2555. Maximize Win From Two Segments

Medium Array Binary Search Sliding Window Leetcode Link

Problem Description

In this problem, you're presented with a number line where various prizes are located at different positions, represented by the sorted integer array `prizePositions`. Your goal is to collect as many of those prizes as possible by selecting two segments on the number line. Each segment is defined by integer endpoints and has a fixed length `k`. Any prize positioned within these segments or on their boundaries is considered collected.

The key complication is that there are constraints on the size of the segments; both must be exactly `k` units long, which means you have to be strategic about where you place them. Two segments may even overlap, allowing for potential strategies where you might cover a dense cluster of prizes with the intersection of both segments. The task is to determine the maximum number of prizes you can win by choosing the two segments with optimal positions.

Intuition

The intuition behind the solution is to utilize a sliding window approach where we iterate over the prizes, keeping track of the number of prizes that can be collected with a segment ending at the current prize's position. We can do this efficiently because the `prizePositions` array is sorted.

As we iterate through the prizes, we use binary search, implemented in Python as `bisect_left`, to find the leftmost position in the `prizePositions` array where we can start a segment of length `k` that ends at the current prize's position (`x`). This allows us to calculate how many prizes we can collect for every possible ending position of a segment.

Since we're allowed to have two segments, we need to determine the best way to combine two such segments to maximize the number of prizes. While iterating, we maintain an array `f` that keeps track of the best solution for the first segment ending at each position. As we compute the maximum prizes for a segment ending at position `x`, we also consider the best solution (`f[j]`) for the segment that would end right before the start of the current segment.

By updating `ans` with `f[j] + i - j`, we continuously keep track of the best combination of two segments up to the current prize's position. `i - j` represents the number of prizes we collect from the current segment, and `f[j]` the most prizes collected from the first segment that does not overlap with the current one. `f[i]` is then updated to be the maximum of the previous value or `i - j`, ensuring `f` remains accurate as we progress.

Solution Approach

The solution follows a dynamic programming strategy with the aid of binary search to efficiently handle the sorted nature of the `prizePositions`. Here's a walkthrough explaining the role of different parts of the code:

1. **Initialization**:
   - `n` is assigned to be the length of the `prizePositions` list and signifies the total number of prizes.
   - `f` is then initialized to be a list of zeros with a length one more than the number of prizes (`n + 1`), where `f[i]` will store the maximum number of prizes that can be obtained by the first segment ending at `prizePositions[i - 1]`. The extra entry allows handling the case where no prizes fit into the first segment.
   - `ans` is initialized to 0, which will ultimately hold the maximum number of prizes obtained with two segments.
2. **Iterating over prizes**:
   - The `for` loop iterates through `prizePositions` with `x` representing the position of the current prize and `i` the index (1-based).
   - Using `bisect_left(prizePositions, x - k)`, we perform a binary search to find the index `j` where a segment of length `k` could start such that it will end exactly at the current prize position `x`. Since our list is 1-based in this scope, `j` actually represents the number of prizes that are unreachable by the segment ending at `x`.
3. **Calculating the maximum number of prizes**:
   - `ans` is updated to check if the current combination of the first segment ending right before `j` and the second segment ending at `x` yields a higher prize count than previously recorded. Here, `f[j]` gives the number of prizes for the optimal first segment and `i - j` represents the number of prizes collected by a segment ending at `x` starting from position `j`.
   - `f[i]` is then updated to hold the maximum number of prizes that can be collected with this current segment as the potential first segment for future iterations: `f[i] = max(f[i - 1], i - j)`.

The algorithm utilizes dynamic programming to remember previous results and applies binary search to make it efficient for finding start points of segments within the sorted prize positions. The choice of `f` being a list of `n + 1` integers and the use of binary search is crucial for keeping the overall time complexity low as it avoids any need for nested loops, and each lookup or operation is done in constant to logarithmic time, leading to an overall time complexity of O(n log n).

The final result, `ans`, gives us the maximum number of prizes selectable with two optimally placed segments of length `k`.

Example Walkthrough

Let's illustrate this solution approach with a small example. Suppose we're given the sorted integer array `prizePositions` and a segment length `k`.

```
prizePositions = [1, 3, 4, 8, 10]
k = 4
```

1. **Initialization**:
   - The list has 5 positions, so `n = 5`.
   - We have `f = [0, 0, 0, 0, 0, 0]` to capture the maximum prizes for any first segment ending at each prize index.
   - `ans` starts with a value of 0.
2. **Iterating over prizes**:
   - We start the loop, for each `x` in `prizePositions`, `i` is the index starting at 1.
   - At `i = 1`, `x = 1`: We can't fit any segment that ends at `1` and is 4 units long because the segment would start before the number line.
   - At `i = 2`, `x = 3`: A segment of length 4 can start at position `prizePositions[0]`, so it collects 2 prizes (`[1, 3]`).
3. **Calculating the maximum number of prizes**:
   - For `i = 2`, using binary search, `j = bisect_left(prizePositions, 3 - 4)`, which gives `j = 0` since no segment of length 4 can end before 3.
   - Since this is the first viable segment, `ans` is updated to be `f[j] + (i - j)`, which is `0 + (2 - 0) = 2`.
   - Update `f[i]` to be `max(f[i - 1], i - j)`, so `f[2]` becomes `max(f[1], 2 - 0)` which is `2`.

Continuing this process:

- At `i = 3`, `x = 4`: A segment of length 4 can start at position `prizePositions[0]`, collecting 3 prizes (`[1, 3, 4]`) with `j = 0`. Now `ans` is updated to `3`.
- `f[3]` becomes `3` as `max(f[2], 3 - 0)`.
- At `i = 4`, `x = 8`: A segment of length 4 can start at `prizePositions[1] = 3`, collecting 2 prizes (`[3, 4]`) with `j = 1`. Now `ans` compares `2 (f[1]) + 2` with `3` and keeps `3`.
- `f[4]` takes the higher of `f[3]` or `3 - 1`, so it becomes `3`.
- At `i = 5`, `x = 10`: A segment of length 4 can start at `prizePositions[2] = 4`, collecting 3 prizes (`[4, 8, 10]`) with `j = 2`. Here `ans` compares `3 (f[2]) + 3` with current `ans` and updates it to `6`.
- Finally, `f[5]` becomes `max(f[4], 5 - 2)` which is `3`.

At the end of the process, `ans` gives us `6`, which is the maximum number of prizes we can collect by optimally placing two segments of length `k` (each collecting `[1, 3, 4]` and `[4, 8, 10]`, with overlap on the prize at position `4`).

The solution efficiently finds the maximum number of prizes using sliding windows and binary search within the constraints given, leading to an optimal pairing of segments with potentially overlapping positions.

Python Solution

```python
from bisect import bisect_left

class Solution:
    def maximizeWin(self, prize_positions: List[int], k: int) -> int:
        # Get the number of available positions
        num_positions = len(prize_positions)
        # Initialize the dp (dynamic programming) array with zeroes
        dp = [0] * (num_positions + 1)
        # 'max_win' will store the maximum number of wins attainable
        max_win = 0

        # Iterate through the prize positions
        for i, position in enumerate(prize_positions, 1):
            # Find the leftmost position in 'prize_positions' where the player could have been
            # to ensure the distance between consecutive prizes is at most 'k'
            j = bisect_left(prize_positions, position - k)

            # The current win is calculated by previous win at 'j' plus the number of positions
            # between 'j' and the current position 'i', update 'max_win' accordingly
            max_win = max(max_win, dp[j] + i - j)

            # Update the dp array with the maximum of the previous value or the new calculated win
            dp[i] = max(dp[i - 1], i - j)

        # Finally, return the maximum number of wins after processing all positions
        return max_win

```

Java Solution

```java
class Solution {

    // Method to maximize the win given a set of prize positions and a distance k.
    public int maximizeWin(int[] prizePositions, int k) {
        int n = prizePositions.length; // Total number of prizes.
        int[] dp = new int[n + 1]; // Dynamic programming array.
        int maxPrizes = 0; // Variable to keep track of the maximum number of prizes.

        // Loop through each prize position.
        for (int i = 1; i <= n; ++i) {
            int currentPosition = prizePositions[i - 1]; // Current position under consideration.
            int j = binarySearch(prizePositions, currentPosition - k); // Finding the eligible position.
            // Update the maximum number of prizes.
            maxPrizes = Math.max(maxPrizes, dp[j] + i - j);
            // Update the DP array with the maximum prizes up to the current index.
            dp[i] = Math.max(dp[i - 1], i - j);
        }
        return maxPrizes; // Return the maximum number of prizes that can be won.
    }

    // Binary search to find the index of the smallest number greater than or equal to x.
    private int binarySearch(int[] nums, int x) {
        int left = 0, right = nums.length;

        // Perform the binary search.
        while (left < right) {
            int mid = (left + right) >> 1; // Find the middle index.
            // Check if the middle element is greater than or equal to x.
            if (nums[mid] >= x) {
                right = mid; // Adjust the right boundary.
            } else {
                left = mid + 1; // Adjust the left boundary.
            }
        }
        // Return the index of the left boundary, which gives the desired position.
        return left;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm> // for std::lower_bound and std::max

class Solution {
public:
    // Function to find the maximum number of prizes that can be won in a game,
    // given the prize positions and the maximum distance k a player can move.
    int maximizeWin(std::vector<int>& prizePositions, int k) {
        int n = prizePositions.size(); // Get the number of prize positions
        std::vector<int> memo(n + 1); // Create a DP array for memoization, initialized to n+1 elements
        int maxPrizes = 0; // This will hold the maximum number of prizes that can be won

        // Iterate over each prize position
        for (int i = 1; i <= n; ++i) {
            int currentPosition = prizePositions[i - 1]; // Get the current prize position

            // Find the first position that is within the distance k from currentPosition
            auto it = lower_bound(prizePositions.begin(), prizePositions.end(), currentPosition - k);
            int j = it - prizePositions.begin(); // Find the index of the position

            // Update maxPrizes to be the larger value between its current value and the prizes won if starting from j
            maxPrizes = std::max(maxPrizes, memo[j] + i - j);

            // Update the DP table with the maximum of the current value or the i - j prizes
            memo[i] = std::max(memo[i - 1], i - j);
        }

        return maxPrizes; // Return the maximum number of prizes that can be won
    }
};

```

Typescript Solution

```typescript
// Maximizes the number of prizes won by jumping exactly k positions to the right each time
function maximizeWin(prizePositions: number[], k: number): number {
    const prizeCount = prizePositions.length;
    // Array to hold the maximum number of prizes that can be won up to a certain position
    const maxPrizes: number[] = Array(prizeCount + 1).fill(0);
    let maxWin = 0;

    // Searches for the leftmost position where the player can jump from to reach the current position or beyond
    const binarySearch = (targetPosition: number): number => {
        let left = 0;
        let right = prizeCount;
        while (left < right) {
            const mid = (left + right) >> 1;
            if (prizePositions[mid] >= targetPosition) {
                right = mid;
            } else {
                left = mid + 1;
            }
        }
        return left;
    };

    // Iterate through all prize positions to calculate the maximum number of prizes that can be won
    for (let i = 1; i <= prizeCount; ++i) {
        const currentPosition = prizePositions[i - 1];
        const jumpFromIndex = binarySearch(currentPosition - k);
        maxWin = Math.max(maxWin, maxPrizes[jumpFromIndex] + i - jumpFromIndex);
        maxPrizes[i] = Math.max(maxPrizes[i - 1], i - jumpFromIndex);
    }

    return maxWin;
}

```

Time and Space Complexity

The given Python code aims to determine the maximum number of prizes one can win within a window or gap of '`k`' positions.

Time Complexity:

The function `maximizeWin` iterates once over the `prizePositions` list, which contains '`n`' elements. For every element '`x`' in 'prizePositions', there is a call to `bisect_left`, which performs binary search on the sorted list of prize positions, having a time complexity of `O(log n)`.

The loop runs for '`n`' times, so considering both the loop and the binary search inside it, the overall time complexity is `O(n log n)`, where '`n`' is the number of elements in the `prizePositions` list.

Space Complexity:

Space complexity is determined by the extra space used in addition to the input data size. The algorithm allocates a list '`f`' with '`n + 1`' elements, where '`n`' is the length of `prizePositions`.

Thus, the space complexity of the code is `O(n)`, which is required for the auxiliary list '`f`'. Since this list scales linearly with the input size, the usage of constant extra variables is negligible compared to the size of '`f`'.