1 3`
	`0`

UCF “Practice” Local Contest — Aug 25, 2018

Git Repository

filename: `git`
(*Difficulty Level:* `Hard`)

From Wikipedia: **Git** is a “version control system” for tracking changes in “computer files” and coordinating work on those files among multiple people. It is primarily used for “source code management” in software development, but it can be used to keep track of changes in any set of files. As with most other distributed version control systems, and unlike most client–server systems, every Git “directory” on every computer is a full-fledged “repository” with complete history and full version tracking abilities, independent of network access or a central server.

We will explore a simplified version of Git. This version will contain a logbook of various commits to the repository in the order they occur.

Command	**Description**
new <versionid>	Create a new repository (version) with the given version id. This branch (version) is empty containing no code.
commit <versionid> <numlines>	Commit *numlines* new lines to the version with the given version id. That is, the version is increased in size by *numlines* of new code.
branch <versionid> <newversion>	Take the current version of the code with *versionid* and start a new branch (version) with id *newversion*. The new version will start with the same code as the parent. Note that any code added to the parent after a child is created will not be part of the child’s code. But, all the code added to the parent before the child is created is part of the child.

The Problem:

You would like to read the logbook and answer some queries about the changes in source code. The logbook is numbered starting from 1 for each command. Each query contains two logbook lines *a* and *b* as well as a version id. You would like to know the amount that the branch (version) given has grown from commands *a* to *b,* inclusive. Note that this value must include any past versions (ancestors) of that version up to the time that the child was created.

Note again that the code is copied from the old version (parent) to the new version (child) at the time of the branch. New commits to the parent after the child is created do not affect the child.

The Input:

The first line of input contains an integer ***n*** ($1 \leq n \leq 10^5$), representing the number of lines in the logbook. The next ***n*** lines each contain commands from the logbook. No version id will be referenced in a command before either being created with new or branched from a previous version id. No version id will be created more than once. At least 1 and no more than 1000 lines will be committed in a single commit.

The next input line contains an integer ***q*** ($1 \leq q \leq 10^5$), representing the number of queries about code in the logbook. The following ***q*** lines each contain two integers ***a*** and ***b*** ($1 \leq a \leq b \leq n$) representing the start and end lines (inclusive) for the query and a ***version id.*** These values are separated by spaces. All version ids are strings of at most 10 characters consisting of digits and letters (both upper and lower case). Note that the version ids are case sensitive.

The Output:

For each query, output a single integer representing the number of lines added to the queried version within the list of commands specified by the query. If the version and all of its predecessors did not exist at that time (or at all), output 0.

(Sample Input/Output on the next page)

Sample Input

```
15
new aaaa
commit aaaa 100
new bbbb
commit bbbb 11
branch aaaa a1
commit aaaa 200
commit a1 4
branch a1 a2
branch a2 a3
commit a3 1000
commit a2 7
branch bbbb b2
commit b2 10
commit b2 7
commit a3 1000
10
1 2 aaaa
2 5 a1
2 4 bbbb
1 11 a2
1 11 a3
9 15 a3
9 15 a2
2 2 a3
2 7 a3
1 15 zzzz
```

Sample Output

```
100
100
11
111
1104
2000
7
100
104
0
```

Explanation for Query #8 (`2 2 a3`**):** version `a3` is a descendent of version `aaaa`. The start and end line for this query is `2` and only version `aaaa` existed at that time with 100 lines of code. So, the output is `100`.

Explanation for Query #9 (`2 7 a3`**):** version `a3` is a descendent of version `a1` (which is a descendent of version `aaaa`). The start and end line for this query is “`2 7`” and version `a1` existed at that time with 104 lines of code. So, the output is `104`.

UCF “Practice” Local Contest — Aug 25, 2018

Magnetic Strip

filename: `strip`
(*Difficulty Level:* `Hard`)

While playing with the magnetic strips that came with his mini whiteboard, Gabe realized that the strips were made from several equal length bands of positive or negative magnets with no particular order. The strips are only magnetic on one side and Gabe enjoys folding them and trying to hold them together by matching positive bands with negative bands.

The Problem:

In order to make the strongest bond, Gabe would like to know the largest contiguous section on a strip that can be perfectly paired with another section on the same strip. Gabe will fold the strip exactly once to pair the bands. Folding can be done both in between bands and on the center of a band. More specifically, if the folding occurs in between band *k* and band *k+1*, after folding band *k+1* will be paired with band *k*, band *k+2* will be paired with band *k-1*, etc. If folding occurs on the center of band *k*, after folding band *k+1* will be paired with band *k-1*, band *k+2* will be paired with band *k-2*, etc.

The Input:

The input consists of one line, a string of ‘+’ and ‘-’ characters denoting positive and negative bands on the magnetic strip. The length of this string is at least 1 and no more than 10^5.

The Output:

On a line by itself, output the maximum length contiguous section that can be paired with another section on the same strip with a single fold. Note that maximum contiguous section can start on any band of the folded strip, i.e., it may not start at the beginning of the strip or at the folded spot.

(Sample Input/Output on the next page)

Sample Input	**Sample Output**
`++`	`0`
`++−+−+−`	`3`
`+−−+++−−−−−+++−−−+++`	`8`
`−+−−+`	`2`

Explanation for the second Sample Input/Output: we fold between band 4 and band 5 so, after folding (flipping) the right-hand side over the left-hand side, we’ll have:

```
 −+−
++−+
```

Explanation for the third Sample Input/Output: we fold between band 10 and band 11 so, after folding (flipping) the right-hand side over the left-hand side, we’ll have:

```
+++−−−+++−
+−−+++−−−−
```

Explanation for the last Sample Input/Output: we fold at the center of band 3 so, after folding (flipping) the right-hand side over the left-hand side, we’ll have:

```
+−*
−+*
```

Note: we have used ‘*’ to indicate the folded band 3.