1578. Minimum Time to Make Rope Colorful

Medium · Greedy · Array · String · Dynamic Programming · Leetcode Link

Problem Description

Alice has `n` balloons tied in a row and each balloon is colored some color. This sequence of colors is given as a string `colors` where `colors[i]` is the color of the `i`th balloon. Alice desires the string of balloons to be "colorful," which means no two adjacent balloons should share the same color. To help achieve this, Bob can remove balloons, but it takes time. The time it takes to remove each balloon is specified in an integer array `neededTime`, where `neededTime[i]` represents the seconds required to remove the `i`th balloon.

The goal is to find the minimum amount of time Bob needs to spend to turn the string of balloons into one where no two consecutive balloons are of the same color.

Intuition

Given that we want to make the rope of balloons colorful with the least amount of time needed, we can intuit that when we encounter a sequence of balloons of the same color, we should remove all but one of them. Instead of randomly choosing which balloons to remove, we should aim to remove the balloons with the least time cost. The optimal balloon to keep is the one with the highest `neededTime` value because removing it would contribute the most to the total time.

Throughout the string of balloons, we iteratively look for sequences of consecutive balloons with the same color. For each of these sequences, we calculate the total time needed to remove all balloons (`s`) and the maximum time needed to remove a single balloon within that sequence (`mx`). To make the sequence colorful, we will keep the balloon with the maximum `neededTime` (the most costly to remove) and remove all others. Hence, we subtract the `mx` from the total `s` and add this to our answer (`ans`).

The algorithm is efficient because it goes through the string once, using a while loop, checking each sequence of same-colored balloons and calculating the time cost on the fly. It has a linear time complexity relative to the length of the `colors` string, making it suitable even for a large number of balloons.

Solution Approach

The solution implementation follows a greedy approach, which entails iterating through the `colors` string and grouping balloons with the same color. During this grouping, the algorithm also keeps track of the sum of `neededTime`s of these balloons (`s`) as well as the maximum `neededTime` for a single balloon in the group (`mx`). To do this, two pointers are used. The first pointer `i` marks the beginning of a group of same-colored balloons, and a second pointer `j` is used to find the end of this group.

The approach can be broken down into the following steps:

1. Initialize `ans` as zero, which will store the minimum total time required to make the rope colorful.
2. Loop through the string `colors` using the index `i`, which serves as the starting point of each group of same-colored balloons:
   - Within this loop, start a nested loop with index `j`, beginning at the same position as `i`.
   - Initialize `s` as zero, which will hold the sum of the `neededTime` for all balloons currently in the sequence.
   - Initialize `mx` as zero, which will keep track of the maximum `neededTime` of a balloon in the current sequence.
   - For each balloon that has the same color as the one at position `i` (i.e., `colors[j] == colors[i]`), accumulate its `neededTime` in `s` and update `mx` if the current balloon's `neededTime[j]` is greater than the current `mx`.
   - Increment `j` for as long as it does not exceed the length of the `colors` string and the color of the `j`th balloon is the same as the `i`th balloon.
   - If the sequence length (`j - i`) is greater than 1, it means there are duplicate balloons. In this case, subtract the maximum `neededTime` (`mx`) from the sum of `neededTime` (`s`) to get the time required to make the current sequence colorful and add this value to `ans`.
   - Set `i` to the value of `j` to begin processing the next sequence.
3. Once the loop is complete, all the `neededTime` for removing necessary balloons to avoid consecutive same colors has been added to `ans`. Return `ans` as the answer.

It is important to note that because this approach always chooses the most expensive balloon to keep in each consecutive sequence of same-colored balloons, the accumulated removal time is as small as possible. Essentially, you're selecting which balloons to remove based on the criteria of the lowest time cost, which aligns with our greedy strategy.

Example Walkthrough

Let's consider an example where `colors = "abcccbad"` and `neededTime = [1, 2, 3, 4, 5, 6, 7, 8]`. Our goal is to make the sequence of balloons colorful by removing consecutive balloons of the same color in the least total time possible.

Using the solution approach outlined earlier, we can work through this step by step:

1. We initialize `ans` as zero. This will keep track of the minimum total time needed.
2. We will loop through the string `colors` using index `i`. For simplicity, we consider each character as a group of same-colored balloons, even if it's a group of one.
   - Start with `i = 0`; this is the first balloon with color 'a'.
   - There are no consecutive balloons with the same color, so `i` is incremented to `1`.
   - Now at `i = 1`; this balloon has color 'b'.
   - There are no consecutive balloons with the same color, so `i` is incremented to `2`.
   - At `i = 2`; this balloon has the color 'c'.
   - We find there is a sequence of balloons with the same color 'c' starting from index `2` to `4`.
   - We set `j = 2` and start the nested loop.
   - Initialize `s = 0` and `mx = 0`.
   - We go through the sequence, updating `s` and `mx` as follows:
     - For `j = 2`, `colors[j] = 'c'`, so we update `s += neededTime[2]` (s becomes 3) and `mx = max(mx, neededTime[2])` (mx becomes 3).
     - For `j = 3`, `colors[j] = 'c'`, `s += neededTime[3]` (s becomes 7) and `mx = max(mx, neededTime[3])` (mx remains 3).
     - For `j = 4`, `colors[j] = 'c'`, `s += neededTime[4]` (s becomes 12) and `mx = max(mx, neededTime[4])` (mx becomes 5).
   - Since `j - i` (4 - 2) is greater than 1, we have duplicate balloons and we calculate the time to make the sequence colorful: `ans += s - mx` (ans becomes 12 - 5 = 7).
   - We set `i` to `5` as the next starting point.
   - Continuing with `i = 5`, there are no more consecutive balloons with the same color, so nothing gets added to `ans`.
3. Moving on, loop continues, but no more sequences of consecutive colors are found.

The final `ans` represents the minimum total time to remove all the consecutive same-colored balloons and is equal to `7` in this example.

Python Solution

```python
from typing import List

class Solution:
    def minCost(self, colors: str, neededTime: List[int]) -> int:
        # Initialize the answer and the iterator i.
        total_time = 0
        i = 0
        n = len(colors)

        # Process the string until we reach the end of colors.
        while i < n:
            # Start of the same color sequence.
            start = i
            # Initialize the sum and the maximum time for the current sequence.
            sum_time = 0
            max_time = 0

            # Iterate over the sequence of same colors.
            while i < n and colors[i] == colors[start]:
                # Add the time for painting this balloon to the sum.
                sum_time += neededTime[i]
                # Update the max_time if this balloon's time is greater than the current max.
                max_time = max(max_time, neededTime[i])
                # Move to the next balloon.
                i += 1

            # If there was more than one balloon in the sequence,
            # add the total time minus the time of the most expensive to repaint balloon.
            if i - start > 1:
                total_time += sum_time - max_time

            # No need for updating i here, as it's already one past the current sequence.

        # Return the total minimum time to repaint the balloons so that no adjacent balloons have the same color.
        return total_time

# Example usage:
# sol = Solution()
# colors = "abaac"
# neededTime = [1,2,3,4,5]
# print(sol.minCost(colors, neededTime))  # Should output 3

```

Java Solution

```java
class Solution {
    public int minCost(String colors, int[] neededTime) {
        // Initialize the total minimum cost to 0
        int totalMinCost = 0;
        // Obtain the total number of balloons
        int balloonsCount = neededTime.length;

        // Iterate through the array of needed time to check for consecutive balloons of the same color
        for (int startIndex = 0, endIndex = 0; startIndex < balloonsCount; startIndex = endIndex) {
            // Move the endIndex to the start of the current sequence
            endIndex = startIndex;
            // Initialize the sum of needed time for all balloons in the current sequence and the maximum needed time
            int sumNeededTime = 0, maxNeededTime = 0;
            // Process consecutive balloons of the same color
            while (endIndex < balloonsCount && colors.charAt(endIndex) == colors.charAt(startIndex)) {
                // Add the needed time of the current balloon to the sum
                sumNeededTime += neededTime[endIndex];
                // Update the maximum needed time for the current sequence
                maxNeededTime = Math.max(maxNeededTime, neededTime[endIndex]);
                // Move to the next balloon
                ++endIndex;
            }
            // If there is more than one balloon of the same color consecutively,
            // increment the total minimum cost by the sum of needed times minus the maximum needed time
            if (endIndex - startIndex > 1) {
                totalMinCost += sumNeededTime - maxNeededTime;
            }
        }
        // Return the total minimum cost to make all balloons have distinct colors
        return totalMinCost;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int minCost(string colors, vector<int>& neededTime) {
        // This variable will hold the minimum total time
        int totalTime = 0;
        // The size of the colors string
        int n = colors.size();

        // Loop through the colors string
        for (int i = 0, j = 0; i < n; i = j) {
            // Look for a sequence of the same color
            j = i;
            // 'sumTimes' holds the sum of time for balloons of the same color group
            int sumTimes = 0;
            // 'maxTime' holds the maximum time needed for a single balloon in the same color group
            int maxTime = 0;

            // While we haven't reached the end of the string and the current color is
            // the same as the one we started this sequence with, add to sumTimes and
            // update maxTime if necessary
            while (j < n && colors[j] == colors[i]) {
                sumTimes += neededTime[j];
                maxTime = max(maxTime, neededTime[j]);
                ++j;
            }

            // If we have more than one balloon with the same color in sequence,
            // add to the total time the sum of needed time minus the maximum time
            // needed for a single balloon (we are keeping the one which takes longest to remove)
            if (j - i > 1) {
                totalTime += sumTimes - maxTime;
            }
        }

        // Return the total time calculated
        return totalTime;
    }
};

```

Typescript Solution

```typescript
function minCost(colors: string, neededTime: number[]): number {
    let totalTime = 0;  // This will hold the minimum total time
    const n = colors.length;  // The size of the colors string

    // Loop through the colors string
    for (let i = 0, j = 0; i < n; i = j) {
        // Start of the same color sequence
        j = i;
        // Hold the sum of times for balloons of the same color group
        let sumTimes = 0;
        // Hold the maximum time needed for a single balloon in the same color group
        let maxTime = 0;

        // While we haven't reached the end of the string and the current color is the same
        // as the one we started this sequence with, add to sumTimes and update maxTime
        while (j < n && colors[j] === colors[i]) {
            sumTimes += neededTime[j];
            maxTime = Math.max(maxTime, neededTime[j]);
            j++;
        }

        // Add to total time the sum of needed times minus the max needed time for this sequence
        // This is because we're removing all but one balloon in the sequence
        if (j - i > 1) {
            totalTime += sumTimes - maxTime;
        }
    }

    // Return the total minimum time calculated
    return totalTime;
}

```

Time and Space Complexity

Time Complexity

The provided code consists of a single while loop that iterates through the string `colors`. Inside the loop, there's another while loop that also iterates through the string but only when it finds consecutive characters that are the same. Despite this nested structure, the inner loop does not lead to a quadratic time complexity, because each character in the string `colors` is visited exactly once. After the inner loop, the index `i` jumps to the position `j`, skipping all the characters that have already been accounted for.

Thus, each character in the string causes an iteration of the outer loop, and the inner loop only runs for characters of a sequence of identical colors once. This leads to a linear time complexity with respect to the length of the string `colors`.

Hence, the time complexity is `O(n)`, where `n` is the length of `colors`.

Space Complexity

The code uses a constant amount of extra space: variables `ans`, `i`, `s`, `mx`, and `j`. When iterating over the input, no additional space that scales with the size of the input is used. The input itself (`colors` and `neededTime`) is not counted towards the space complexity.

Therefore, the space complexity is `O(1)` since it does not allocate any additional space that grows with the input size.