2483. Minimum Penalty for a Shop

Medium String Prefix Sum

Problem Description

In this problem, we're given a string `customers` which represents the customer visit log of a shop over a period of time where each hour is indicated by a character in the string. If `customers[i]` is `'Y'`, then customers visit the shop during the i-th hour. If `customers[i]` is `'N'`, no customers visit during that hour. The goal is to determine the earliest hour to close the shop so that the penalty is minimized.

The penalty is defined as the sum of two components:

1. The number of hours the shop is open but no customers visit (`'N'` when the shop is open).
2. The number of hours the shop is closed but customers do come (`'Y'` when the shop is closed).

The purpose of the problem is to find the closing time that will result in the smallest penalty possible.

Intuition

The intuition behind the solution involves calculating the accumulated customer visits over time and selecting the closure time that minimizes the penalty, using a prefix sum approach.

Here's the reasoning:

1. We first create a prefix sum array `s` that records the cumulative number of customers up to each hour. This allows us to quickly calculate the number of customers who would visit whether the shop is open or closed at any given time.
2. We then iterate over each possible closing hour from `0` to `n` and compute the penalty at that hour. The penalty is the sum of two values:
   - The number of hours the shop was open but had no customers (`j - s[j]`), and
   - The number of customers who would have come if the shop was closed after the j-th hour (`s[-1] - s[j]`).
3. By comparing the penalties for all possible closing hours, we can determine the minimum penalty and the corresponding closing hour.

The key to the solution is understanding that by using the prefix sum of customer visits, we can efficiently compute the penalty for closing at every possible hour without recalculating from scratch each time. This approach allows us to find the optimal closing time with a linear scan and a constant time calculation for each potential closing hour.

Solution Approach

The solution uses an algorithm that leverages the prefix sum pattern to compute the optimal closing time for the shop.

Here's the step-by-step explanation of the implementation based on the reference solution approach:

1. **Initialize the Prefix Sum Array**: We initialize a prefix sum array `s` with a size of `n + 1` (where `n` is the length of the input string `customers`). We need `n + 1` to also take into account the case where the shop does not open at all (closes at hour `0`).
2. **Calculate Prefix Sum**: We iterate through the `customers` string and for each character, we add to the current prefix sum the value `1` if the character is `'Y'`, indicating a customer visit. This is stored in `s[i + 1]`, effectively creating a cumulative count of visits for each hour.
3. **Iterate Over Possible Closing Times**: We then iterate over every possible closing hour `j` from `0` to `n` and calculate the penalty for closing at each of those times. The calculation is as follows:
   - The penalty for hours when the shop is open but no customers come is `j - s[j]`.
   - The penalty for hours when the shop is closed but customers come is `s[-1] - s[j]`.

   The total penalty at hour `j` is the sum of these two values `t = j - s[j] + s[-1] - s[j]`.
4. **Determine Minimum Penalty and Corresponding Hour**: While iterating, we maintain two variables `ans` (answer) and `cost` (current minimum penalty). If the penalty for closing at hour `j` (`t`) is less than the current `cost`, we update the `ans` to `j` and `cost` to `t`. This ensures that by the end of the iteration, `ans` will hold the earliest hour at which the shop should close to minimize the penalty.
5. **Return the Optimal Closing Time**: After completing the iteration over all possible closing hours, `ans` is returned as the final result, representing the optimal closing time.

The implementation makes use of linear time complexity, as it involves single passes through the data: once to build the prefix sum array and once to determine the optimal closing hour. No nested loops are required. The space complexity is linear as well, due to the additional prefix sum array.

```
class Solution:
    def bestClosingTime(self, customers: str) -> int:
        n = len(customers)
        s = [0] * (n + 1)
        for i, c in enumerate(customers):
            s[i + 1] = s[i] + int(c == 'Y')
        ans, cost = 0, inf
        for j in range(n + 1):
            t = j - s[j] + s[-1] - s[j]
            if cost > t:
                ans, cost = j, t
        return ans
```

The use of `inf` in the code represents an infinitely large number to ensure that the first calculated penalty will always be smaller, allowing the algorithm to set the initial minimum penalty properly.

This approach effectively balances the two types of penalties to determine the earliest closing time that minimizes overall penalty.

Example Walkthrough

Let's use a small example to illustrate the solution approach with a given `customers` string "YNYNNY".

First, below is the step-by-step calculation using the aforementioned solution approach:

1. **Initialize the Prefix Sum Array**: We initialize a prefix sum array `s` of size `n + 1 = 7` (since there are 6 hours in the given string). It will start with all zeros: `[0, 0, 0, 0, 0, 0, 0]`.
2. **Calculate Prefix Sum**: Now, we process the customer string "YNYNNY". For each 'Y' we add `1` to the running total of the prefix sum array:
   - For the first hour 'Y', `s` becomes `[0, 1, 1, 1, 1, 1, 1]`.
   - The second hour 'N', no change as no customers came.
   - The third hour 'Y', `s` becomes `[0, 1, 1, 2, 2, 2, 2]`.
   - The fourth hour 'N', no change.
   - The fifth hour 'N', no change.
   - The last hour 'Y', `s` becomes `[0, 1, 1, 2, 2, 2, 3]`.

   By the end, the prefix sum array `s` looks like this: `[0, 1, 1, 2, 2, 2, 3]`.
3. **Iterate Over Possible Closing Times**: Next, we compute the penalty for each possible closing time:
   - Closing at hour `0`, penalty `t = 0 - 0 + 3 - 0 = 3`.
   - Closing at hour `1`, penalty `t = 1 - 1 + 3 - 1 = 2`.
   - Closing at hour `2`, penalty `t = 2 - 1 + 3 - 1 = 3`.
   - Closing at hour `3`, penalty `t = 3 - 2 + 3 - 2 = 2`.
   - Closing at hour `4`, penalty `t = 4 - 2 + 3 - 2 = 3`.
   - Closing at hour `5`, penalty `t = 5 - 2 + 3 - 2 = 4`.
   - Closing at hour `6`, penalty `t = 6 - 3 + 3 - 3 = 3`.
4. **Determine Minimum Penalty and Corresponding Hour**: As we iterate, we find the minimum penalty:
   - After checking all hours, the minimum penalties are `2`, which occur when closing at hour `1` and `3`.
5. **Return the Optimal Closing Time**: Since we want the earliest closing time with the minimum penalty, the answer would be `1`, the first instance where the penalty reaches its minimum value.

By following this implementation, the `bestClosingTime` method would return `1`, which is the hour the shop should close to minimize penalties, given that we have two instances with the same minimum penalty and we choose the earliest.

Solution Implementation

Python

```
class Solution:
    def bestClosingTime(self, customers: str) -> int:
        n = len(customers)
        # Initialize an array to keep track of the cumulative number of customers up to each point
        cumulative_customers = [0] * (n + 1)
        for i, customer in enumerate(customers):
            # Increment cumulative count based on presence of a customer ('Y' or 'N')
            cumulative_customers[i + 1] = cumulative_customers[i] + int(customer == 'Y')

        # Initialize variables for the optimal answer and its associated cost
        best_time, lowest_cost = 0, float('inf')

        # Evaluate each potential closing time to find the one with the lowest cost
        for current_time in range(n + 1):
            # Cost is calculated as the sum of the customers missed before closing
            # and the customers that come after closing
            missed_customers = current_time - cumulative_customers[current_time]
            customers_after_closing = cumulative_customers[-1] - cumulative_customers[current_time]
            total_cost = missed_customers + customers_after_closing

            # If the current cost is lower than the lowest known cost, update best_time and lowest_cost
            if lowest_cost > total_cost:
                best_time, lowest_cost = current_time, total_cost

        # Return the best time to close that minimizes the total cost
        return best_time
```

Java

```
class Solution {

    // Method to determine the best closing time for the customers
    public int bestClosingTime(String customers) {
        // Get the length of the customers string which depicts whether a customer is present (Y) or not (N)
        int length = customers.length();
        // Create an array to store the cumulative sum of customers up to each point
        int[] cumulativeSum = new int[length + 1];

        // Calculate the cumulative sum of customers
        for (int i = 0; i < length; ++i) {
            cumulativeSum[i + 1] = cumulativeSum[i] + (customers.charAt(i) == 'Y' ? 1 : 0);
        }

        // Initialize variables to store the best closing time and its associated cost
        int bestTime = 0;
        int lowestCost = Integer.MAX_VALUE;

        // Evaluate each potential closing time from 0 to n to find the minimum cost
        for (int i = 0; i <= length; ++i) {
            // Cost for each time is calculated by the number of customers before i and
            // the number of vacant time slots after i
            int cost = i - cumulativeSum[i] + (cumulativeSum[length] - cumulativeSum[i]);

            // If the cost for this time is lower than the current lowest cost, update bestTime and lowestCost
            if (lowestCost > cost) {
                bestTime = i;
                lowestCost = cost;
            }
        }

        // Return the best closing time that minimizes the cost
        return bestTime;
    }
}
```

C++

```
class Solution {
public:
    // Calculates the best time to close the shop based on the customer presence data.
    int bestClosingTime(string customers) {
        int totalCustomers = customers.size();

        // The prefix sum array where s[i] represents the number of 'Y' up to index i.
        vector<int> prefixSum(totalCustomers + 1, 0);
        for (int i = 0; i < totalCustomers; ++i) {
            prefixSum[i + 1] = prefixSum[i] + (customers[i] == 'Y' ? 1 : 0);
        }

        int bestTime = 0; // To store the best time to close.
        int minCost = INT_MAX; // Initialize it with the maximum possible value.

        // Consider closing at each possible time and calculate the cost.
        for (int currentTime = 0; currentTime <= totalCustomers; ++currentTime) {
            // Cost is the number of customers who have not arrived yet plus
            // the number of customers who arrived after currentTime.
            int cost = currentTime - prefixSum[currentTime] + prefixSum[totalCustomers] - prefixSum[currentTime];

            // If the new cost is smaller, update the best closing time and minimum cost.
            if (minCost > cost) {
                bestTime = currentTime;
                minCost = cost;
            }
        }

        // Return the best closing time after iterating through all possibilities.
        return bestTime;
    }
};
```

TypeScript

```
// Calculates the best time to close the shop based on the customer presence data.
function bestClosingTime(customers: string): number {
    const totalCustomers = customers.length;

    // The prefix sum array where prefixSum[i] represents the number of 'Y' up to index i.
    const prefixSum = new Array(totalCustomers + 1).fill(0);
    for (let i = 0; i < totalCustomers; ++i) {
        prefixSum[i + 1] = prefixSum[i] + (customers[i] === 'Y' ? 1 : 0);
    }

    let bestTime = 0; // To store the best time to close.
    let minCost = Number.MAX_SAFE_INTEGER; // Initialize it with the maximum safe integer value in JavaScript.

    // Consider closing at each possible time and calculate the cost.
    for (let currentTime = 0; currentTime <= totalCustomers; ++currentTime) {
        // Cost is the number of customers who have not arrived yet plus
        // the number of customers who arrived after currentTime.
        const cost = currentTime - prefixSum[currentTime] + prefixSum[totalCustomers] - prefixSum[currentTime];

        // If the new cost is smaller, update the best closing time and minimum cost.
        if (minCost > cost) {
            bestTime = currentTime;
            minCost = cost;
        }
    }

    // Return the best closing time after iterating through all possibilities.
    return bestTime;
}
```

```
class Solution:
    def bestClosingTime(self, customers: str) -> int:
        n = len(customers)
        # Initialize an array to keep track of the cumulative number of customers up to each point
        cumulative_customers = [0] * (n + 1)
        for i, customer in enumerate(customers):
            # Increment cumulative count based on presence of a customer ('Y' or 'N')
            cumulative_customers[i + 1] = cumulative_customers[i] + int(customer == 'Y')

        # Initialize variables for the optimal answer and its associated cost
        best_time, lowest_cost = 0, float('inf')

        # Evaluate each potential closing time to find the one with the lowest cost
        for current_time in range(n + 1):
            # Cost is calculated as the sum of the customers missed before closing
            # and the customers that come after closing
            missed_customers = current_time - cumulative_customers[current_time]
            customers_after_closing = cumulative_customers[-1] - cumulative_customers[current_time]
            total_cost = missed_customers + customers_after_closing

            # If the current cost is lower than the lowest known cost, update best_time and lowest_cost
            if lowest_cost > total_cost:
                best_time, lowest_cost = current_time, total_cost

        # Return the best time to close that minimizes the total cost
        return best_time
```

Time and Space Complexity

The given Python code is designed to determine the best time to close a store based on when customers are inside the store. The input is a string where each character represents a time unit and indicates whether a customer is present at that time (`'Y'` for yes, `'N'` for no).

Time Complexity:

We analyze the time complexity by considering the operations performed within the function:

- Initializing `s` with the length of the customers string: $O(1)$.
- Creating a prefix sum array `s` of length $n+1$: This takes $O(n)$ time as it involves iterating over the entire `customers` string once.
- The loop to calculate the best closing time:
  - We have a loop that iterates $n+1$ times.
  - Inside the loop, all operations (calculating `t`, comparison, and assignment) are constant time $O(1)$.

The predominant factor in the time complexity is the loop that runs $n+1$ times with constant-time operations within it.

Consequently, the time complexity of the code is:

- $O(n + 1) \times O(1) = O(n)$

Space Complexity:

For space complexity, we consider the additional space used by the algorithm apart from the input:

- The array `s` that holds the prefix sums is of length $n+1$, so it requires $O(n)$ space.
- Variables `ans`, `cost`, and `t` use constant space $O(1)$.

Thus, the space complexity of the code is:

- $O(n) + O(1) = O(n)$