395. Longest Substring with At Least K Repeating Characters

Medium

Given a string `s` and an integer `k`, return *the length of the longest substring of* `s` *such that the frequency of each character in this substring is greater than or equal to* `k`.

Example 1:

```
Input: s = "aaabb", k = 3
Output: 3
Explanation: The longest substring is "aaa", as 'a' is repeated 3 times.
```

Example 2:

```
Input: s = "ababbc", k = 2
Output: 5
Explanation: The longest substring is "ababb", as 'a' is repeated 2 times and
'b' is repeated 3 times.
```

Constraints:

- `1 <= s.length <= 10^4`
- `s` consists of only lowercase English letters.
- `1 <= k <= 10^5`

Solution

Overview

We want to find the longest substring in a given string `s` where each character is repeated at least `k` times. This is an interesting problem that can be solved using different algorithm paradigms like Divide and Conquer and the Sliding Window Approach. We will start by discussing the brute force approach, moving towards more efficient implementations.

Let's discuss each approach in detail.

Approach 1: Brute Force

Intuition

The naive approach would be to generate all possible substrings for a given string `s`. For each substring, we must check if all the characters are repeated at least `k` times. Among all the substrings that satisfy the given condition, return the length of the longest substring.

Algorithm

- Generate substrings from string `s` starting at index `start` and ending at index `end`.
- Use the `countMap` array to store the frequency of each character in the substring.
- The `isValid` method uses `countMap` to check whether every character in substring has at least `k` frequency.
- Track the maximum substring length and return the result.

Implementation

Complexity Analysis

- Time Complexity : O(n2)\mathcal{O}(n^{2})O(n2), where nnn is equal to length of string sss. The nested for loop that generates all substrings from string sss takes O(n2)\mathcal{O}(n^{2})O(n2) time, and for each substring, we iterate over countMap\text{countMap}countMap array of size 262626. This gives us time complexity as O(26·n2)\mathcal{O}(26 \cdot n^{2})O(26·n2) = O(n2)\mathcal{O}(n^{2})O(n2).

This approach is exhaustive and results in *Time Limit Exceeded (TLE)*.

- Space Complexity: O(1)\mathcal{O}(1)O(1) We use constant extra space of size 26 for `countMap` array.

Approach 2: Divide And Conquer

Intuition

Divide and Conquer is one of the popular strategies that work in 2 phases.

- Divide the problem into subproblems. (Divide Phase).
- Repeatedly solve each subproblem independently and combine the result to solve the original problem. (Conquer Phase).

We could apply this strategy by recursively splitting the string into substrings and combine the result to find the longest substring that satisfies the given condition. The longest substring for a string starting at index `start` and ending at index `end` can be given by,

```
longestSustring(start, end) = max(longestSubstring(start, mid),
longestSubstring(mid+1, end))
```

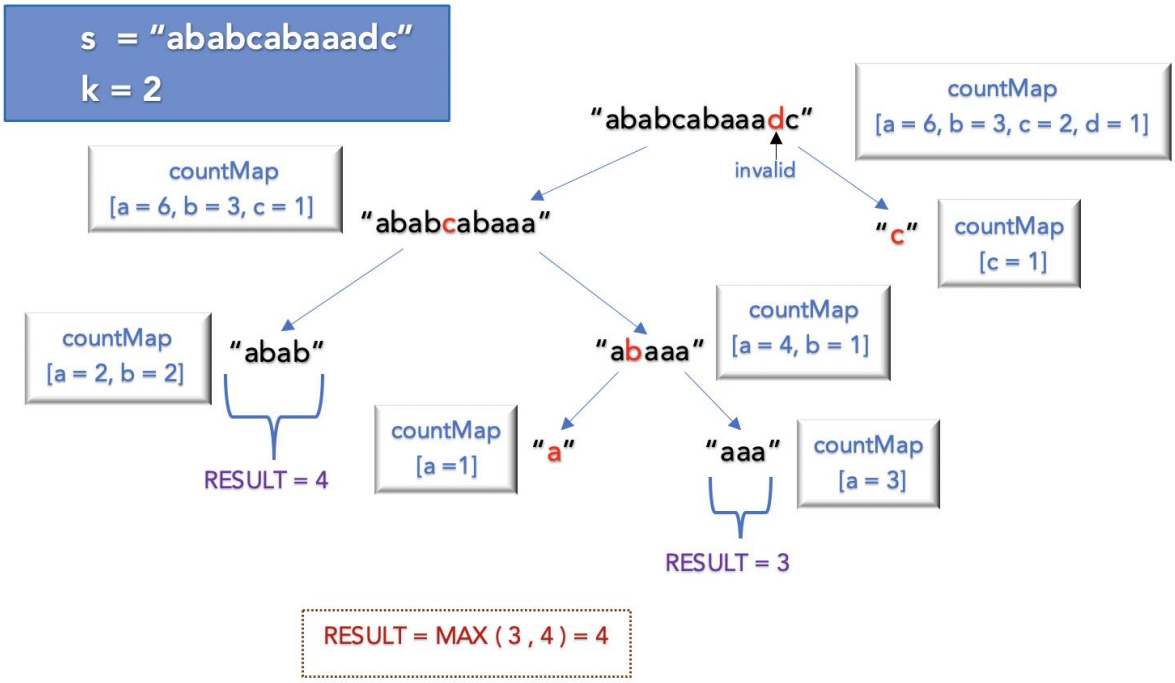
Finding the split position *(mid)*

The string would be split only when we find an invalid character. An invalid character is the one with a frequency of less than `k`. As we know, the invalid character cannot be part of the result, we split the string at the index where we find the invalid character, recursively check for each split, and combine the result.

Algorithm

- Build the `countMap` with the frequency of each character in the string `s`.
- Find the position for `mid` index by iterating over the string. The `mid` index would be the first invalid character in the string.
- Split the string into 2 substrings at the `mid` index and recursively find the result.

To make it more efficient, we ignore all the invalid characters after the mid index as well, thereby reducing the number of recursive calls.



Implementation

Complexity Analysis

- Time Complexity : O(N2)\mathcal{O}(N ^ {2})O(N2), where NNN is the length of string sss. Though the algorithm performs better in most cases, the worst case time complexity is still O(N2)\mathcal{O}(N ^ {2})O(N2).

In cases where we perform split at every index, the maximum depth of recursive call could be O(N)\mathcal{O}(N)O(N). For each recursive call it takes O(N)\mathcal{O}(N)O(N) time to build the `countMap` resulting in O(n2)\mathcal{O}(n ^ {2})O(n2) time complexity.

- Space Complexity: O(N)\mathcal{O}(N)O(N) This is the space used to store the recursive call stack. The maximum depth of recursive call stack would be O(N)\mathcal{O}(N)O(N).

Approach 3: Sliding Window

Intuition

There is another intuitive method to solve the problem by using the Sliding Window Approach. The sliding window slides over the string `s` and validates each character. Based on certain conditions, the sliding window either expands or shrinks.

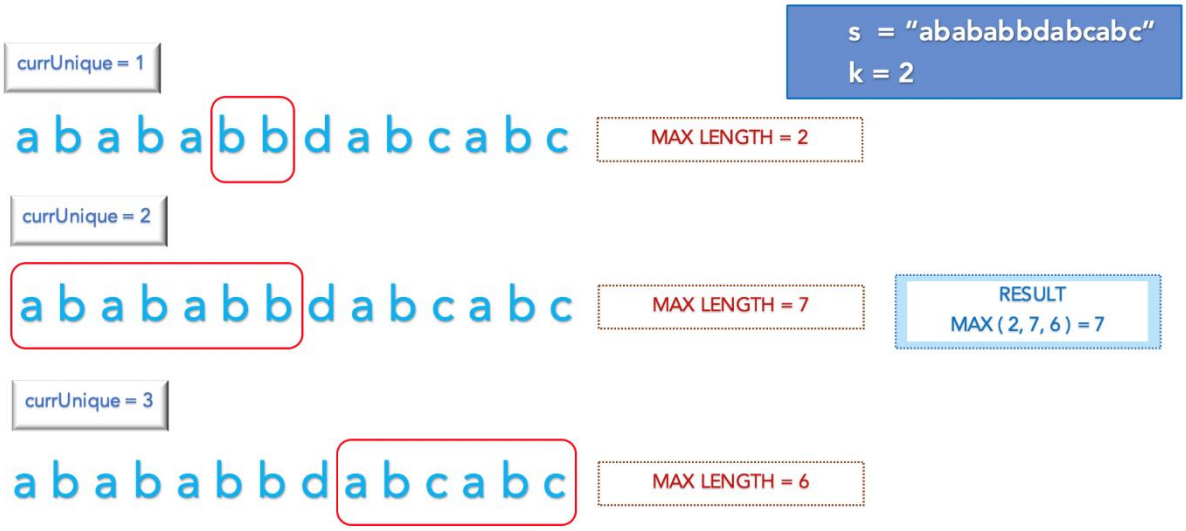
A substring is valid if each character has at least `k` frequency. The main idea is to find all the valid substrings with a different number of unique characters and track the maximum length. Let's look at the algorithm in detail.

Algorithm

1. Find the number of unique characters in the string `s` and store the count in variable `maxUnique`. For `s` = `aabcbacad`, the unique characters are `a,b,c,d` and `maxUnique = 4`.
2. Iterate over the string `s` with the value of `currUnique` ranging from `1` to `maxUnique`. In each iteration, `currUnique` is the maximum number of unique characters that must be present in the sliding window.
3. The sliding window starts at index `windowStart` and ends at index `windowEnd` and slides over string `s` until `windowEnd` reaches the end of string `s`. At any given point, we shrink or expand the window to ensure that the number of unique characters is not greater than `currUnique`.

- If the number of unique character in the sliding window is less than or equal to `currUnique`, expand the window from the right by adding a character to the end of the window given by `windowEnd`
- Otherwise, shrink the window from the left by removing a character from the start of the window given by `windowStart`.

4. Keep track of the number of unique characters in the current sliding window having at least `k` frequency given by `countAtLeastK`. Update the result if all the characters in the window have at least `k` frequency.



Implementation

Complexity Analysis

- Time Complexity : O(maxUnique·N)\mathcal{O}(\text{maxUnique} \cdot N)O(maxUnique·N). We iterate over the string of length NNN, maxUnqiue\text{maxUnqiue}maxUnqiue times. Ideally, the number of unique characters in the string would not be more than 262626 `(a to z)`. Hence, the time complexity is approximately O(26·N)\mathcal{O}(26 \cdot N)O(26·N) = O(N)\mathcal{O}(N)O(N)
- Space Complexity: O(1)\mathcal{O}(1)O(1) We use constant extra space of size 26 to store the `countMap`.