2730. Find the Longest Semi-Repetitive Substring

Problem Description

String)

Medium

Sliding Window

semi-repetitive if it contains at most one consecutive pair of the same digit. Any additional pairs of identical consecutive digits would disqualify the substring from being semi-repetitive. Our goal is to determine the maximum length of any such semirepetitive substrings within s. For example, in the string "122344", the longest semi-repetitive substring would be "1223" with the length of 4, as it contains only one pair of consecutive '2's.

The problem tasks us with finding the longest semi-repetitive substring within a given numeric string s. A string is considered

Intuition

contract depending on the conditions we are looking for. In this particular problem, our window expands whenever we add characters that do not form more than one consecutive pair with the same digit. When we encounter a situation where our window includes more than one consecutive pair, we need to

involving substrings or subarrays. The idea is to keep track of a window (or a range) within the string, which can expand or

To solve the problem, the intuition is to use a sliding window approach. A sliding window is a common technique used in problems

contract the window from the left side. Imagine running through the string from left to right, keeping track of the longest window encountered thus far. Initially, the window can simply grow as characters are added. If we encounter a second consecutive pair of the same digit, we need to move

exceed one, the window can grow. But if an excess is detected, the left edge of the window will move to reduce the count,

ensuring the substring always remains semi-repetitive. The maximum window length is tracked all along and continuously

the left edge of the window to the right until we are left with at most one consecutive pair within our window. At each step, we compare the size of the current window to the maximum size found so far and update if necessary. The solution code follows this reasoning by setting up a loop through the characters of the string while keeping track of two

pointers denoting the window's start and end and a count to record instances of consecutive pairs. As long as the count does not

updated as the loop proceeds. This approach effectively parses the entire string and yields the correct maximum size of the semi-repetitive substring. **Solution Approach** To implement the solution, the algorithm utilizes two pointers and a counter as central aspects of the sliding window technique.

o j is initialized to 0 and represents the start of the sliding window. • i is used to iterate over the string, effectively defining the end of the sliding window.

Counter (cnt): The counter keeps track of consecutive pairs within the current window.

Initialize an accumulator ans to store the maximum length encountered, a counter cnt to count consecutive pairs, a start

With these components, the algorithm proceeds as follows:

beginning of the string.

class Solution:

n = len(s)

return ans

Example Walkthrough

10.

string.

Python

class Solution:

Solution Implementation

ans = cnt = j = 0

for i in range(n):

cnt += 1

if i and s[i] == s[i - 1]:

ans = max(ans, i - j + 1)

The window size is now 4, update ans (ans = 4).

The window size is now 5, update ans (ans = 5).

At i = 5, s[i] = "4", proceed to next iteration.

At i = 4, s[i] = "1" which does not form a consecutive pair.

At i = 6, s[i] = "4", which makes a consecutive pair with s[i-1].

Increment cnt (cnt = 2) because now there are two consecutive '4's.

Increment j so the new start of the window is at j = 1.

At i = 7, s[i] = "5", move to the next iteration.

Return the maximum length found, which is ans = 6.

removed one such pair from the window.

Two Pointers (j and i):

Here is the breakdown of the algorithm with these components:

If the current character at i is the same as the previous character (s[i] = s[i - 1]), increment the counter cnt because we've found a consecutive pair. However, this should be done if i is not the first character to avoid checking before the

While the counter cnt is greater than 1, which means there are more than one consecutive pairs in our current window,

increment j to move the start of the window to the right. This effectively reduces the size of the window from the beginning

After adjusting the window to ensure it is semi-repetitive, calculate the length of the current window (i - j + 1) and update

side. If the character at the new start j and the next character j + 1 form a consecutive pair, decrement cnt as we've

Iterate through the string with index i from 0 to n-1 (inclusive), where n is the length of the string.

pointer j at position 0, and the end pointer i which will iterate over the string.

- Once all possible windows have been considered (post loop completion), return the maximum length ans found during the iteration. Here's the code enclosed with backticks for proper markdown display:
- while cnt > 1: if s[j] == s[j + 1]: cnt -= 1 j += 1

This algorithm makes only a single pass through the string, therefore having a time complexity of O(n) where n is the length of the

Initialize the answer variable ans to store the maximum length found (ans = 0). Initialize the count variable cnt to keep track

```
string, and a space complexity of 0(1) as it uses a fixed number of extra variables.
```

ans if the current window is larger than the previous maximum.

def longestSemiRepetitiveSubstring(self, s: str) -> int:

```
of consecutive pairs (cnt = 0). Set the start of the sliding window j = 0.
Start iterating with i = 0.

    Since the first character doesn't have a previous character, just continue to next iteration.

At i = 1, s[i] = "1" (the second '1'), s[i-1] = "1" (the first '1').
Since s[i] == s[i-1], increment cnt by 1 (cnt = 1).
At i = 2, s[i] = "2", no consecutive pair is found.
• The window size is now 3 (from index j = 0 to i = 2), update ans (ans = 3).
At i = 3, s[i] = "3", continue to next because no consecutive pair is found.
```

Let's walk through an example to illustrate how the algorithm works with the following input string s = "11231445".

Now cnt > 1, initiate while loop to shrink the window from the left. \circ Since s[j] == s[j+1] (both are '1'), decrement cnt by 1 (cnt = 1).

The window size is now 6 (from j = 1 to i = 6), update ans (ans = 6).

The loop has now considered all possible windows within the string.

- The longest semi-repetitive substring in s = "11231445" is then "123144", with a maximum length of 6. This is consistent with the problem's requirements—a substring with at most one consecutive pair (in this case, the pair of '4's).
- # Initialize variables; ans for storing the maximum length, # count for counting consecutive repetitions, and j as the left pointer max_length = count = left_pointer = 0

current_length = right_pointer - left_pointer + 1

int stringLength = s.length(); // Length of the input string

if (s.charAt(j) == s.charAt(j + 1)) {

maxLength = Math.max(maxLength, i - j + 1);

return maxLength; // Return the maximum length found

--repeatedCharsCount;

--repeatCount;

function longestSemiRepetitiveSubstring(s: string): number {

// Move the start index forward

startIndex++;

++left;

return longestLength;

max_length = max(max_length, current_length)

public int longestSemiRepetitiveSubstring(String s) {

as the previous one, increment the count of repetitions

if right_pointer > 0 and s[right_pointer] == s[right_pointer - 1]:

If we are not at the first character and the current character is the same

If there are more than one consecutive repetition, move the left pointer

def longest_semi_repetitive_substring(self, s: str) -> int:

Loop through the string using the right pointer i

for right_pointer in range(string_length):

Initialize the length of the string

string_length = len(s)

count += 1

Return the maximum length found

return max_length

++j;

Java

C++

};

TypeScript

class Solution {

class Solution {

while count > 1:

If the character at the left pointer is followed by the same character, # decrement the count because we are moving past this repetition if s[left_pointer] == s[left_pointer + 1]: count -= 1 # Move the left pointer to the right left_pointer += 1 # Update max_length with the length of the current semi-repetitive substring

The algorithm effectively finds the longest substring satisfying the semi-repetitive condition with a single pass over the input

int maxLength = 0; // Initialize the maximum length to zero // Start with two pointers i and j at the beginning of the string s and a count 'repeatedCharsCount' to record consecutiv for (int i = 0, j = 0, repeatedCharsCount = 0; i < stringLength; ++i) {</pre> // Check if the current character is the same as the previous character, increase repeatedCharsCount if (i > 0 && s.charAt(i) == s.charAt(i - 1)) { ++repeatedCharsCount; // If the repeatedCharsCount is more than one (more than two repeated characters), // move the start pointer 'j' forward until repeatedCharsCount is at most one while (repeatedCharsCount > 1) {

// Update the maximum length of the substring encountered so far

- public: // Function to find the length of longest semi-repetitive substring. // A semi-repetitive substring is a substring where no character appears // more than twice in a row. int longestSemiRepetitiveSubstring(string s) { // Initialize the size of the string and variable to keep track of the // maximum length of the semi-repetitive substring found so far. int stringSize = s.size(); int longestLength = 0; // Initialize pointers for substring window and a counter to track consecutive // character repetition count. for (int left = 0, right = 0, repeatCount = 0; right < stringSize; ++right) {</pre> // If the character is the same as the previous one, increase repetition count. if (right > 0 && s[right] == s[right - 1]) { ++repeatCount; // If repeatCount is more than 1, it means the character has appeared // more than twice. Shift the left pointer to the right to reduce the count. while (repeatCount > 1) { if (s[left] == s[left + 1]) {
- let startIndex = 0; // Starting index of the current substring let repeatCount = 0; // Count of consecutive repeating characters // Iterate through each character in the string for (let index = 0; index < length; index++) {</pre> // If the current and previous characters are the same, increment the repeat count if (index > 0 && s[index] === s[index - 1]) { repeatCount++; // If there are more than one consecutive repeating characters, move the start index forward while (repeatCount > 1) { // If the character at the start index is the same as its next character, decrement the repeat count if (s[startIndex] === s[startIndex + 1]) { repeatCount--;

longestLength = Math.max(longestLength, index - startIndex + 1);

decrement the count because we are moving past this repetition

Update max_length with the length of the current semi-repetitive substring

// Return the length of the longest semi-repetitive substring

if s[left_pointer] == s[left_pointer + 1]:

current_length = right_pointer - left_pointer + 1

Move the left pointer to the right

max_length = max(max_length, current_length)

let longestLength = 0; // This will keep track of the longest semi-repetitive substring

// Update longestLength with the maximum length found so far.

longestLength = max(longestLength, right - left + 1);

// Return the length of the longest semi-repetitive substring.

const length = s.length; // Store the length of the input string s

return longestLength; class Solution: def longest_semi_repetitive_substring(self, s: str) -> int: # Initialize the length of the string $string_length = len(s)$ # Initialize variables; ans for storing the maximum length, # count for counting consecutive repetitions, and j as the left pointer max_length = count = left_pointer = 0 # Loop through the string using the right pointer i for right_pointer in range(string_length): # If we are not at the first character and the current character is the same # as the previous one, increment the count of repetitions if right_pointer > 0 and s[right_pointer] == s[right_pointer - 1]: count += 1 # If there are more than one consecutive repetition, move the left pointer while count > 1: # If the character at the left pointer is followed by the same character,

// Store the maximum length between the current longest and the length of the current semi-repetitive substring

Time and Space Complexity **Time Complexity**

return max_length

count -= 1

left_pointer += 1

Return the maximum length found

loop, which gives us a complexity of O(n) where n is the length of the string s. Within this loop, it handles a while loop that only decrements the count and moves a second pointer, j, forward, also at most n times over the course of the entire function. Each

character is considered at most twice (once when i passes over it and once when j passes over it), so the while loop does not increase the overall time complexity beyond O(n). Thus, the overall time complexity of the algorithm remains O(n). **Space Complexity**

dependent on the size of the input. No additional data structures that scale with input size are used.

The space complexity of the function is 0(1). It only uses a fixed number of additional variables (ans, cnt, j, i, and n) that are not

The function longestSemiRepetitiveSubstring iterates through each character in the input string s exactly once using a single