2516. Take K of Each Character From Left and Right

Approach 1: Recursion (Time Limit Exceeded)

Intuition

At first glance, it seems feasible to solve this by checking all possible choices: on each step, we could either take a character from the left or from the right. By tracking the count of each character collected along the way, we could determine the minimum steps required to reach at least k occurrences for each character.

This naturally suggests a recursive approach. We can visualize the problem as a decision tree, where each branch corresponds to picking a character from one of the two ends. As we move through this tree, we update the count of each character collected. When the counts meet or exceed k for all characters, we log the steps taken.

However, this approach leads to an exponential time complexity. Each decision doubles the number of possible paths, resulting in a time complexity of $O(2^n)$, where n is the length of the string. For longer strings, this rapidly becomes impractical, as the number of recursive calls grows exponentially. While this method might work for smaller cases, it is unsuitable for larger strings due to the excessive computation time required.

2516. Take K of Each Character From Left and Right

```
class Solution {
    int minMinutes = INT_MAX;

public:
    int takeCharacters(string s, int k) {
        if (k == 0) return 0;
        vector<int> count(3, 0);
        solve(s, k, 0, s.length() - 1, count, 0);
        return minMinutes == INT_MAX ? -1 : minMinutes;
    }

private:
    void solve(string& s, int k, int left, int right, vector<int> count,
               int minutes) {
        // Base case: check if we have k of each character
        if (count[0] >= k && count[1] >= k && count[2] >= k) {
            minMinutes = min(minMinutes, minutes);
            return;
        }
        // If we can't take more characters
        if (left > right) return;

        // Take from left
        vector<int> leftCount = count;
        leftCount[s[left] - 'a']++;
        solve(s, k, left + 1, right, leftCount, minutes + 1);

        // Take from right
        vector<int> rightCount = count;
        rightCount[s[right] - 'a']++;
        solve(s, k, left, right - 1, rightCount, minutes + 1);
    }
};
```

2516. Take K of Each Character From Left and Right

Complexity Analysis

Let n be the length of the string.

Time complexity: $O(2^n)$

The solve function uses a recursive backtracking where, at each step, it has two choices.

This binary decision at each position leads to a total of 2^n possible combinations in the worst case. Even though there are base cases that can terminate some recursive paths early (e.g., when the required counts are met or when the left index exceeds the right), in the worst-case scenario where the solution requires exploring all possible subsets, the time complexity remains exponential.

Additionally, built-in functions like min operate in constant time $O(1)$, and copying the count array (which has a fixed size of 3) also takes constant time. Therefore, these do not affect the overall exponential time complexity.

Space complexity: $O(n)$

The primary space consumption comes from the recursion stack. In the worst case, the depth of recursion can reach n when characters are taken one by one from either end until the entire string is processed. Each recursive call uses a constant amount of additional space (for variables like $leftCount$ and $rightCount$), so the overall space complexity is linear with respect to the length of the string.

The count array has a fixed size of 3, contributing only $O(1)$ space. Therefore, the dominant factor is the recursion depth, leading to a space complexity of $O(n)$.