1177. Can Make Palindrome from Substring

Medium · Bit Manipulation · Array · Hash Table · String · Prefix Sum   Leetcode Link

Problem Description

The problem requires us to determine whether a substring of a given string `s` can be rearranged and possibly modified by replacing up to `k` letters to form a palindrome. This needs to be done for a series of queries, each represented by a triplet `[left, right, k]`. Specifically, for each query, we are allowed to:

1. Rearrange the substring `s[left...right]`.
2. Replace up to `k` letters in the substring with any other lowercase English letter.

If the substring can be made into a palindrome using the above operations, the result for that query is `true`; otherwise, it is `false`. We are to return an array of boolean values representing the result for each query.

A palindrome is a string that reads the same forward and backward. For a string to be a palindrome, each character must have a matching pair except for at most one character, which can be in the middle of the palindrome if the string length is odd.

Intuition

The intuition behind the solution approach is to first understand the characteristics of a palindrome. For a string to be a palindrome, each character except for at most one must occur an even number of times (they can be mirrored around the center of the string).

Given this, we can reformulate the problem as: At most how many characters in the target substring have an odd number of occurrences, and whether this number can be reduced to at most one with at most `k` character replacements.

To efficiently compute the number of characters with odd occurrences in any given substring, the solution employs prefix sums. Specifically, it calculates the count of each character of the alphabet up to each position in the string. This provides a quick way to determine the counts of each letter in any substring.

Here's the step-by-step approach of the solution:

1. Create a list `ss` to keep track of the prefix sums – the count of each character up to each index of the string.
2. Iterate over the string `s` to fill up the `ss` list with the counts.
3. For each query, use the `ss` list to calculate the number of characters with an odd count in the target substring.
4. Check if the half of the number of odd-count characters is less than or equal to `k`, since each pair of odd-count characters can be replaced by any other character to make them even.
5. For each query, append the result (`true` or `false`) to the answer list `ans`.
6. Return the `ans` list as output once all queries are processed.

This approach allows us to efficiently answer each query without having to directly manipulate the substring, reducing the problem to a question of counting occurrences which can be solved in constant time for each query.

Solution Approach

The solution uses the concept of prefix sums and bitwise operations to solve the problem efficiently. Here's how it works, with reference to the key steps in the Python code implementation:

1. **Initialization**: A two-dimensional list `ss` is initiated where `ss[i][j]` represents the count of the `j-th` letter (where `a` is `0`, `b` is `1`, and so on) up to the `i-th` position in the string `s`. This list is initialized with `n + 1` rows and `26` columns (since there are `26` letters in the English alphabet) filled initially with zeros. This extra row is for handling the prefix sum from the beginning of the string.

   ```python
   ss = [[0] * 26 for _ in range(n + 1)]
   ```

2. **Populating Prefix Sums**: As we iterate through the string `s`, a temporary copy of the previous row of the `ss` list is made, and then the count of the current character is updated by incrementing the corresponding counter.

   ```python
   for i, c in enumerate(s, 1):
       ss[i] = ss[i - 1][:]
       ss[i][ord(c) - ord("a")] += 1
   ```

3. **Processing Queries**: For each query `[l, r, k]`, we calculate the number of characters with an odd number of occurrences within the substring `s[l...r]`. This is done by using the corresponding prefix sums to find the total count of each letter in the substring and then applying a bitwise AND operation with `1` (which is equivalent to checking if the count is odd).

   ```python
   cnt = sum((ss[r + 1][j] - ss[l][j]) & 1 for j in range(26))
   ```

   The above line calculates the difference between the counts at the position after the end of the substring (`r + 1`) and at the start of the substring (`l`). This difference gives us the count of each character in the substring. We then check if this count is odd by using the bitwise AND operation with `1`.

4. **Checking for Palindrome Potential**: The number of odd-count characters that need to be paired off (which requires replacement) is `cnt // 2`. We check if this number is less than or equal to `k`, which signifies whether we can turn the substring into a palindrome through `k` or fewer character replacements.

   ```python
   ans.append(cnt // 2 <= k)
   ```

   This check is appended to our result list `ans`.

5. **Returning Results**: Once all queries are processed, the list `ans` containing the result of each query is returned.

   ```python
   return ans
   ```

The use of prefix sums allows for a quick calculation of character occurrences within any given substring in `O(1)` time, after an `O(n)` preprocessing phase. The overall complexity of the solution is `O(n + q)`, where `n` is the length of the string and `q` is the number of queries, making it highly efficient for the problem at hand.

Example Walkthrough

Let's consider the string `s = "aabbcc"` and queries `queries = [[0,5,2], [1,4,1]]`.

Initialization with Prefix Sums: We first initialize our `ss` list with extra space to handle cases when the substring starts at index `0`. The initialized `ss` list looks like this initially (considering a simplified alphabet of three letters for this illustration):

```
ss = [[0, 0, 0],  // before the first character ('a')
      [0, 0, 0],  // before the second character ('a') and so on
      [0, 0, 0],
      [0, 0, 0],
      [0, 0, 0],
      [0, 0, 0],
      [0, 0, 0]]  // after the last character ('c')
```

Populating Prefix Sums: After populating the `ss` list with the prefix sums of each character, the list reflects the following (the `-th` index in the inner lists corresponds to the alphabetically `j-th` character):

```
ss = [[0, 0, 0],  // before 'a'
      [1, 0, 0],  // before 'a'
      [2, 0, 0],  // before 'b'
      [2, 1, 0],  // before 'b'
      [2, 2, 0],  // before 'c'
      [2, 2, 1],  // before 'c'
      [2, 2, 2]]  // after 'c'
```

Query 1: `[0, 5, 2]` We need to check the substring `s[0...5]` which is "aabbcc". For this substring, the counts of each character are 2 `a`'s, 2 `b`'s, and 2 `c`'s. The count of odd-occurring characters `cnt` is therefore `0`. We don't need any replacements to make a palindrome, so the result for this query is `true`.

Query 2: `[1, 4, 1]` This time, we're looking at the substring `s[1...4]` which is "abbc". Here, the count of each character is 1 `a`, 2 `b`s, and 1 `c`. Therefore, we have `cnt=2` characters occurring an odd number of times ("a" and "c"). We need 1 replacement to make either "a" or "c" match the other character (e.g., change "c" to "a" to form "abba"). Since we are allowed to replace up to `k=1` characters, the result for this query is `true`.

Answer: Thus, the array of boolean values representing the result for each query is `[true, true]`.

Python Solution

```python
class Solution:
    def canMakePaliQueries(self, s: str, queries: List[List[int]]) -> List[bool]:
        # Calculate the length of the string.
        string_length = len(s)

        # Initialize a prefix sum array where each element is a list representing the count of letters up to that index.
        prefix_sum = [[0] * 26 for _ in range(string_length + 1)]

        # Populate the prefix sum matrix with the counts of each character.
        for index, char in enumerate(s, 1):
            prefix_sum[index] = prefix_sum[index - 1][:]
            prefix_sum[index][ord(char) - ord("a")] += 1

        # Initialize a list to store the answers for the queries.
        answers = []

        # Process each query in the list of queries.
        for start, end, max_replacements in queries:
            # Calculate the count of odd occurrences of each letter in the range [start, end].
            odd_count = sum((prefix_sum[end + 1][j] - prefix_sum[start][j]) & 1 for j in range(26))

            # A palindrome can be formed if the half of odd_count is less than or equal to the allowed max_replacements.
            can_form_palindrome = odd_count // 2 <= max_replacements

            # Add the result for the current query to the answers list.
            answers.append(can_form_palindrome)

        # Return the answers list containing results for all queries.
        return answers

```

Java Solution

```java
class Solution {
    public List<Boolean> canMakePaliQueries(String s, int[][] queries) {
        int stringLength = s.length();
        // Prefix sum array to keep count of characters up to the ith position
        int[][] charCountPrefixSum = new int[stringLength + 1][26];

        // Fill the prefix sum array with character counts
        for (int i = 1; i <= stringLength; ++i) {
            // Copy the counts from previous index
            for (int j = 0; j < 26; ++j) {
                charCountPrefixSum[i][j] = charCountPrefixSum[i - 1][j];
            }
            // Increment the count of the current character
            charCountPrefixSum[i][s.charAt(i - 1) - 'a']++;
        }

        // List to store results of queries
        List<Boolean> results = new ArrayList<>();

        // Process each query
        for (int[] query : queries) {
            int left = query[0], right = query[1], maxReplacements = query[2];
            int oddCount = 0;

            // Compute the count of characters with odd occurrences in the
            // substring [left, right]
            for (int j = 0; j < 26; ++j) {
                oddCount += (charCountPrefixSum[right + 1][j] - charCountPrefixSum[left][j]) & 1;
            }

            // Add true if half of the oddCount is less than or equal to allowed replacements (maxReplacements)
            results.add(oddCount / 2 <= maxReplacements);
        }

        // Return the list containing results of queries
        return results;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <cstring>
using namespace std;

class Solution {
public:
    vector<bool> canMakePaliQueries(string s, vector<vector<int>>& queries) {
        int n = s.size();
        // Create a 2D array to keep track of character frequency up to each position in the string
        int charCountPrefixSum[n + 1][26];
        memset(charCountPrefixSum, 0, sizeof(charCountPrefixSum)); // Initialize the array with 0

        // Populate the prefix sum array with character frequency counts
        for (int i = 1; i <= n; ++i) {
            for (int j = 0; j < 26; ++j) {
                charCountPrefixSum[i][j] = charCountPrefixSum[i - 1][j];
            }
            charCountPrefixSum[i][s[i - 1] - 'a']++;
        }

        vector<bool> answers; // This will store the answers for each query

        // Go through each query to check for palindrome possibility
        for (auto& query : queries) {
            int left = query[0], right = query[1], maxReplacements = query[2];
            int oddCount = 0; // Variable to track the count of characters appearing odd number of times

            // Count how many characters appear an odd number of times within the query's range
            for (int j = 0; j < 26; ++j) {
                oddCount += (charCountPrefixSum[right + 1][j] - charCountPrefixSum[left][j]) & 1;
            }

            // A palindrome can be formed if the half of the odd count is less than or equal to allowed replacements
            answers.emplace_back(oddCount / 2 <= maxReplacements);
        }

        return answers; // Return the final answers for all queries
    }
};

```

Typescript Solution

```typescript
function canMakePaliQueries(s: string, queries: number[][]): boolean[] {
    const lengthOfString = s.length;
    const charCountPrefixSum: number[][] = Array(lengthOfString + 1)
        .fill(0)
        .map(() => Array(26).fill(0)); // Array to store the prefix sum of character counts

    // Calculate the prefix sum of character counts
    for (let i = 1; i <= lengthOfString; ++i) {
        charCountPrefixSum[i] = charCountPrefixSum[i - 1].slice(); // Copy previous count
        ++charCountPrefixSum[i][s.charCodeAt(i - 1) - 'a'.charCodeAt(0)]; // Increment count of current character
    }

    const result: boolean[] = [];
    // Process each query
    for (const [left, right, maxReplacements] of queries) {
        let oddCount = 0; // Count of characters that appear an odd number of times

        // Calculate the number of characters with an odd count in the substring
        for (let j = 0; j < 26; ++j) {
            oddCount += (charCountPrefixSum[right + 1][j] - charCountPrefixSum[left][j]) & 1;
        }

        // Push true if half of the odd count is less than or equal to k, otherwise false.
        result.push((oddCount >> 1) <= maxReplacements);
    }

    return result; // Return the array of boolean results for each query
}

```

Time and Space Complexity

The given Python code provides a solution for determining if a substring of the input string `s` can be rearranged to form a palindrome with at most `k` replacements. The computation of this solution involves pre-computing the frequency of each character in the alphabet at each index of the string `s`, and then using that information to answer each query.

Time Complexity:

1. Building the prefix sum array `ss` takes `O(n * 26)` time, where `n` is the length of the string `s`, since we iterate over the string and for each character, we copy the previous counts and update the count of one character.
2. Answering each query involves calculating the difference in character counts between the right and left indices for each of the 26 letters, which is `O(26)`. This is done for each query. If there are `q` queries, this part of the algorithm takes `O(q * 26)` time.
3. The total time complexity is therefore `O(n * 26 + q * 26)` which simplifies to `O(n + q)` when multiplied by the constant 26 factor for the alphabet size.

Hence, the time complexity is `O(n + q)`.

Space Complexity:

1. The prefix sum array `ss` uses `O(n * 26)` space to store the count of characters up to each index in the string `s`.
2. The space for the answer list is `O(q)`, where `q` is the number of queries.
3. As such, the total space complexity is the sum of the space for the prefix sum array and the space for the answer list, which is `O(n * 26 + q)`.

Thus, the space complexity is `O(n * 26 + q)` which can be approximated to `O(n)` since the size of the alphabet is constant.