Previous Next

316. Remove Duplicate Letters

Aug. 19, 2019 | 12K views

Average Rating: 4.64 (25 votes)

Given a string which contains only lowercase letters, remove duplicate letters so that every letter appears once and only once. You must make sure your result is the smallest in lexicographical order among all possible results.

Example 1:

```
Input: "bcabc"
Output: "abc"
```

Example 2:

```
Input: "cbacdcbc"
Output: "acdb"
```

Note: This question is the same as 1081: https://leetcode.com/problems/smallest-subsequence-of-distinct-characters/

Solution

Intuition

First we should make sure we understand what "lexicographical order" means. Comparing strings doesn't work the same way as comparing numbers. Strings are compared from the first character to the last one. Which string is greater depends on the comparison between *the first unequal corresponding character* in the two strings. As a result any string beginning with `a` will always be less than any string beginning with `b`, regardless of the ends of both strings.

Because of this, the optimal solution will have *the smallest characters as early as possible*. We draw two conclusions that provide different methods of solving this problem in $O(N)$:

1. The leftmost letter in our solution will be the smallest letter such that the suffix from that letter contains every other. This is because we know that the solution must have one copy of every letter, and we know that the solution will have the lexicographically smallest leftmost character possible.

   If there are multiple smallest letters, then we pick the leftmost one simply because it gives us more options. We can always eliminate more letters later on, so the optimal solution will always remain in our search space.

2. As we iterate over our string, if character `i` is greater than character `i+1` and another occurrence of character `i` exists later in the string, deleting character `i` will **always** lead to the optimal solution. Characters that come later in the string `i` don't matter in this calculation because `i` is in a more significant spot. Even if character `i+1` isn't the best yet, we can always replace it for a smaller character down the line if possible.

Since we try to remove characters as early as possible, and picking the best letter at each step leads to the best solution, "greedy" should be going off like an alarm.

Approach 1: Greedy - Solving Letter by Letter

Algorithm

We use idea number one from the intuition. In each iteration, we determine leftmost letter in our solution. This will be **the smallest character such that its suffix contains at least one copy of every character in the string**. We determine the rest our answer by recursively calling the function on the suffix we generate from the original string (leftmost letter is removed).

Implementation

Java | Python | Copy

```
from collections import Counter

class Solution:
    def removeDuplicateLetters(self, s: str) -> str:

        # find pos - the index of the leftmost letter in our solution
        # we create a counter and end the iteration once the suffix doesn't have each unique character
        # pos will be the index of the smallest character we encounter before the iteration ends
        c = Counter(s)
        pos = 0
        for i in range(len(s)):
            if s[i] < s[pos]: pos = i
            c[s[i]] -=1
            if c[s[i]] == 0: break
        # our answer is the leftmost letter plus the recursive call on the remainder of the string
```

Note that the code in this section is a translated / commented version of the code in this post originally written by lixx2100.

Complexity Analysis

- Time complexity : $O(N)$. Each recursive call will take $O(N)$. The number of recursive calls is bounded by a constant (26 letters in the alphabet), so we have $O(N) * C = O(N)$.
- Space complexity : $O(N)$. Each time we slice the string we're creating a new one (strings are immutable). The number of slices is bound by a constant, so we have $O(N) * C = O(N)$.

Approach 2: Greedy - Solving with Stack

Algorithm

We use idea number two from the intuition. We will keep a stack to store the solution we have built as we iterate over the string, and we will delete characters off the stack whenever it is possible and it makes our string smaller.

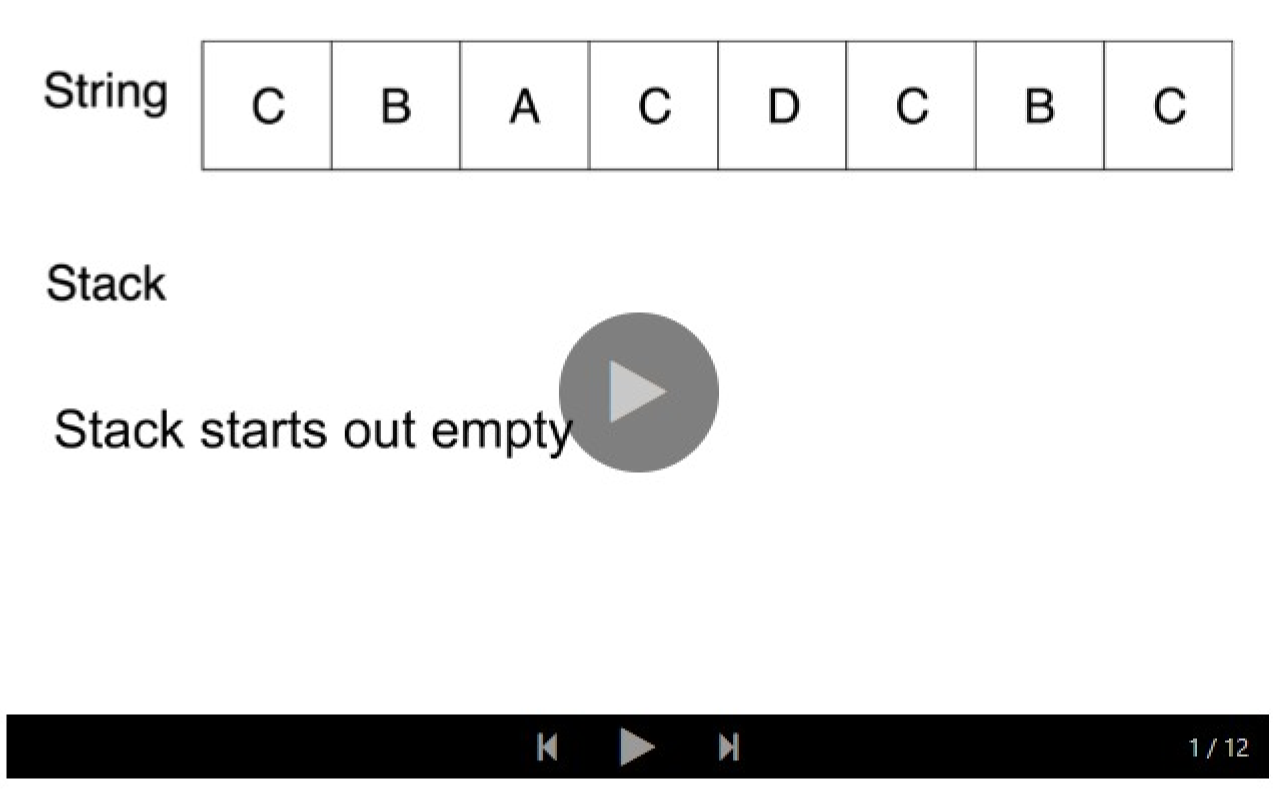
Each iteration we add the current character to the solution if it hasn't already been used. We try to remove as many characters as possible off the top of the stack, and then add the current character

The conditions for deletion are:

1. The character is greater than the current characters
2. The character can be removed because it occurs later on

At each stage in our iteration through the string, we greedily keep what's on the stack as small as possible.

The following animation makes this more clear:



Implementation

Java | Python | Copy

```
class Solution:
    def removeDuplicateLetters(self, s) -> int:

        stack = []

        # this lets us keep track of what's in our solution in O(1) time
        seen = set()

        # this will let us know if there are no more instances of s[i] left in s
        last_occurrence = {c: i for i, c in enumerate(s)}


        for i, c in enumerate(s):

            # we can only try to add c if it's not already in our solution
            # this is to maintain only one of each character
            if c not in seen:
                # if the last letter in our solution:
                #    1. exists
                #    2. is greater than c so removing it will make the string smaller
                #    3. it's not the last occurrence
                # we remove it from the solution to keep the solution optimal
                while stack and c < stack[-1] and i < last_occurrence[stack[-1]]:
                    seen.discard(stack.pop())
                seen.add(c)
                stack.append(c)
        return ''.join(stack)
```

Complexity Analysis

- Time complexity : $O(N)$. Although there is a loop inside a loop, the time complexity is still $O(N)$. This is because the inner while loop is bounded by the total number of elements added to the stack (each time it fires an element goes). This means that the *total* amount of time spent in the inner loop is bounded by $O(N)$, giving us a total time complexity of $O(N)$
- Space complexity : $O(1)$. At first glance it looks like this is $O(N)$, but that is not true! `seen` will only contain unique elements, so it's bounded by the number of characters in the alphabet (a constant). You can only add to `stack` if an element has not been seen, so `stack` also only consists of unique elements. This means that *both* `stack` and `seen` are bounded by constant, giving us $O(1)$ space complexity.

Rate this article:

Previous Next

Comments: 9 Sort By

Type comment here... (Markdown is supported)

Preview Post

Neal_Yang ★ 286 September 8, 2019 6:48 AM

Greedy problem is hard to find the trick

8 Share Reply

haoyangfan ★ 911 November 20, 2019 10:34 AM

@alwinpeng could you please elaborate on the second conclusion mentioned under "intuition" section? In particular, I don't quite get the meaning of **Characters that come later in the string i don't matter in this calculation because i is in a more significant spot. Even if character i+1 isn't the best yet, we can always replace it for a smaller character down the line if possible**

Thanks

Read More

4 Share Reply

SHOW 1 REPLY

lenchen1112 ★ 1006 December 10, 2019 3:45 PM

If we say approach 2 only needs O(1) space due to alphabet size, then it doesn't need to use a set `seen` for O(1) time searching. Just search stack directly will be enough.

1 Share Reply

rayaprolu ★ 1 August 23, 2019 6:49 PM

Isn't there a much easier solution with O(1) space? Just keep a array of size 26 , and loop through the list once, to mark if the lowercase letter exists or not. Then just loop through the array of size 26 (from a to z) to construct your output string. Perhaps i'm not understanding the question correctly?

1 Share Reply

SHOW 1 REPLY

djw612 ★ 8 July 9, 2020 8:44 AM

In the first approach, each recursive call you iterate through the string one time to replace the substring. Is `replaceAll()` counted as O(1) in the solution? What's worse, in new version of java `substring()` itself is O(n).

0 Share Reply

theseungjin ★ 168 January 15, 2020 2:52 PM

The animation slide for stack approach isn't quite the same as the code implementation. The code implementation pops from the stack first before appending a new character

0 Share Reply

letrungkien7 ★ 329 January 6, 2020 5:47 PM

https://leetcode.com/submissions/detail/290200721/ is a duplicate of this.
I solved the above with Greedy, but the stack IDEA is brilliant, couldn't come up with that.

0 Share Reply

haoyangfan ★ 911 November 28, 2019 7:31 AM

for the second solution, a slightly difference implementation using `collections.Counter` instead of manually doing the dict comprehension

```
class Solution:
```

Read More

0 Share Reply

SriharshaY ★ 0 August 28, 2019 11:21 AM

Can we use a set in python , push all the characters in the string into it and then sort the set?
Is it valid or am i missing something?

0 Share Reply

SHOW 2 REPLIES