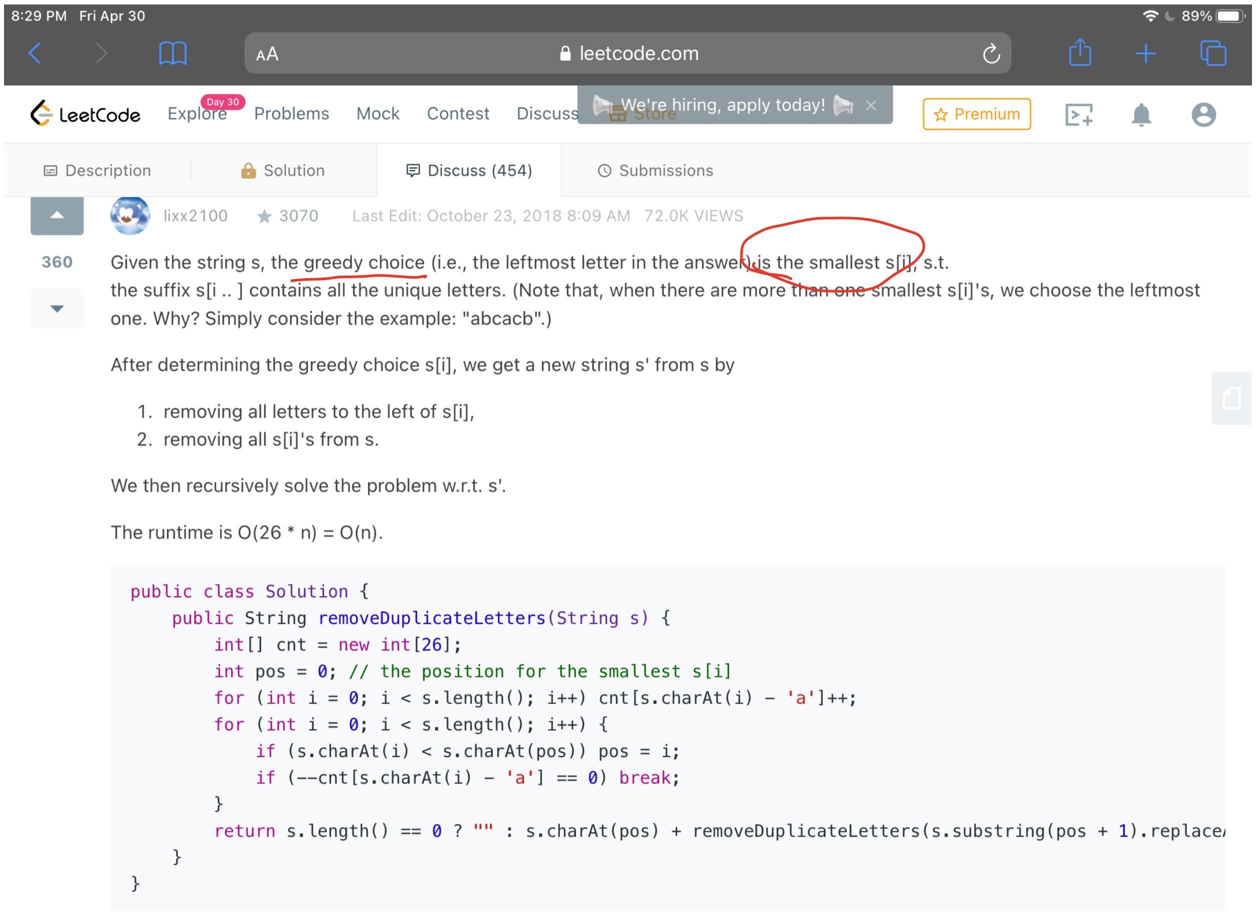


lixx2100 ★ 3070 Last Edit: October 23, 2018 8:09 AM 72.0K VIEWS

Given the string s, the greedy choice (i.e., the leftmost letter in the answer) is the smallest s[i], s.t.
the suffix s[i ..] contains all the unique letters. (Note that, when there are more than one smallest s[i]'s, we choose the leftmost one. Why? Simply consider the example: "abcacb".)

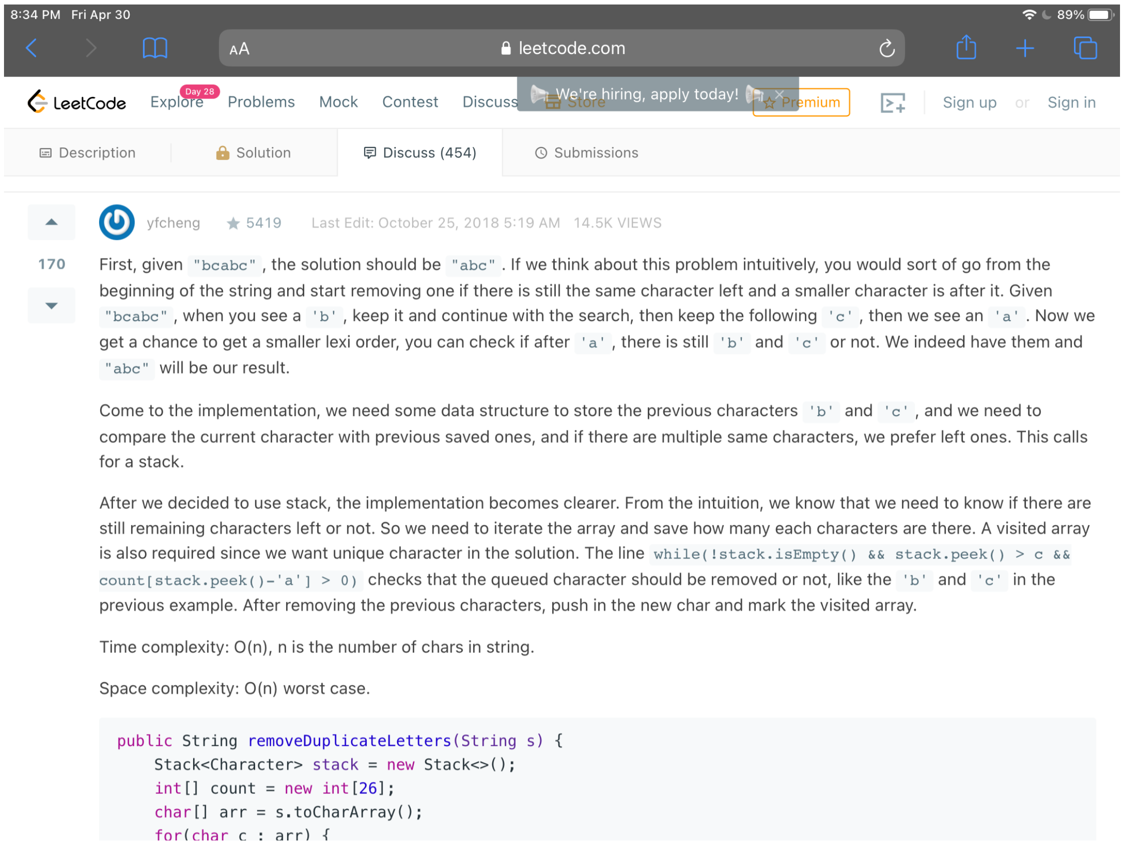
After determining the greedy choice s[i], we get a new string s' from s by

1. removing all letters to the left of s[i],
2. removing all s[i]'s from s.

We then recursively solve the problem w.r.t. s'.

The runtime is O(26 * n) = O(n).

```
public class Solution {
    public String removeDuplicateLetters(String s) {
        int[] cnt = new int[26];
        int pos = 0; // the position for the smallest s[i]
        for (int i = 0; i < s.length(); i++) cnt[s.charAt(i) - 'a']++;
        for (int i = 0; i < s.length(); i++) {
            if (s.charAt(i) < s.charAt(pos)) pos = i;
            if (--cnt[s.charAt(i) - 'a'] == 0) break;
        }
        return s.length() == 0 ? "" : s.charAt(pos) + removeDuplicateLetters(s.substring(pos + 1).replace
    }
}
```


8:34 PM Fri Apr 30 89%

leetcode.com

LeetCode Explore Day 28 Problems Mock Contest Discuss Store We're hiring, apply today! Premium Sign up or Sign in

Description Solution Discuss (454) Submissions

170

yfcheng ★ 5419 Last Edit: October 25, 2018 5:19 AM 14.5K VIEWS

First, given `"bcabc"`, the solution should be `"abc"`. If we think about this problem intuitively, you would sort of go from the beginning of the string and start removing one if there is still the same character left and a smaller character is after it. Given `"bcabc"`, when you see a `'b'`, keep it and continue with the search, then keep the following `'c'`, then we see an `'a'`. Now we get a chance to get a smaller lexi order, you can check if after `'a'`, there is still `'b'` and `'c'` or not. We indeed have them and `"abc"` will be our result.

Come to the implementation, we need some data structure to store the previous characters `'b'` and `'c'`, and we need to compare the current character with previous saved ones, and if there are multiple same characters, we prefer left ones. This calls for a stack.

After we decided to use stack, the implementation becomes clearer. From the intuition, we know that we need to know if there are still remaining characters left or not. So we need to iterate the array and save how many each characters are there. A visited array is also required since we want unique character in the solution. The line `while(!stack.isEmpty() && stack.peek() > c && count[stack.peek()-'a'] > 0)` checks that the queued character should be removed or not, like the `'b'` and `'c'` in the previous example. After removing the previous characters, push in the new char and mark the visited array.

Time complexity: O(n), n is the number of chars in string.

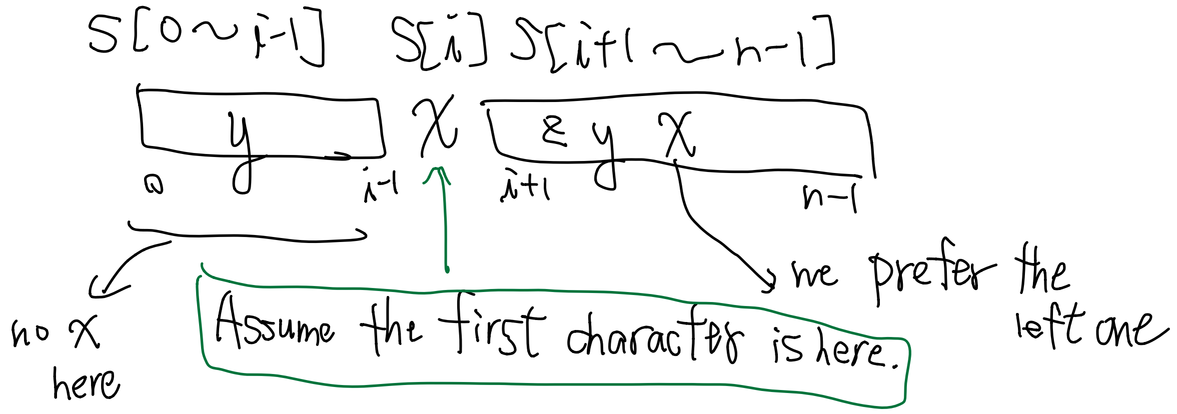
Space complexity: O(n) worst case.

```
public String removeDuplicateLetters(String s) {
    Stack<Character> stack = new Stack<>();
    int[] count = new int[26];
    char[] arr = s.toCharArray();
    for(char c : arr) {
```

I think using the recursive solution to understand stack approach is a bit easier.

Intuition: If searching a word is hard, why not just think of "how to find the first character"?

Of course, we want it as "small" as possible (a<b<c<d....)



It must satisfy $\min(S[0\sim i-1]) > x$,

Observe:

In the stack solution, it will ALWAYS find the leftmost x correctly.

Because it will pop out character $y > x$ if y still has characters after the leftmost x.

After pushing x into the stack,

x will NEVER be popped out again.

($\because$ visited[x] is true)

(And $\because$ Assume there is a character z, that can force x be popped out (i.e. $x > z$) while maintaining z $\square$ contain all unique characters.

This will contradict our assumption that x is the first smallest character)

After the first character is sucessfully found,

it is easy to see

finding the second character of s

$==$ finding the first character

of $\boxed{s[i+1 \sim n-1].\underline{\text{replace}(x, "")}}$

because visit[x] is true,
x will be totally ignored
just as if it does not exist