Documentation on Longest Repeating Character Replacement

Table of Contents

1. Problem Statement

Given a string s consisting of uppercase English letters and an integer k, you can replace at most k characters in s to make the longest possible substring with the same character.

Return the length of the longest substring that can be formed.

Example 1:

- Input: s = "ABAB", k = 2
- Output: 4
- Explanation: Replace the two 'A's with 'B's or vice versa.

Example 2:

- Input: s = "AABABBA", k = 1
- Output: 4
- Explanation: Replace one 'A' to form "AABBBBA", longest substring = "BBBB" (length 4).

Constraints:

- $1 \leq s.length \leq 10^5$
- s consists of only uppercase English letters.
- $0 \leq k \leq s.length$

2. Intuition

The problem requires us to determine the longest contiguous substring with repeating characters, allowing up to k changes. Instead of brute-forcing all possible substrings, we can use the **Sliding Window** technique to efficiently track the most frequent character and adjust the window size dynamically.

3. Key Observations

1. The length of a valid window should be such that **at most k characters are different from the most frequent character**.
2. If max_freq is the count of the most frequent character in the current window, then the number of characters that need to be changed is:
3. Window_Size - max_freq

   This value should be at most k.

4. If the condition is violated, **shrink the window from the left**.
5. The maximum valid window encountered will be our answer.

4. Approach

We use a **Sliding Window** approach:

1. **Expand the window** by moving right and update the frequency count of characters.
2. **Calculate the most frequent character** count in the window (max_freq).
3. If (right - left + 1) - max_freq > k, **shrink the window from the left**.
4. Keep track of the maximum window size found.

Algorithm Steps:

1. Initialize:
   - count = {} to track character frequencies.
   - left = 0 to mark the start of the window.
   - max_freq = 0 to track the most frequent character in the window.
   - max_length = 0 to store the maximum length of a valid substring.
2. Iterate over right:
   - Update the frequency of s[right].
   - Update max_freq.
   - Check if the window is valid (window_size - max_freq ≤ k).
   - If invalid, shrink the window (left += 1).
   - Update max_length.
3. Return max_length.

5. Edge Cases

1. **Single character string:**
    - Example: s = "A", k = 0
    - Expected output: 1
2. **String with all same characters:**
    - Example: s = "AAAA", k = 2
    - Expected output: 4
3. **String with distinct characters and k = 0:**
    - Example: s = "ABCDEF", k = 0
    - Expected output: 1
4. **k is large enough to replace the entire string:**
    - Example: s = "ABABAB", k = 6
    - Expected output: 6
5. **Empty string (s = ""):**
    - Expected output: 0

6. Complexity Analysis

Time Complexity

- The right pointer moves through the string **O(N)** times.
- The left pointer also moves through the string **O(N)** times in the worst case.
- **Total: O(N) + O(N) = O(N).**

Space Complexity

- We store character frequencies in a dictionary of at most **26** entries (O(1)).
- Other variables (max_freq, left, right) use **O(1)** space.
- **Overall: O(1).**

7. Alternative Approaches

Brute Force ($O(N^2)$)

- Try all substrings and check if they can be converted with at most k replacements.
- **Not feasible for large constraints (10^5).**

Binary Search + Sliding Window (O(N log N))

- Use binary search to find the maximum valid window size.

- Check validity using a sliding window for each length.
- **More complex but still efficient.**

8. Test Cases

```
solution = Solution()
print(solution.characterReplacement("ABAB", 2))  # Output: 4
print(solution.characterReplacement("AABABBA", 1))  # Output: 4
print(solution.characterReplacement("AAAA", 2))  # Output: 4
print(solution.characterReplacement("ABCDEF", 0))  # Output: 1
print(solution.characterReplacement("ABABAB", 6))  # Output: 6
print(solution.characterReplacement("", 2))  # Output: 0
print(solution.characterReplacement("A", 0))  # Output: 1
```

9. Final Thoughts

- The **Sliding Window** approach provides an optimal **O(N)** solution.
- This problem is a **classic example of the sliding window technique**, widely used in substring problems.
- Understanding **how to maintain a valid window** and adjust it dynamically is key.
- Alternative approaches exist but are less efficient.