452. Minimum Number of Arrows to Burst Balloons

Medium Greedy Array Sorting Leetcode Link

Problem Description

In this LeetCode problem, we are presented with a scenario involving a number of spherical balloons that are taped to a wall, represented by the XY-plane. Each balloon is specified by a pair of integers `[x_start, x_end]`, which represent the horizontal diameter of the balloon on the X-axis. However, the balloons' vertical positions, or Y-coordinates, are unknown.

We are tasked with finding the minimum number of arrows that need to be shot vertically upwards along the Y-axis from different points on the X-axis to pop all of the balloons. An arrow can pop a balloon if it is shot from a point `x` such that `x_start <= x <= x_end` for that balloon. Once an arrow is shot, it travels upwards infinitely, bursting any balloon that comes in its path.

The goal is to determine the smallest number of arrows necessary to ensure that all balloons are popped.

Intuition

To solve this problem, we need to look for overlapping intervals among the balloons' diameters. If multiple balloons' diameters overlap with each other, a single arrow can burst all of them.

We can approach this problem by:

1. Sorting the balloons based on their `x_end` value. This allows us to organize the balloons in a way that we always deal with the balloon that ends first. By doing this, we ensure that we can shoot as many balloons as possible with a single arrow.
2. Scanning through the sorted balloons and initializing `last`, the position of the last shot arrow, to negative infinity (since we haven't shot any arrow yet).
3. For each balloon in the sorted list, we check if the balloon's `x_start` is greater than `last`, which would mean this balloon doesn't overlap with the previously shot arrow's range and requires a new arrow. If so, we increment the arrow count `ans` and update `last` with the balloon's `x_end`, marking the end of the current arrow's reach.
4. If a balloon's start is within the range of the last shot arrow `(x_start <= last)`, it is already burst by the previous arrow, so we don't need to shoot another arrow.
5. We keep following step 3 and 4 until all balloons are checked. The arrow count `ans` then gives us the minimum number of arrows required to burst all balloons.

By the end of this process, we have efficiently found the minimum number of arrows needed, which is the solution to the problem.

Solution Approach

The implementation of the solution involves a greedy algorithm, which is one of the standard strategies to solve optimization problems. Here, the algorithm tests solutions in sequence and selects the local optimum at each step with the hope of finding the global optimum.

In this specific case, the greedy choice is to sort balloons by their right bound and burst as many balloons as possible with one arrow before moving on to the next arrow. The steps of the algorithm are implemented as follows in the given Python code:

1. First, a sort operation is applied on the `points` list. The key for sorting is set to `lambda x: x[1]`, which sorts the balloons in ascending order based on their ending x-coordinates (`x_end`).

   ```
   sorted(points, key=lambda x: x[1])
   ```

2. A variable `ans` is initialized to 0 to keep track of the total number of arrows used.
3. Another variable `last` is initialized to negative infinity (`-inf`). This variable is used to store the x-coordinate of the last shot arrow that will help us check if the next balloon can be burst by the same arrow or if we need a new arrow.
4. A `for` loop iterates through each balloon in the sorted list `points`. The loop checks if the current balloon's start coordinate is greater than `last`. If the condition is true, it implies that the current arrow cannot burst this balloon, hence we increment `ans` and set `last` to this balloon's end coordinate:

   ```
   last = b
   ```

   This ensures that any subsequent balloon that starts before `b` (the current `last`) can be burst by the current arrow.
5. If the start coordinate of the balloon is not greater than `last`, it means the balloon overlaps with the range of the current arrow and will be burst by it, so `ans` is not incremented.
6. After the loop finishes, the variable `ans` has the minimum number of arrows required, which is then returned as the final answer.

The use of the greedy algorithm along with sorting simplifies the problem and allows the solution to be efficient with a time complexity of O(n log n) due to the sort operation (where n is the number of balloons) and a space complexity of O(1), assuming the sort is done in place on the input list.

Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose we have the following set of balloons with their `x_start` and `x_end` values represented as intervals:

```
Balloons: [[1,6], [2,8], [7,12], [10,16]]
```

According to the approach:

1. First, we sort the balloons by their ending points (`x_end`):

   ```
   Sorted Balloons: [[1,6], [2,8], [7,12], [10,16]]
   ```

   Since our balloons are already sorted by their `x_end`, we don't need to change the order.
2. We initialize `ans` to `0` since we haven't used any arrows yet, and `last` to negative infinity to signify that we have not shot any arrows.
3. We begin iterating over the balloons list:

   a. For the first balloon `[1,6]`, `x_start` is greater than `last` (`-inf` in this case), so we need a new arrow. We increment `ans` to `1` and update `last` to `6`.

   b. The next balloon `[2,8]` has `x_start <= last` (since `2 <= 6`), so it overlaps with the range of the last arrow. Therefore, we do not increment `ans`, and `last` remains `6`.

   c. Moving on to the third balloon `[7,12]`, `x_start` is greater than `last` (`7 > 6`), indicating no overlap with the last arrow's range. We increment `ans` to `2` and update `last` to `12`.

   d. Finally, for the last balloon `[10,16]`, since `x_start <= last` (as `10 <= 12`), it can be popped by the previous arrow, so we keep `ans` as it is.
4. After checking all balloons, we have used `2` arrows as indicated by `ans`, which is the minimum number of arrows required to pop all balloons.

By following this greedy strategy, we never miss the opportunity to pop overlapping balloons with a single arrow, ensuring an optimal solution.

Python Solution

```python
class Solution:
    def findMinArrowShots(self, points: List[List[int]]) -> int:
        # Initialize counter for arrows and set the last arrow position to negative infinity
        num_arrows, last_arrow_pos = 0, float('-inf')

        # Sort the balloon points by their end positions
        sorted_points = sorted(points, key=lambda x: x[1])

        # Loop through the sorted balloon points
        for start, end in sorted_points:
            # If the start of the current balloon is greater than the position
            # of the last arrow, we need a new arrow
            if start > last_arrow_pos:
                # Increment the number of arrows needed
                num_arrows += 1
                # Update the position for the last arrow
                last_arrow_pos = end

        # Return the minimum number of arrows required
        return num_arrows

```

Java Solution

```java
class Solution {
    public int findMinArrowShots(int[][] points) {
        // Sort the "points" array based on the end point of each interval.
        Arrays.sort(points, Comparator.comparingInt(interval -> interval[1]));

        // Initialize the counter for the minimum number of arrows.
        int arrowCount = 0;

        // Use a "lastArrowPosition" variable to track the position of the last arrow.
        // Initialize to a very small value to ensure it is less than the start of any interval.
        long lastArrowPosition = Long.MIN_VALUE;

        // Iterate through each interval in the sorted array.
        for (int[] interval : points) {
            int start = interval[0]; // Start of the current interval
            int end = interval[1];   // End of the current interval

            // If the start of the current interval is greater than the "lastArrowPosition",
            // it means a new arrow is needed for this interval.
            if (start > lastArrowPosition) {
                arrowCount++;  // Increment the number of arrows needed.
                lastArrowPosition = end;  // Update the position of the last arrow.
            }
        }

        // Return the minimum number of arrows required to burst all balloons.
        return arrowCount;
    }
}

```

C++ Solution

```cpp
#include <vector>    // Include the vector header for using the vector container
#include <algorithm> // Include the algorithm header for using the sort function

// Definition for the class Solution where our method will reside
class Solution {
public:
    // Method to find the minimum number of arrows needed to burst all balloons
    int findMinArrowShots(std::vector<std::vector<int>>& points) {
        // Sort the input vector based on the ending coordinate of the balloons
        std::sort(points.begin(), points.end(), [](const std::vector<int>& point1, const std::vector<int>& point2) {
            return point1[1] < point2[1];
        });

        int arrowCount = 0;                 // Initialize the count of arrows to zero
        long long lastBurstPosition = -(1LL << 60); // Use a very small value to initialize the position of the last burst

        // Iterate over all balloons
        for (const auto& point : points) {
            int start = point[0], end = point[1]; // Extract start and end points of the balloon

            // If the start point of the current balloon is greater than the last burst position
            // it means a new arrow is needed
            if (start > lastBurstPosition) {
                ++arrowCount;                 // Increment the arrow count
                lastBurstPosition = end;      // Update the last burst position with the end of the current balloon
            }
        }

        return arrowCount; // Return the total number of arrows needed
    }
};

```

Typescript Solution

```typescript
// Function to determine the minimum number of arrows
// required to burst all balloons
function findMinArrowShots(points: number[][]): number {
    // Sort the points by the end coordinates
    points.sort((a, b) => a[1] - b[1]);

    // Initialize the counter for the minimum number of arrows
    let arrowsNeeded = 0;

    // Initialize the position where the last arrow was shot
    // It starts at the smallest possible value so the first balloon gets shot
    let lastArrowPosition = -Infinity;

    // Iterate over all points (balloons)
    for (const [start, end] of points) {
        // If the current balloon's start position is
        // greater than the position where the last arrow was shot,
        // it means a new arrow is needed for this balloon
        if (lastArrowPosition < start) {
            // Increment the arrow counter
            arrowsNeeded++;
            // Update the last arrow's position to the current balloon's end position
            // as we can shoot it at the end and still burst it
            lastArrowPosition = end;
        }
    }

    // Return the total number of arrows needed
    return arrowsNeeded;
}

```

Time and Space Complexity

The time complexity of the given code can be broken down into two major parts: the sorting of the input list and the iteration over the sorted list.

1. **Sorting:**
   - The `sorted()` function has a time complexity of `O(n log n)`, where `n` is the number of intervals in `points`.
   - This is the dominant factor in the overall time complexity as it grows faster than linear with the size of the input.
2. **Iteration:**
   - After sorting, the code iterates over the sorted list only once.
   - The iteration has a linear time complexity of `O(n)`, where `n` is the number of intervals.

Combining these two operations, the overall time complexity of the algorithm is `O(n log n)` due to the sorting step which dominates the iteration step.

The space complexity is determined by the additional space used by the algorithm apart from the input.

1. **Additional Space:**
   - The `sorted()` function returns a new list that is a sorted version of `points`, which consumes `O(n)` space.
   - The variables `ans` and `last` use a constant amount of space `O(1)`.

The overall space complexity of the algorithm is `O(n)` to account for the space required by the sorted list.