1933. Check if String Is Decomposable Into Value-Equal Substrings

String **Easy**

Problem Description

substrings where each substring consists of identical characters (value-equal string), and follows a specific pattern: all but one of these substrings must have a length of 3, and exactly one substring should have a length of 2. For instance, the string "55566777" can be split into "555", "66", and "777". As you can see, all substrings are value-equal, one

In this problem, we are given a string s that consists only of digits. The task is to figure out if we can split the string into multiple

substring is of length 2, and the others are of length 3. Therefore, the function should return true. On the other hand, if we have a string like "55567777", it cannot be split into the required pattern of substrings since after

splitting into "555", "6", and "777", "77", we are left with an extra "77" of length 2 which breaks the rule of having exactly one substring of length 2. The function should return true if such a decomposition is possible, or false otherwise.

The intuition behind the solution is to iterate through the string and count the length of consecutive characters that are the same.

For every such group, we can immediately identify if the length of the group is incompatible with our rules (i.e., if the length mod

Intuition

the value-equal substring of length 2. A few key points to notice here: • If we encounter more than one group where the length mod 3 equals 2, we must return false since we can only have one such substring. • If all groups have lengths that are multiples of 3, we do not have our required substring of length 2, and the answer should be false.

3 equals 1, we cannot make valid substrings out of it). If the group's length mod 3 equals 2, it means we have our candidate for

- Solution Approach
- The solution follows a straightforward approach to solve the problem iteratively without using any complex data structures or algorithms.

• We only return true if we have exactly one group with a length mod 3 equals 2 and all other groups are multiples of 3.

length of the string n.

• i which points to the beginning of the current group of identical characters. • j which finds the end of the group.

Starting point i is initialized at the start of the string. Then, the algorithm enters a loop that continues as long as i is less than the

j is initialized to the same value as i and then increments as long as the next character in the string is equal to s[i], representing the same value-equal character group.

Once the end of the group is identified, we calculate the length of the group by subtracting i from j. We use this length to

problem rules, and thus the function returns false.

2. Initialize a counter cnt2 to 0, which will keep track of groups of length 2.

3. Set j = i, and in this case j = 0, since we are at the start of the string.

Inside the loop, we use two pointers:

- determine if the group contributes to the pattern we're looking for. The remainders after division by 3 lead to three cases:
- If the length of the group modulo 3 is 1, this means the group cannot be decomposed to meet the problem's requirements

need to do anything.

Otherwise, the function returns false.

string, and uses constant extra space.

We start our first loop:

Continue this process:

making one substring of length 3), so the function returns false. If the length of the group modulo 3 is 2, we've potentially found our one substring of length 2. We increment the cnt2 counter

(since we would either need one extra character to make two substrings of length 3 or have one character left over after

to mark this occurrence. If cnt2 becomes greater than 1, it means we've found more than one such group, which violates the

If the length of the group modulo 3 is 0, it perfectly fits into the pattern as one or multiple substrings of length 3, and we don't

- After processing the group, i is set to j to start examining the next group of identical characters. Once we've finished scanning the string, the function checks if exactly one group of length 2 was found by checking if cnt2 is equal to 1. If true, we know that the string can be decomposed according to the given rules, and the function returns true.
- **Example Walkthrough**

This is an efficient approach since it scans the string only once, making the time complexity O(n), where n is the length of the

1. Initialize i to 0, because that is where we'll start our search for consecutive character groups.

7. \mathbf{i} is now 3, and we restart the process with $\mathbf{j} = \mathbf{i}$. We increment \mathbf{j} as "5" is consistent till index 5.

11. The length 2 modulo 3 is 2, so we have found a potential group of length 2. We increment cnt2 to 1.

9. For the next group "777", i = 6 and j = 9. The length is 3, modulo 3 is 0, which is fine.

10. We now encounter "66" with i = 9 and j = 11. The length is j - i = 11 - 9 = 2.

12. cnt2 equals 1, which is good, because we need exactly one group of length 2.

Since we have reached the end of the string, we now check:

8. At j = 6, s[j] is not "5" anymore, so our group is "555". The length is 3, and modulo 3 is 0 again, fitting perfectly.

Let's walk through a small example using the string s = "22255577766" to illustrate the solution approach.

4. Increment j as long as s[j] is equal to s[i]. The characters at index 0, 1, and 2 are all "2", so j will increment through these indices. 5. When j reaches 3, s[j] is no longer equal to s[i], so we've found our first group "222". The length of this group is j - i = 3 - 0 = 3. 6. The length 3 modulo 3 is 0, so this group perfectly fits as a substring of length 3. We update i = j and move to the next group:

Solution Implementation

def isDecomposable(self, s: str) -> bool:

if sequence_length % 3 == 1:

count_of_twos += sequence_length % 3 == 2

return False

return count_of_twos == 1

Iterate through the string

while index < length:</pre>

Since all other groups were of length 3 and we found exactly one group of length 2, the function would return true, indicating that the string s can be decomposed according to the given rules.

index, length = 0, len(s) # Initialize starting index and get the length of the string

If the sequence is not divisible by 3 and leaves remainder 1, it's not decomposable

If there is a sequence with a length that leaves a remainder of 2, increment the count

There should not be more than one sequence with a length that leaves a remainder of 2

The string is decomposable if we find exactly one sequence with a remainder of 2

// Initialize the starting index 'start', get the length of the string 'length',

// and a counter for sequences of length 2 ('twoCount').

current_char_index = index # Start index of the current character sequence

Find the index where the current character sequence ends

while current_char_index < length and s[current_char_index] == s[index]:</pre> current_char_index += 1 # Length of the current sequence sequence_length = current_char_index - index

count_of_twos = 0 # Counter for sequences with a length that leaves a remainder of 2 when divided by 3

```
if count_of_twos > 1:
    return False
# Move to the next sequence
index = current_char_index
```

class Solution {

bool isDecomposable(string s) {

public:

};

Java

Python

class Solution:

```
class Solution {
    public boolean isDecomposable(String s) {
        int startIndex = 0; // Initialize the start index of a sequence of same characters
        int strLength = s.length(); // The total length of the string
        int singlePairCount = 0; // Count of sequences where there are two characters (a pair)
        // Iterate over the entire string to check each sequence
       while (startIndex < strLength) {</pre>
            int endIndex = startIndex;
            // Find the end index until which the characters are the same
            while (endIndex < strLength && s.charAt(endIndex) == s.charAt(startIndex)) {</pre>
                endIndex++;
            int sequenceLength = endIndex - startIndex; // Length of the current sequence
            // If the sequence length divided by 3 leaves a remainder of 1, then it's not decomposable
            if (sequenceLength % 3 == 1) {
                return false;
            // If the sequence length divided by 3 leaves a remainder of 2, increment the singlePairCount
           // And if there is more than one such pair, it's not decomposable
            if (sequenceLength % 3 == 2) {
                singlePairCount++;
                if (singlePairCount > 1) {
                    return false; // More than one pair found, thus not decomposable
            // Move to the next sequence
            startIndex = endIndex;
       // The string is decomposable if there is exactly one single pair of characters
        return singlePairCount == 1;
```

```
int start = 0;
       int length = s.size();
       int twoCount = 0;
       // Iterate over the string.
       while (start < length) {</pre>
           // Initialize 'end' to find the end of the current sequence of the same characters.
           int end = start;
           // Increment 'end' while we have the same character as at the start of this sequence.
           while (end < length && s[end] == s[start]) {</pre>
                ++end;
            int sequenceLength = end - start;
           // If any sequence of characters is not divisible by 3 with a remainder of 0 or 2,
           // it means the string cannot be decomposed as required.
           if (sequenceLength % 3 == 1) {
                return false;
           // If the remainder is 2 and we have already found a sequence of such type,
           // return false because we can only have one sequence with a remainder of 2.
           if (sequenceLength % 3 == 2) {
                twoCount++; // Record that we found a two-character sequence
                if (twoCount > 1) {
                    return false; // If more than one such sequence is found, return false.
           // Move the start to the beginning of the next sequence.
            start = end;
       // Check if there was exactly one sequence with a length that is a multiple of 3 plus 2.
       return twoCount == 1;
TypeScript
    let startIndex = 0;
   const length = s.length;
    let twoCount = 0;
```

```
/**
   * Checks if the given string can be decomposed into a sequence of strings with each substring being a consecutive sequence of th
   * and exactly one of these substrings is of length 2 (all others must be of length 3).
   * @param {string} s - The input string to be checked for decomposability.
   * @returns {boolean} - Returns true if the string can be decomposed as required, otherwise false.
  function isDecomposable(s: string): boolean {
      // Iterate through the string.
      while (startIndex < length) {</pre>
          // Find the end index of the current sequence of the same characters.
          let endIndex = startIndex;
          while (endIndex < length && s.charAt(endIndex) === s.charAt(startIndex)) {</pre>
              endIndex++;
          const sequenceLength = endIndex - startIndex;
          // Check if any sequence of characters has a length not decomposable with a remainder of 0 or 2.
          if (sequenceLength % 3 === 1) {
              return false;
          // Track sequences of length 2 (remainder of 2 when divided by 3).
          if (sequenceLength % 3 === 2) {
              twoCount++;
              // Having more than one sequence of this type makes it non-decomposable.
              if (twoCount > 1) {
                  return false;
          // Move to the next sequence.
          startIndex = endIndex;
      // Ensure there was exactly one sequence of length 2.
      return twoCount === 1;
class Solution:
   def isDecomposable(self, s: str) -> bool:
        index, length = 0, len(s) # Initialize starting index and get the length of the string
        count_of_twos = 0 # Counter for sequences with a length that leaves a remainder of 2 when divided by 3
       # Iterate through the string
       while index < length:</pre>
            current_char_index = index # Start index of the current character sequence
            # Find the index where the current character sequence ends
            while current_char_index < length and s[current_char_index] == s[index]:</pre>
                current_char_index += 1
            # Length of the current sequence
            sequence_length = current_char_index - index
            # If the sequence is not divisible by 3 and leaves remainder 1, it's not decomposable
            if sequence_length % 3 == 1:
                return False
            # If there is a sequence with a length that leaves a remainder of 2, increment the count
            count of twos += sequence length % 3 == 2
            # There should not be more than one sequence with a length that leaves a remainder of 2
            if count_of_twos > 1:
                return False
```

Move to the next sequence

index = current_char_index

return count_of_twos == 1

Time and Space Complexity

The string is decomposable if we find exactly one sequence with a remainder of 2

Time Complexity The time complexity of the given code is O(n), where n is the length of the input string s. This is because the while loop iterates

over each character in the string exactly once, with i advancing to j at the end of each iteration. Even though there is a nested while loop, it only serves to increase j to the next character that is different from s[i], and each character is visited only once by this inner loop. Therefore, the overall number of operations is linear with respect to the size of the input string. **Space Complexity**