**Documentation for Minimum Number of Arrows to Burst Balloons**

**Table of Contents**



## 1. Problem Statement

You are given a set of spherical balloons, each represented as a 2D integer array points, where:

- points[i] = [$x_{start}$, $x_{end}$] denotes a balloon with a horizontal diameter between $x_{start}$ and $x_{end}$.

You can shoot arrows vertically upwards from any x position.

- A balloon [$x_{start}$, $x_{end}$] is burst if $x_{start} \leq x \leq x_{end}$.
- Arrows travel infinitely upward, bursting all balloons in their path.

Goal: Find the minimum number of arrows required to burst all balloons.

**Example 1**

Input: points = [[10,16],[2,8],[1,6],[7,12]]
Output:2

Explanation:

- Arrow 1 at x=6 bursts [1,6] and [2,8].
- Arrow 2 at x=11 bursts [7,12] and [10,16].

**Example 2**

Input: points = [[1,2],[3,4],[5,6],[7,8]]
Output: 4

Explanation:

- Each balloon needs a separate arrow.

**Example 3**

Input:points = [[1,2],[2,3],[3,4],[4,5]]
Output:2

Explanation:

- Arrow 1 at x=2 bursts [1,2] and [2,3].
- Arrow 2 at x=4 bursts [3,4] and [4,5].

## 2. Intuition

To minimize arrows, we should aim to burst as many balloons as possible with each shot.

- If two balloons overlap, one arrow can burst both.
- If they do not overlap, we need separate arrows.

Greedy Choice

- Always aim to shoot arrows at the earliest possible $x_{end}$ of an overlapping group.
- This ensures maximum coverage with minimal arrows.

## 3. Key Observations

- Sorting the balloons by xend helps us make optimal greedy choices.
- We only need a new arrow when a balloon's xstart is beyond the current xend.

**Example:**

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

Arrow at x=6 → covers [1,6] & [2,8]

Arrow at x=12 → covers [7,12] & [10,16]

## 4. Approach

Step 1: Sort the Balloons

- Sort points by their $x_{end}$ (smallest first).
- Sorting ensures we always deal with the smallest $x_{end}$ first, optimizing for minimal arrows.

Step 2: Traverse and Count Arrows

- Initialize arrows = 1 (at least one arrow is needed).
- Start with prev_end = points[0][1] (end of first balloon).
- For each balloon (start, end):
  - If start > prev_end, it means we must shoot a new arrow.
  - Otherwise, the balloon is already covered by the previous arrow.

Step 3: Return the Arrow Count

- The count of arrows used is the answer.

## 5. Edge Cases

| Edge Case | Explanation |
|---|---|
| Single Balloon | Needs only 1 arrow. |
| No Overlapping Balloons | Each balloon requires a separate arrow. |
| All Overlapping Balloons | Can be burst with a single arrow. |
| Large Input Size ($10^5$ balloons) | Sorting takes O(n log n), which is efficient. |
| Negative x-coordinates | The algorithm should handle negative values correctly. |

## 6. Complexity Analysis

Time Complexity

- Sorting O(n log n)
- One-pass traversal O(n)
- Total: O(n log n)

Space Complexity

- O(1) (in-place sorting and constant extra variables)

## 7. Alternative Approaches

i. Dynamic Programming (DP)
   a. We could use DP to find optimal arrow positions.
   b. However, DP has $O(n^2)$ complexity, making it slower than the greedy approach.

ii. Interval Merging Approach
   a. Merge overlapping intervals and count the groups.
   b. Works similarly but sorting by $x_{end}$ is more intuitive.

## 8. Test Cases

```
solution = Solution()
# Test Case 1
assert solution.findMinArrowShots([[10,16],[2,8],[1,6],[7,12]]) == 2
# Test Case 2
assert solution.findMinArrowShots([[1,2],[3,4],[5,6],[7,8]]) == 4
# Test Case 3
assert solution.findMinArrowShots([[1,2],[2,3],[3,4],[4,5]]) == 2
# Test Case 4 - Large input
assert solution.findMinArrowShots([[1,10**9],[2,10**9],[3,10**9],[4,10**9]]) == 1
print("All test cases passed!")
```

## 9. Final Thoughts

- The Greedy Algorithm is the optimal solution.
- Sorting by $x_{end}$ ensures minimum arrows are used.
- Time Complexity is efficient at O(n log n), making it suitable for large inputs.