2167. Minimum Time to Remove All Cars Containing Illegal Goods

Hard String Dynamic Programming Leetcode Link

Problem Description

In this problem, we are dealing with a 0-indexed binary string `s`, where each character represents a train car. A car can either contain illegal goods (denoted by `'1'`) or not (denoted by `'0'`). The goal is to remove all cars with illegal goods in the least amount of time possible. We have three operations that we can use:

1. Removing a car from the left end of the train takes 1 unit of time.
2. Removing a car from the right end of the train also takes 1 unit of time.
3. Removing a car from any other position takes 2 units of time.

A car removal operation is directed towards a car containing illegal goods, and we want to know the minimum time required to remove all such cars.

Intuition

To solve this problem, we use a dynamic programming approach. The intuition lies in breaking down the problem into simpler sub-problems:

1. The minimum time to remove all 'bad' cars from the beginning (left side) of the train up to any given position.
2. The minimum time to remove all 'bad' cars from the end (right side) of the train from any given position to the end.

The goal is to find out the optimal point (or points) where we switch from removing cars from the left to removing cars from the right, or vice versa, to minimize the total time. We solve for two arrays, `pre` and `suf`, where `pre[i]` is the minimum time to remove all cars with illegal goods from the first `i` cars, and `suf[i]` is the minimum time to remove all cars with illegal goods from position `i` to the end.

For `pre`:

- If a car at position `i` does not contain illegal goods (`'0'`), then `pre[i + 1] = pre[i]` since no action is needed.
- If it does contain illegal goods (`'1'`), we check if it's cheaper to remove it individually, taking 2 units of time, or remove all the `i+1` cars from the left which take `i+1` units of time.

For `suf`:

- If a car at position `i` does not contain illegal goods (`'0'`), then `suf[i] = suf[i + 1]` as no action is needed here too.
- If it does contain illegal goods (`'1'`), we check if it's cheaper to remove it individually, taking 2 units of time, or remove all cars from position `i` to the end which will take `n-i` units of time (where `n` is the total number of cars).

Finally, we iterate through the arrays `pre` and `suf` to find the minimum combined time taking into account both the left and right removals at each position. The minimum value from this combination gives us the least amount of time needed to remove all the cars containing illegal goods.

Solution Approach

The solution implementation uses a dynamic programming approach with two additional arrays `pre` and `suf` to keep track of the optimal removal times from the left and from the right, respectively.

Here's the detailed implementation of the solution step by step:

1. Initialize the `pre` and `suf` arrays with zeros, and set their sizes to one more than the length of the string `s`. This is done to handle the prefix up to `i` and suffix from `i` inclusively, easily without running into array index issues.

2. Iterate through the string `s` from left to right. If you encounter a `'0'` (no illegal goods), simply copy the value from the previous index of `pre`. This means no additional time is needed to handle this car. If there is a `'1'` (illegal goods exist), calculate the minimum time between the time stored in `pre` at the previous index plus 2 (removing this particular car) and `i + 1` (removing all cars from the left up to the current position) and store that minimum value in `pre[i + 1]`.

   ```
   for i, c in enumerate(s):
       pre[i + 1] = pre[i] if c == '0' else min(pre[i] + 2, i + 1)
   ```

3. Similarly, iterate through the string `s` from right to left to fill in the `suf` array. You do the mirror operation of what was done with `pre`: if a `'0'` is encountered, carry over the value from the right index of `suf`. If a `'1'` is found, calculate the minimum time between the time stored in `suf` at the next index plus 2 (removing this particular car) and `n - i` (removing all cars from the current position to the end).

   ```
   for i in range(n - 1, -1, -1):
       suf[i] = suf[i + 1] if s[i] == '0' else min(suf[i + 1] + 2, n - i)
   ```

4. Once you have the `pre` and `suf` arrays constructed, find the point where the sum of removal times from the left and from the right is minimal. Iterate through the combined view of `pre` (excluding the zeroth position) and `suf` (excluding the last position) and find the minimum sum of corresponding pairs.

   ```
   return min(a + b for a, b in zip(pre[1:], suf[1:]))
   ```

In this approach, the dynamic programming pattern helps avoid redundant calculations by building up the solution using previously computed values, and the two-pointer technique is not explicitly used but is conceptually present as the solution involves evaluating the sum of pairs of prefix and suffix solutions.

Example Walkthrough

Let's illustrate the solution approach with a small example. Assume we have a binary string `s = "001011"` where `1` indicates the presence of illegal goods in a train car.

1. **Initialize `pre` and `suf` Arrays:** First, we initialize `pre` and `suf` arrays to store the minimum removal times from the left and from the right, respectively.

   For `s` having a length of `6`, `pre` and `suf` will be of length `7`.

2. **Fill the `pre` Array:** Next, we iterate from left to right:

   - For `s[0] = '0'`: `pre[1] = pre[0] = 0` (no illegal goods, so no removal time needed).
   - For `s[1] = '0'`: `pre[2] = pre[1] = 0`.
   - For `s[2] = '1'`: `pre[3] = min(pre[2] + 2, 3) = min(0 + 2, 3) = 2` (cheaper to remove this car only).
   - For `s[3] = '0'`: `pre[4] = pre[3] = 2`.
   - For `s[4] = '1'`: `pre[5] = min(pre[4] + 2, 5) = min(2 + 2, 5) = 4` (cheaper to remove both bad cars separately).
   - For `s[5] = '1'`: `pre[6] = min(pre[5] + 2, 6) = min(4 + 2, 6) = 6` (cheaper to remove all cars from the left).

   Now, `pre = [0, 0, 0, 2, 2, 4, 6]`.

3. **Fill the `suf` Array:** We mirror the above process, but from right to left:

   - For `s[5] = '1'`: `suf[5] = min(suf[6] + 2, 1) = min(0 + 2, 1) = 1` (cheaper to remove cars from the right).
   - For `s[4] = '1'`: `suf[4] = min(suf[5] + 2, 2) = min(1 + 2, 2) = 2`.
   - For `s[3] = '0'`: `suf[3] = suf[4] = 2`.
   - For `s[2] = '1'`: `suf[2] = min(suf[3] + 2, 4) = min(2 + 2, 4) = 4` (either way is the same, so minimal is 4).
   - For `s[1] = '0'`: `suf[1] = suf[2] = 4`.
   - For `s[0] = '0'`: `suf[0] = suf[1] = 4`.

   Now, `suf = [4, 4, 4, 2, 2, 1, 0]`.

4. **Find Minimal Combined Time:** Lastly, we iterate through the combined view of `pre` (excluding the zeroth index) and `suf` (excluding the last index) and find the minimum sum.

   - For `i = 1`: sum is `pre[1] + suf[1] = 0 + 4 = 4`.
   - For `i = 2`: sum is `pre[2] + suf[2] = 0 + 4 = 4`.
   - For `i = 3`: sum is `pre[3] + suf[3] = 2 + 2 = 4`.
   - For `i = 4`: sum is `pre[4] + suf[4] = 2 + 2 = 4`.
   - For `i = 5`: sum is `pre[5] + suf[5] = 4 + 1 = 5`.
   - For `i = 6`: sum is `pre[6] + suf[6] = 6 + 0 = 6`.

The minimum time from these sums is `4` units of time, so that is the least amount of time required to remove all the cars containing illegal goods. This can be interpreted as removing the first bad car individually and the other two bad cars from the right end of the train.

Python Solution

```python
class Solution:
    def minimumTime(self, s: str) -> int:
        # Calculate the length of the string s
        length = len(s)

        # Initialize prefix and suffix lists of length + 1 to accommodate edge cases
        prefix = [0] * (length + 1)
        suffix = [0] * (length + 1)

        # Calculate the prefix time cost for each position in the string
        # where cost is minimum of removing '1's by flipping twice each or turning all to '0's up to that point
        for i, char in enumerate(s):
            if char == '0':
                # If current character is '0', no additional cost is added
                prefix[i + 1] = prefix[i]
            else:
                # If current character is '1', choose the lesser cost between adding 2 to the previous cost or
                # flipping all to this position
                prefix[i + 1] = min(prefix[i] + 2, i + 1)

        # Calculate the suffix time cost similarly but in reverse from the end of the string
        for i in range(length - 1, -1, -1):
            if s[i] == '0':
                suffix[i] = suffix[i + 1]
            else:
                suffix[i] = min(suffix[i + 1] + 2, length - i)

        # Find the minimum cost to make the entire string '0's by considering both prefix and suffix costs
        minimum_cost = min(prefix_cost + suffix_cost for prefix_cost, suffix_cost in zip(prefix[1:], suffix[:-1]))

        return minimum_cost

```

Java Solution

```java
class Solution {

    /* This method finds the minimum time required to clear a string `s` of obstacles ('1's) */
    public int minimumTime(String s) {
        // Get the length of the input string s
        int length = s.length();

        // Create prefix and suffix arrays to store the minimum cost to clear up to that index
        int[] prefixCost = new int[length + 1];
        int[] suffixCost = new int[length + 1];

        // Calculate the minimum cost to clear obstacles from the start to each index
        for (int i = 0; i < length; ++i) {
            if (s.charAt(i) == '0') {
                // If there is no obstacle, then the cost is the same as the previous
                prefixCost[i + 1] = prefixCost[i];
            } else {
                // The cost is either removing this obstacle and all before it (i + 1) or
                // turning off the previously considered obstacles and turning this one on (prefixCost[i] + 2)
                prefixCost[i + 1] = Math.min(prefixCost[i] + 2, i + 1);
            }
        }

        // Calculate the minimum cost to clear obstacles from the end to each index
        for (int i = length - 1; i >= 0; --i) {
            if (s.charAt(i) == '0') {
                // If there is no obstacle, then the cost is the same as the next
                suffixCost[i] = suffixCost[i + 1];
            } else {
                // The cost is either removing this obstacle and all after it (n - i) or
                // turning off the subsequently considered obstacles and turning this one on (suffixCost[i + 1] + 2)
                suffixCost[i] = Math.min(suffixCost[i + 1] + 2, length - i);
            }
        }

        // Initialize the answer to the maximum possible value
        int minimumTotalCost = Integer.MAX_VALUE;

        // Find the minimum among the sum of prefix and suffix costs at every split point
        for (int i = 1; i <= length; ++i) {
            minimumTotalCost = Math.min(minimumTotalCost, prefixCost[i] + suffixCost[i]);
        }

        // Return the minimum total cost calculated
        return minimumTotalCost;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int minimumTime(string s) {
        int size = s.size();  // Variable to store the size of the input string.

        vector<int> prefix(size + 1);  // Vector to store the prefix minimum cost.
        vector<int> suffix(size + 1);  // Vector to store the suffix minimum cost.

        // Calculate prefix minimum cost.
        // Loop through the string and fill the 'prefix' vector with the calculated costs.
        for (int i = 0; i < size; ++i) {
            if (s[i] == '0') {
                // If the character is '0', the cost is same as the previous one.
                prefix[i + 1] = prefix[i];
            } else {
                // If the character is '1', take the minimum between the cost of flipping this car
                // and the chain of cars till now, or just removing all cars including this one.
                prefix[i + 1] = min(prefix[i] + 2, i + 1);
            }
        }

        // Calculate suffix minimum cost similarly, from the end of the string.
        for (int i = size - 1; i >= 0; --i) {
            if (s[i] == '0') {
                // Cost is same as the next one for '0' character.
                suffix[i] = suffix[i + 1];
            } else {
                // Minimum between the cost of flipping this car
                // and the chain of cars till now, or just removing all cars including this one.
                suffix[i] = min(suffix[i + 1] + 2, size - i);
            }
        }

        int minCost = INT_MAX;  // Initialize minimum cost to maximum possible integer value.

        // Loop to find the minimum total cost by combining prefix and suffix costs.
        // Iterate through boundaries where prefix and suffix meet.
        for (int i = 0; i <= size; ++i) {
            minCost = min(minCost, prefix[i] + suffix[i]);
        }

        return minCost;  // Return the minimum cost calculated.
    }
};

```

Typescript Solution

```typescript
// Define the minimumTime function which calculates the minimum time required
// to remove or flip cars represented by a string of '0's and '1's.
function minimumTime(s: string): number {
    const size: number = s.length; // Store the size of the input string.

    const prefix: number[] = new Array(size + 1).fill(0); // Array to store the prefix minimum cost.
    const suffix: number[] = new Array(size + 1).fill(0); // Array to store the suffix minimum cost.

    // Calculate prefix minimum cost by iterating over each character in the string.
    for (let i = 0; i < size; ++i) {
        if (s.charAt(i) === '0') {
            // If the character is '0', the cost is the same as the previous one.
            prefix[i + 1] = prefix[i];
        } else {
            // If the character is '1', choose the minimum between the cost
            // of flipping this car and the cost of removing all cars including this one.
            prefix[i + 1] = Math.min(prefix[i] + 2, i + 1);
        }
    }

    // Calculate suffix minimum cost similarly, starting from the end of the string.
    for (let i = size - 1; i >= 0; --i) {
        if (s.charAt(i) === '0') {
            // Cost is the same as the next one for '0' character.
            suffix[i] = suffix[i + 1];
        } else {
            // Choose the minimum between the cost of flipping this car and
            // the cost of removing all cars including this one.
            suffix[i] = Math.min(suffix[i + 1] + 2, size - i);
        }
    }

    let minCost: number = Number.MAX_SAFE_INTEGER; // Initialize minimum cost to the maximum safe integer value.

    // Find the minimum total cost by combining prefix and suffix costs.
    // Iterate through the possible boundaries where prefix and suffix meet.
    for (let i = 0; i <= size; ++i) {
        minCost = Math.min(minCost, prefix[i] + suffix[i]);
    }

    return minCost; // Return the minimum cost calculated.
}

// Example usage:
// const result: number = minimumTime("0010");
// console.log(result); // Outputs the minimum time required to remove or flip cars.

```

Time and Space Complexity

The time complexity of the provided code is `O(n)`, where `n` is the length of the input string `s`. This is because the code iterates over the string twice: once to fill the `pre` array with the minimum operations required to remove '1's from the start of the string and once to fill the `suf` array with the minimum operations required to remove '1's from the end of the string. Each element is computed in constant time, hence the first loop takes `O(n)` time and the second also takes `O(n)` time.

The space complexity of the code is also `O(n)`. Two extra arrays, `pre` and `suf`, each of size `n + 1`, are used to store the intermediate results for the prefix and suffix operations, respectively. The space required by these auxiliary arrays is proportional to the length of the string.