Documentation

The problem **"Longest Substring with At Least K Repeating Characters"** requires finding the longest contiguous substring where every character appears at least k times. If no such substring exists, the function should return 0. For example, given "aaabb" and k = 3, the output is 3 because "aaa" is the longest valid substring. Similarly, for "ababbc" with k = 2, the result is 5 as "ababb" meets the criteria. The challenge lies in efficiently finding this substring without brute force checking all possibilities.

Intuition

The key insight is that characters appearing fewer than k times cannot be part of the final substring. Instead of checking every possible substring, we can use these characters as **split points** to divide the string and recursively process the segments. This allows us to explore valid substrings rather than considering all possibilities efficiently.

Key Observations

First, if the length of s is smaller than k, no valid substring can exist, so we return 0. If every character in s appears at least k times, the entire string is a valid answer. Otherwise, we identify characters appearing fewer than k times and use them as split points. This divides s into smaller substrings that can be processed separately. The longest valid substring is then the maximum result obtained from these recursive calls.

Approach

The function first checks if s is too short; if so, it returns 0. Next, it counts the frequency of each character. If all characters meet the frequency condition, len(s) is returned. Otherwise, we find the first character appearing fewer than k times and use it as a **split point** to break s into smaller substrings. The function then recursively finds the longest valid substring within each segment and returns the maximum length found.

Edge Cases

Several edge cases should be considered. If all characters in s appear at least k times, the entire string is the answer. If s is shorter than k, the result is 0. When s contains unique characters and $k > 1$, no valid substring exists. The function should also handle cases where multiple valid substrings exist, ensuring the longest one is selected.

Complexity Analysis

The **recursive approach's time complexity** is approximately **$O(N \log N)$** in the average case but can degrade to **$O(N^2)$** in the worst case if splits occur frequently. The **space complexity** depends on the recursion depth, which is **$O(\log N)$** in the best case but can reach **$O(N)$** in the worst case.

Alternative Approaches

A possible alternative is a **sliding window approach**, which maintains a valid substring while dynamically updating character frequencies. By iterating through different numbers of unique characters, a valid substring can be found efficiently. This method can achieve **$O(N)$** time complexity but is more complex to implement than the recursive approach.

Test Cases and Final Thoughts

Several test cases should be used to verify the solution, including cases where s is fully valid, contains no valid substrings, or requires multiple splits. The recursive approach is effective due to its divide-and-conquer nature, efficiently pruning unnecessary calculations. While alternative solutions exist, the recursive method balances clarity and performance, making it a strong choice for solving this problem.