2055. Plates Between Candles

Medium Array String Binary Search Prefix Sum

Leetcode Link

Problem Description

In this problem, we are given a string `s` representing a table where `'*'` characters signify plates and `'|'` characters represent candles placed on the table. Alongside the string, we receive an array of queries `queries`, with each query given as a pair of indices `[left_i, right_i]`. This pair indicates a specific substring of `s` — from index `left_i` to `right_i`, inclusive.

The task is to determine how many plates are situated between candles within these substrings. It's important to note that a plate is only counted if it has at least one candle to its left and one candle to its right within the boundary of the substring defined by the query.

Our goal is to return an array of integers where each element corresponds to the count of plates between candles for each query in `queries`.

Intuition

The brute force approach to solving this would entail scanning the substring for each query to count the plates between candles, which could lead to a significant amount of repeated work, especially with overlapping or similar queries.

To handle this problem efficiently, we employ a prefix sum array and two additional arrays to track the positions of the leftmost and rightmost candles relative to each position.

The prefix sum array, `presum`, gets constructed such that `presum[i]` stores the sum of plates encountered from the start of the string up to index `i - 1`. This allows us to calculate the number of plates between any two candle positions quickly by subtracting the appropriate prefix sums.

However, before we can use this array, we need to know where the boundaries are — the closest candles to the left and right of each plate. For that, we introduce two arrays: `left` and `right`. As we iterate through the string, we update `left[i]` with the last seen candle position on the left of index `i` and `right[i]` with the upcoming candle position on the right of index `i`. These arrays help us locate the nearest candles for any given plate quickly.

With these arrays ready, we can now process each query. We take the given query range `[l, r]` and use the `right` array to find the first candle to the right of `l` and the `left` array to find the last candle to the left of `r`. If these exist and are properly positioned (the right of `l` is less than the left of `r`), we've identified the boundaries of the relevant candle-plate-candle sequence. Then we can use the differences of the prefix sums to find the number of plates between these two candles.

This approach allows us to process queries efficiently since we reduced the problem to constant-time look-ups and a single subtraction operation per query, after the initial setup of the prefix sum and left/right boundary arrays.

Solution Approach

The solution approach breaks down into several key steps corresponding to the following algorithmic patterns and data structures:

Step 1: Prefix Sum Array

The first step is constructing the `presum` array. It accumulates the count of plates (`'*'`) as we iterate through the string `s`. `presum[i]` is computed as `presum[i - 1] + (s[i - 1] == '*')`. This setup will later enable us to quickly calculate the number of plates between any two indices.

Step 2: Leftmost and Rightmost Candle Arrays

We initialize two arrays, `left` and `right`, to record the positions of the nearest candles on the left and right sides of each index in `s`.

- As we scan the string from left to right, each `left[i]` is updated to `i` if `s[i]` is a candle. If not, it carries over the position of the last encountered candle.
- Conversely, scanning from right to left, `right[i]` is set to `i` if `s[i]` is a candle. Otherwise, it retains the position of the next candle to the right.

The purpose of these arrays is to determine, for each plate, the positions of the closest candles that potentially enclose it in a valid sequence.

Step 3: Query Processing

For each query `[l, r]` in `queries`:

- We find the closest candle to the right of `l` using `right[l]` and the closest candle to the left of `r` using `left[r]`. Let's refer to these positions as `i` and `j`, respectively.
- We then check if `i` and `j` form a valid boundary — that is, if `i` is before `j`. If not, no plates can be counted for this query.
- If they do form a valid boundary, the number of plates between these candles is the difference between the `presum` just after the left candle and just before the right candle: `presum[j] - presum[i + 1]`.
- This result is stored in the `ans` array at the position corresponding to the current query.

The final output is the `ans` array, which contains the count of plates between candles for each query.

The algorithm leverages prefix sums, array scanning, and good use of boundary tracking to optimize the repeated calculations involved in answering each query. This results in an efficient solution that only requires a linear scan of the input string and the queries.

Example Walkthrough

Let's take an example to illustrate the solution approach:

Given a string `s = "||*||**|*"` and queries `queries = [[2, 5], [1, 8]]`. *We want to count the number of plates ('*') that are located between candles ('|') for each query range.*

Step 1: Prefix Sum Array

We create the `presum` array as follows:

- `s = "||*||**|*"`
- `presum = [0, 0, 0, 1, 2, 3, 3, 4, 5, 5]`

Each element in `presum` indicates the total number of plates to the left of it.

Step 2: Leftmost and Rightmost Candle Arrays

Next, we construct the `left` and `right` arrays:

- `left = [-1, -1, 2, 2, 2, 2, 6, 6, 6, 6]`
- `right = [2, 2, 2, 6, 6, 6, 6, 9, 9, 9]`

The `left` array is constructed by scanning from left to right and marking the latest position of a candle. Similarly, `right` is constructed by scanning from right to left.

Step 3: Query Processing

Now we answer each query:

1. For the query `[2, 5]`, we check:
   - Nearest right candle from 2 (inclusive): `right[2] = 2`
   - Nearest left candle from 5 (inclusive): `left[5] = 2`
   - Since `right[2]` equals `left[5]`, there are no valid boundaries within this substring, so the result for this query is 0.
2. For the query `[0, 9]`, we check:
   - Nearest right candle from 0: `right[0] = 2`
   - Nearest left candle from 9: `left[9] = 6`
   - We have a valid boundary because `2 < 6`.
   - The number of plates is `presum[6] - presum[2 + 1]` which equals `3 - 1 = 2`. There are 2 plates between the candles at positions 2 and 6.

After processing both queries, our `ans` array, which holds the resulting counts, is `[0, 2]`.

This example completes the walk-through of the solution approach using an example string and a couple of queries. The algorithm efficiently computes the desired count using prefix sums, and nearest left/right candle tracking to avoid repeated calculations for each query.

Python Solution

```python
from typing import List

class Solution:
    def platesBetweenCandles(self, s: str, queries: List[List[int]]) -> List[int]:
        n = len(s)  # Length of the string 's'
        m = len(queries)  # Number of queries

        # Initialize a prefix sum array that will count the number of plates ('*') up to index 'i'
        prefix_sum = [0] * (n + 1)
        for i, char in enumerate(s):
            # Build the prefix sum array
            prefix_sum[i + 1] = prefix_sum[i] + (char == '*')

        # Initialize arrays to store the index of the closest candle to the left and right
        closest_left_candle = [0] * n
        left_candle_index = -1  # Initialize to -1 as no candle is seen yet
        for i, char in enumerate(s):
            # Update the index when a candle is found
            if char == '|':
                left_candle_index = i
            closest_left_candle[i] = left_candle_index

        # Populate the closest_right_candle array in reverse
        for i in range(n - 1, -1, -1):
            if s[i] == '|':
                right_candle_index = i
            closest_right_candle[i] = right_candle_index

        # Initialize the answer array, one entry for each query
        answer = [0] * len(queries)

        # Process each query
        for query_index, (start, end) in enumerate(queries):
            # Find the closest right candle for the left index
            # And the closest left candle for the right index
            candle_to_the_right = closest_right_candle[start]
            candle_to_the_left = closest_left_candle[end]

            # If valid candle indexes are found and they do not overlap
            # (a candle to the right exists before a candle to the left)
            # Calculate the number of plates between these candles
            if candle_to_the_right != -1 and candle_to_the_left != -1 and candle_to_the_right < candle_to_the_left:
                answer[query_index] = prefix_sum[candle_to_the_left] - prefix_sum[candle_to_the_right + 1]

        # Return the result list containing the number of plates for each query
        return answer
```

Java Solution

```java
class Solution {
    // Method to calculate the number of plates between candles based on a set of queries
    public int[] platesBetweenCandles(String s, int[][] queries) {
        int length = s.length(); // Length of the string

        // Array to hold the prefix sum of plates up to each position
        int[] prefixSum = new int[length + 1];
        for (int i = 0; i < length; ++i) {
            // Increment the count if a plate is found
            prefixSum[i + 1] = prefixSum[i] + (s.charAt(i) == '*' ? 1 : 0);
        }

        // Arrays to hold the index of the nearest left and right candles to each position
        int[] nearestLeftCandle = new int[length];
        int[] nearestRightCandle = new int[length];

        // Calculate the nearest left candle positions
        for (int i = 0, lastCandleIndex = -1; i < length; ++i) {
            // Update the last candle index if a candle is found
            if (s.charAt(i) == '|') {
                lastCandleIndex = i; // Update last seen candle
            }
            nearestLeftCandle[i] = lastCandleIndex;
        }

        // Calculate the nearest right candle positions
        for (int i = length - 1, lastCandleIndex = -1; i >= 0; --i) {
            if (s.charAt(i) == '|') {
                lastCandleIndex = i; // Update last seen candle
            }
            nearestRightCandle[i] = lastCandleIndex;
        }

        // Array to hold the results of queries
        int[] answer = new int[queries.length];

        // Iterate over each query to determine the number of plates between candles
        for (int i = 0; i < queries.length; ++i) {
            // Find the nearest right candle index from the start of the current query
            int startIndex = nearestRightCandle[queries[i][0]];
            // Find the nearest left candle index from the end of the current query
            int endIndex = nearestLeftCandle[queries[i][1]];

            // If valid candle indices are found and the start index is before the end index
            // (a candle to the right exists before a candle to the left)
            // Calculate the number of plates by subtracting the prefix sums
            if (startIndex != -1 && endIndex != -1 && startIndex < endIndex) {
                answer[i] = prefixSum[endIndex] - prefixSum[startIndex + 1];
            }
        }

        return answer;
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <string>

using std::vector;
using std::string;

class Solution {
public:
    // Function to calculate the number of plates between candles
    vector<int> platesBetweenCandles(string s, vector<vector<int>>& queries) {
        int n = s.size();
        // Create a prefix sum array to keep track of number of plates
        vector<int> prefixSum(n + 1, 0);
        // Calculating the prefix sum of the plates
        for (int i = 0; i < n; ++i) {
            prefixSum[i + 1] = prefixSum[i] + (s[i] == '*');
        }

        // Arrays to store the nearest left and right candle positions
        vector<int> nearestLeftCandle(n);
        vector<int> nearestRightCandle(n);

        // Find the nearest left candle for each position
        for (int i = 0, lastCandleIndex = -1; i < n; ++i) {
            if (s[i] == '|') lastCandleIndex = i;
            nearestLeftCandle[i] = lastCandleIndex;
        }
        // Find the nearest right candle for each position
        for (int i = n - 1, lastCandleIndex = -1; i >= 0; --i) {
            if (s[i] == '|') lastCandleIndex = i;
            nearestRightCandle[i] = lastCandleIndex;
        }

        // Answer vector to store the result for each query
        vector<int> answer(queries.size());
        // Processing each query
        for (int k = 0; k < queries.size(); ++k) {
            // Find the nearest right candle from the start index of the query
            int startCandleIndex = nearestRightCandle[queries[k][0]];
            // Find the nearest left candle from the end index of the query
            int endCandleIndex = nearestLeftCandle[queries[k][1]];
            // If both candles are valid and the start candle is to the left of the end candle
            if (startCandleIndex != -1 && endCandleIndex != -1 && startCandleIndex < endCandleIndex) {
                // Calculate the number of plates between candles and store it in the answer
                answer[k] = prefixSum[endCandleIndex] - prefixSum[startCandleIndex + 1];
            }
        }
        return answer;
    }
};
```

Typescript Solution

```typescript
// Importing necessary libraries
import { Vector, String } from "prelude-ts";

// Function to calculate the number of plates between candles
function platesBetweenCandles(s: String, queries: number[][]): number[] {
    const n = s.length(); // Length of the string
    // Create a prefix sum array to keep track of the number of plates
    const prefixSum: number[] = new Array(n + 1).fill(0);

    // Calculating the prefix sum for the plates
    for (let i = 0; i < n; ++i) {
        prefixSum[i + 1] = prefixSum[i] + (s[i] == '*' ? 1 : 0);
    }

    // Arrays to store the nearest left and right candle positions
    const nearestLeftCandle: number[] = new Array(n);
    const nearestRightCandle: number[] = new Array(n);

    // Find the nearest left candle for each position
    let lastCandleIndex: number = -1;
    for (let i = 0; i < n; ++i) {
        if (s[i] == '|') lastCandleIndex = i;
        nearestLeftCandle[i] = lastCandleIndex;
    }

    // Find the nearest right candle for each position
    lastCandleIndex = -1;
    for (let i = n - 1; i >= 0; --i) {
        if (s[i] == '|') lastCandleIndex = i;
        nearestRightCandle[i] = lastCandleIndex;
    }

    // Answer vector to store the result for each query
    const answer: number[] = new Array(queries.length);

    // Process each query
    for (let k = 0; k < queries.length; ++k) {
        // Find the nearest right candle from the start index of the query
        const startCandleIndex = nearestRightCandle[queries[k][0]];
        // Find the nearest left candle from the end index of the query
        const endCandleIndex = nearestLeftCandle[queries[k][1]];

        // If both candles are valid and the start candle is to the left of the end candle
        if (startCandleIndex !== -1 && endCandleIndex !== -1 && startCandleIndex < endCandleIndex) {
            // Calculate the number of plates between candles and store it in the answer
            answer[k] = prefixSum[endCandleIndex] - prefixSum[startCandleIndex + 1];
        } else {
            // In case there is no valid segment of plates between candles, the result is 0
            answer[k] = 0;
        }
    }

    return answer;
}
```

Time and Space Complexity

The given Python code defines a method `platesBetweenCandles` to calculate the number of plates between candles for a series of queries.

Time Complexity:

Let's break it down step by step:

1. We first compute the prefix sum array `presum`, which takes $O(n)$ time, where n is the length of the string `s`.
2. Then we create two arrays `left` and `right` that store the index of the nearest left and right candle for each position in the string. Building these arrays takes $O(n)$ time each since we iterate over all elements from left to right and right to left, respectively.
3. Finally, we iterate over all the queries. For each query, it takes $O(1)$ time to compute the answer using the prefix sums and the `left` and `right` arrays. Since there are q queries, this step takes $O(q)$ time in total.

Therefore, the overall time complexity of the code is $O(n) + O(n) + O(q) = O(n + q)$.

Space Complexity:

1. We use $O(n)$ space for the prefix sum array `presum` of size $n + 1$.
2. We also use $O(n)$ space for each of the `left` and `right` arrays.
3. Finally, the space for output ans is $O(q)$ where q is the number of queries.

Therefore, the total space used is $O(n) + O(n) + O(n) + O(q) = O(n + q)$.

In conclusion, the time complexity of the code is $O(n + q)$, and the space complexity is $O(n + q)$, where n is the length of the string and q is the number of queries.