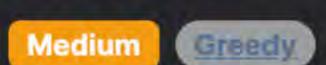
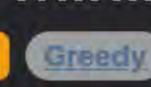
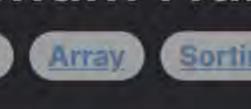
## 452. Minimum Number of Arrows to Burst Balloons





Problem Description



overlap with each other, a single arrow can burst all of them.



Leetcode Link

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.

In this LeetCode problem, we are presented with a scenario involving a number of spherical balloons that are taped to a wall,

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.

We are tasked with finding the minimum number of arrows that need to be shot vertically upwards along the Y-axis from different

Intuition

## To solve this problem, we need to look for overlapping intervals among the balloons' diameters. If multiple balloons' diameters

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.
- 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.

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

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

before moving on to the next arrow. The steps of the algorithm are implemented as follows in the given Python code:

## 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

set last to this balloon's end coordinate:

global optimum.

1 last = b

ascending order based on their ending x-coordinates (x\_end). 1 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.

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

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

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 and has the minimum number of arrows required, which is then returned as the final answer.

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.

The use of the greedy algorithm along with sorting simplifies the problem and allows the solution to be efficient with a time

x\_end values represented as intervals: 1 Balloons: [[1,6], [2,8], [7,12], [10,16]] According to the approach:

Let's illustrate the solution approach with a small example. Suppose we have the following set of balloons with their x\_start and

1. First, we sort the balloons by their ending points (x\_end):

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

Example Walkthrough

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

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.

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

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.

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

Python Solution class Solution:

18 # Return the minimum number of arrows required 19 20 return num\_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

not increment ans, and last remains 6.

update last to 6.

all balloons.

solution.

10

11

13

14

15

16

17

9

10

11

12

13

14

16

17

18

19

20

21

22 23

24

25

26

27

28

29 }

arrows.

- 4. After checking all balloons, we have used 2 arrows as indicated by ans, which is the minimum number of arrows required to pop
  - 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
- 21 Java Solution

# If the start of the current balloon is greater than the position

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

# of the last arrow, we need a new arrow

# Increment the number of arrows needed

# Update the position for the last arrow

// Sort the "points" array based on the end point of each interval.

int start = interval[0]; // Start of the current interval

arrowCount++; // Increment the number of arrows needed.

// Return the minimum number of arrows required to burst all balloons.

#include <algorithm> // Include the algorithm header for using the sort function

int end = interval[1]; // End of the current interval

// it means a new arrow is needed for this interval.

// Initialize the counter for the minimum number of arrows.

// Iterate through each interval in the sorted array.

Arrays.sort(points, Comparator.comparingInt(interval -> interval[1]));

// 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.

// If the start of the current interval is greater than the "lastArrowPosition",

lastArrowPosition = end; // Update the position of the last arrow.

// Include the vector header for using the vector container

# Loop through the sorted balloon points

for start, end in sorted\_points:

if start > last\_arrow\_pos:

last\_arrow\_pos = end

public int findMinArrowShots(int[][] points) {

long lastArrowPosition = Long.MIN\_VALUE;

if (start > lastArrowPosition) {

for (int[] interval : points) {

int arrowCount = 0;

return arrowCount;

num\_arrows += 1

# 30 C++ Solution

1 #include <vector>

class Solution {

```
// 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) {
10
               return point1[1] < point2[1];</pre>
11
           });
12
13
                                              // Initialize the count of arrows to zero
14
           int arrowCount = 0;
15
            long long lastBurstPosition = -(1LL \ll 60); // Use a very small value to initialize the position of the last burst
16
           // Iterate over all balloons
17
           for (const auto& point : points) {
18
                int start = point[0], end = point[1]; // Extract start and end points of the balloon
19
20
21
               // If the start point of the current balloon is greater than the last burst position
22
               // it means a new arrow is needed
23
               if (start > lastBurstPosition) {
24
                   ++arrowCount;
                                                  // Increment the arrow count
                                                   // Update the last burst position with the end of the current balloon
25
                   lastBurstPosition = end;
26
27
28
29
           return arrowCount; // Return the total number of arrows needed
30
31 };
32
Typescript Solution
```

### 24 lastArrowPosition = end; 25 26 27 // Return the total number of arrows needed 28

return arrowsNeeded;

// Function to determine the minimum number of arrows

// Sort the points by the end coordinates

points.sort((a, b) => a[1] - b[1]);

let lastArrowPosition = -Infinity;

// Iterate over all points (balloons)

if (lastArrowPosition < start) {</pre>

// Increment the arrow counter

for (const [start, end] of points) {

arrowsNeeded++;

function findMinArrowShots(points: number[][]): number {

// Initialize the counter for the minimum number of arrows

// Initialize the position where the last arrow was shot

// If the current balloon's start position is

// it means a new arrow is needed for this balloon

// It starts at the smallest possible value so the first balloon gets shot

// Update the last arrow's position to the current balloon's end position

// greater than the position where the last arrow was shot,

// as we can shoot it at the end and still burst it

2 // required to burst all balloons

let arrowsNeeded = 0;

9

10

11

12

13

14

16

17

18

19

20

23

29

31

30 }

the sorted list.

1. Sorting:

1. Additional Space:

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

- 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.

The sorted() function has a time complexity of O(n log n), where n is the number of intervals in points.

- Combining these two operations, the overall time complexity of the algorithm is 0(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.

• 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 0(1). The overall space complexity of the algorithm is O(n) to account for the space required by the sorted list.