Problem Solving Workshop #35 Tech Interviews and Competitive Programming Meetup

May 20, 2018 https://www.meetup.com/tech-interviews-and-competitive-programming/

Instructor: Eugene Yarovoi (can be contacted through the group Meetup page above under Organizers)

More practice questions: leetcode.com, glassdoor.com, geeksforgeeks.org
Books: Elements of Programming Interviews, Cracking the Coding Interview
Have questions you want answered? Contact the instructor, or ask on Quora. You can post questions and follow the instructor and other people who write about algorithms.

Try to find optimized solutions, and provide a time and space complexity analysis with every solution.

Problem #1, "Time Traveling Counter"

(i) **[Easy]** Design a class whose purpose is to accumulate counts of Strings. It should have the following methods:

```
int getCount(String str)
void add(String str)
```

`add` inserts one more instance of the given string. `getCount` returns how many times the given string was added. If you try to get the count of a string that was never added, 0 should be returned (it should not be an exception or cause a problem for your code -- this is an expected use case).

(ii) **[Easy-Medium]** Now, you have access to a `getTime()` function that returns a `long` representing the current time (strictly increasing every time you call it). How would you modify your implementation to have the following method signatures instead:

```
int getCount(String str, long timestamp)
void add(String str)
```

`getCount` should now return the count as it would have been at the specified timestamp. If that timestamp is at the same time as when an `add` happens, the `add` is included in the number returned by `getCount`.

For example, if the following sequence of calls is made at the following times:

T = 0 `add("foo")`
T = 2 `add("bar")`
T = 3 `add("foo")`
T = 5 `getCount("foo", 2)` -> returns 1, because the action at time T=3 hadn't happened yet
T = 6 `getCount ("foo", 3)` -> returns 2, because at T = 3 both "foo" have been inserted

(iii) **[Easy-Medium]** Same as (ii), but support the `void reset(String str)` method, which resets the count of `str` at the current time. Concatenating to the above example, if then we have:

T = 7 `reset("foo")`

T = 8 `getCount("foo", 7)` -> returns 0, because the foos were already deleted at T = 7
T = 9 `getCount("foo", 6)` -> returns 2, because the foos had not yet been deleted at T = 6

(iv) **[Medium-Hard]** Same as (iii), but now we wish to allow for ***retroactive corrections*** to the counters by supporting passing a timestamp in `add` and `reset`. So, now there are additional overloaded methods that look like this:

```
void add(String str, long timestamp)
void reset(String str, long timestamp)
```

The effect of these methods should be that, for all `getCount` queries that follow, the data structure should act as though the `add` and `reset` actions had been enacted at the specified timestamps.

Problem #2, "Unique Letter Score"

The unique letter score of a string is the number of letters that appear only once in it. For example, the unique letter score of "wow" is 1 (only "o" is unique) and the unique letter score of "payday" is 2 ("p" and "d" are unique). Given a string S, find the sum of all unique letter scores over all of its possible contiguous substrings. A substring is defined by some unique indices i, j where you take S[i...j] as the substring. In other words, find:

Sum from i=0 to i=S.len-1 (Sum from j=i to j=S.len-1 (UniqueLetterScore(S[i...j])))

(where S[i...j] is taken to be inclusive on both ends)

[Easy] Find a naive solution. You should try seeing if you can find the $O(N^2)$ naive solution (many other naive solutions have worse complexity).
[Medium] Solve in O(N).

Problem #3, "2D brackets" (Hard)

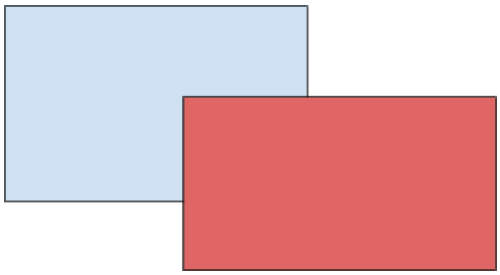
You are probably familiar with the 1-D "valid parentheses" problem:

Given a string such as "()(())", determine if the string is validly parenthesized. A string is said to be validly parenthesized if all the opening and closing parentheses can be matched up in the way you'd normally expect parentheses to behave: a closing parenthesis must be matched to the immediately preceding unmatched opening parenthesis, and all the parentheses must be used. (If you are unfamiliar with this problem, you should solve it first!)

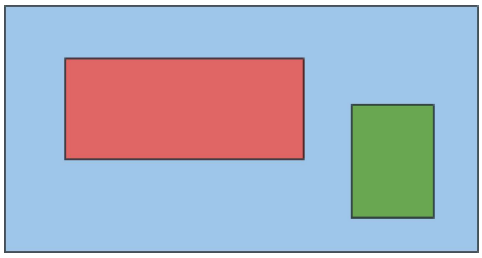
In this problem, we have a 2-D version of this. The rule is that a closing parenthesis has to be matched to an opening parenthesis that is above and to the left of it (similar to how in the 1-D problem, the opening paren has to be to the left), and when matched, it creates a rectangle:

Furthermore, we require that after the matching, the resulting rectangles don't have intersecting edges (the rectangles can overlap, but have to be nested inside each other if so).

In other words, this kind of configuration is not allowed:



But this one is fine:



This is similar to how in 1-D, the parentheses have to be nested inside one another. In 1-D for example, we don't allow this solution

```
(   (   )   )
|-------|
    |-------|
```

And instead require this one:

```
(   (   )   )
|-----------|
   |---|
```

Given a list of 2-D coordinates of opening parentheses and a list of 2-D coordinates of closing parentheses, determine if you can match them all up in pairs following the rules (no rectangles have intersecting edges, and each closing paren is matched to an opening paren that is above and to the left of

it). Output the mapping of opening parentheses to closing parentheses. If there are multiple solutions, output any one, and if there is no possible solution, report this fact.