<https://leetcode.com/problems/minimum-number-of-arrows-to-burst-balloons/description/>

There are some spherical balloons taped onto a flat wall that represents the XY-plane. The balloons are represented as a 2D integer array points where points[i] = [xstart, xend] denotes a balloon whose **horizontal diameter** stretches between xstart and xend. You do not know the exact y-coordinates of the balloons.

Arrows can be shot up **directly vertically** (in the positive y-direction) from different points along the x-axis. A balloon with xstart and xend is **burst** by an arrow shot at x if xstart <= x <= xend. There is **no limit** to the number of arrows that can be shot. A shot arrow keeps traveling up infinitely, bursting any balloons in its path.

Given the array points, return *the* ***minimum*** *number of arrows that must be shot to burst all balloons*.

**Example 1:**

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

Output: 2

Explanation: The balloons can be burst by 2 arrows:

- Shoot an arrow at x = 6, bursting the balloons [2,8] and [1,6].

- Shoot an arrow at x = 11, bursting the balloons [10,16] and [7,12].

              ^

        ^   10|    16

        |    -|-----

    2   | 8   |

    ----|--   |

   1    6     |

   -----|     |

        |7    12

        |-----|

        |     |

   arrow=6   arrow=11

**Example 2:**

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

Output: 4

Explanation: One arrow needs to be shot for each balloon for a total of 4 arrows.

**Example 3:**

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

Output: 2

Explanation: The balloons can be burst by 2 arrows:

- Shoot an arrow at x = 2, bursting the balloons [1,2] and [2,3].

- Shoot an arrow at x = 4, bursting the balloons [3,4] and [4,5].

**Constraints:**

* 1 <= points.length <= 10^5
* points[i].length == 2
* -2^31 <= xstart < xend <= 2^31 - 1

**Attempt 1: 2023-12-04**

**Solution 1: Interval merge strategy (60 min)**

class Solution {

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

// We need to sort with 'end' first to start with minimum 'end' interval

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

// Test out by: [[-2147483646,-2147483645],[2147483646,2147483647]]

// a[1] - b[1] overflow integer range, instead substraction we have

// to use 'Integer.compare()'

// Refer to

// https://monicagranbois.com/blog/java/comparing-integers-using-integercompare-vs-subtraction/

Arrays.sort(points, (a, b) -> Integer.compare(a[1], b[1]));

// Setup first interval 'end' as initial previous max 'end'

int prev\_max\_end = points[0][1];

// For first interval we need at least one arrow to burst

int arrow\_count = 1;

for(int i = 1; i < points.length; i++) {

// If new coming [start, end] interval 'start' position

// has intersection with previous max 'end' position,

// then the ith interval can share the same arrow with

// previous interval who own previous max 'end'

// And '=' coming from example below, we can see '2' is shared

// by two intervals as both 'end' and 'start', and able to shoot

// by same arrow, same for '4'

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

// Output: 2

// Explanation: The balloons can be burst by 2 arrows:

// - Shoot an arrow at x = 2, bursting the balloons [1,2] and [2,3].

// - Shoot an arrow at x = 4, bursting the balloons [3,4] and [4,5].

if(points[i][0] <= prev\_max\_end) {

continue;

}

// Otherwise we need to record this new interval's new end,

// and also need a new arrow to burst the new interval

arrow\_count++;

prev\_max\_end = points[i][1];

}

return arrow\_count;

}

}

Time Complexity: O(N\*logN)

Space Complexity: O(N)

**Refer to**

<https://leetcode.com/problems/minimum-number-of-arrows-to-burst-balloons/solutions/93703/share-my-explained-greedy-solution/>

No offense but the currently highest voted java solution is not ideal, the sorting can be adjusted so that there's no need to check again in the for loop.

**Idea:**

We know that eventually we have to shoot down every balloon, so for each ballon there must be an arrow whose position is between **balloon[0]**and **balloon[1]**inclusively. Given that, we can sort the array of balloons by their **ending position**. Then we make sure that **while we take care of each balloon in order, we can shoot as many following balloons as possible.**

So what position should we pick each time? We should shoot as to the right as possible, because since balloons are sorted, this gives you the best chance to take down more balloons. Therefore the position should always be **balloon[i][1]** for the *i*th balloon.

This is exactly what I do in the for loop: check how many balloons I can shoot down with one shot aiming at the **ending position of the current balloon**. Then I skip all these balloons and start again from the next one (or the leftmost remaining one) that needs another arrow.

**Example:**

balloons = [[7,10], [1,5], [3,6], [2,4], [1,4]]

After sorting, it becomes:

balloons = [[2,4], [1,4], [1,5], [3,6], [7,10]]

So first of all, we shoot at position **4**, we go through the array and see that all first 4 balloons can be taken care of by this single shot. Then we need another shot for one last balloon. So the result should be 2.

**Code:**

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

if (points.length == 0) {

return 0;

}

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

int arrowPos = points[0][1];

int arrowCnt = 1;

for (int i = 1; i < points.length; i++) {

if (arrowPos >= points[i][0]) {

continue;

}

arrowCnt++;

arrowPos = points[i][1];

}

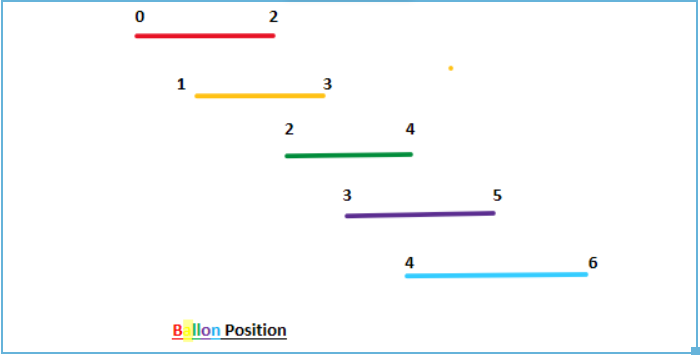
return arrowCnt;

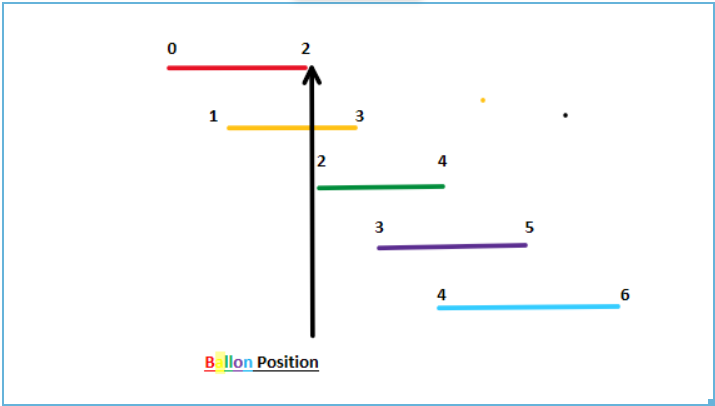
}

**Refer to**

<https://leetcode.com/discuss/study-guide/2166045/line-sweep-algorithms>

In this kind of problem, we don't use marking on axis, instead axis is already given, we sort it.





Now if we strike first balloon , we can strike at the end-most point i.e. **2**

We save this as **prev = 2**

Any balloon starting before we don't need extra arrow since this arrow is enough to burst the balloon.

The moment start point > prev , we know we need a new balloon again we save the end-most point in the prev and repeat the process.

Many problem can be solved using this technique.

class Solution {

public:

int findMinArrowShots(vector<vector<int>>& points) {

auto cmp = [&](const vector<int>& a, const vector<int>& b){

return a[1] < b[1];

};

sort(points.begin(), points.end(), cmp);

int prev\_end = points[0][1];

int ans = 1;

for(int i =1; i < points.size(); ++i){

if(points[i][0] > prev\_end){

++ans;

prev\_end = points[i][1];

}

}

return ans;

}

};

**Refer to**

<https://algo.monster/liteproblems/452>

## 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:

* 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.
* 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).
* 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.
* 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.
* 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:

* 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).
* A variable ans is initialized to 0 to keep track of the total number of arrows used.
* 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.
* 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:
* This ensures that any subsequent balloon that starts before b (the current last) can be burst by the current arrow.
* 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.
* 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:

* First, we sort the balloons by their ending points (x\_end):
* Since our balloons are already sorted by their x\_end, we don't need to change the order.
* 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.
* 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.

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.

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;

}

}

## 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.

**Sorting:**

The sorted() function has a time complexity of O(nlogn), 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.

**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(nlogn) 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.

**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.

# **Comparing integers using [Integer.compare](http://Integer.compare) vs subtraction**

**Refer to**

<https://monicagranbois.com/blog/java/comparing-integers-using-integercompare-vs-subtraction/>

Use [Integer.compare](http://Integer.compare) to avoid overflow that can occur when subtracting two integers.

I took [Oracle’s Java 8 MOOC: Lamdbas And Streams](https://apexapps.oracle.com/pls/apex/f?p=44785:145:0::::P145_EVENT_ID,P145_PREV_PAGE:4887,143) class. I had a question about using [Integer.compare](http://Integer.compare) that arose from one of the homework questions. I am recording the question and answer here so I can find it easily in the future, instead of having it buried in a thread about a homework assignment using the Streams API. The original post is on the [Oracle community forums](https://community.oracle.com/message/13227819" \l "13227819).

My question was:

My answers were pretty similar to what has already been posted. However, for the sort in question 7 I did:

sorted((word1, word2) -> Integer.compare(word1.length(), word2.length()))

whereas most implementations in this thread have done something like:

sorted((word1, word2) -> word1.length() - word2.length())

Is there any advantage to using subtraction vs using the [Integer.compare](http://Integer.compare) method? Or is just different ways of doing the same thing?

The reply from [Stuart Marks-Oracle](https://community.oracle.com/people/Stuart Marks-Oracle?customTheme=otn) was:

For this particular problem, there is essentially no difference. However, there are cases when comparing integers, where writing a comparator that simply subtracts the two values can fail to produce the right answer. Consider the following:

List<Integer> list = Arrays.asList(Integer.MAX\_VALUE, -1, 0, 1, Integer.MIN\_VALUE);

list.sort((i1, i2) -> i1 - i2);

System.out.println(list);

// output

[-1, 0, 1, 2147483647, -2147483648]

This is clearly wrong. Can you see why? Overflow, that’s why! When comparing numbers of extreme magnitude, such as MAX\_VALUE and -1, these are subtracted, giving a result with the wrong sign. That’s why the sorting fails in the case. The correct way to compare integers is, as you note, using [Integer.compare](http://Integer.compare)():

List<Integer> list = Arrays.asList(Integer.MAX\_VALUE, -1, 0, 1, Integer.MIN\_VALUE);

list.sort(Integer::compare);

System.out.println(list);

// output

[-2147483648, -1, 0, 1, 2147483647]

There’s nothing magical about [Integer.compare](http://Integer.compare)(). It’s basically a conditional expression that uses comparison operations instead of subtraction:

return (x < y) ? -1 : ((x == y) ? 0 : 1);

In the exercise, string lengths are always non-negative, so subtracting them cannot result in overflow. However, I always recommend using [Integer.compare](http://Integer.compare)(), so that you don’t have to prove that overflow cannot occur.

**Solution 2: Priority Queue (10 min)**

**实际就是把Solution 1中的arrow\_count用maxPQ替代了，其他的完全一致**

class Solution {

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

Arrays.sort(points, (a, b) -> Integer.compare(a[1], b[1]));

PriorityQueue<Integer> maxPQ = new PriorityQueue<>((a, b) -> Integer.compare(b, a));

maxPQ.offer(points[0][1]);

for(int i = 1; i < points.length; i++) {

if(points[i][0] <= maxPQ.peek()) {

continue;

}

maxPQ.offer(points[i][1]);

}

return maxPQ.size();

}

}

Time Complexity: O(N\*logN)

Space Complexity: O(N)

**Refer to**

<https://leetcode.com/problems/minimum-number-of-arrows-to-burst-balloons/solutions/889797/same-as-meeting-rooms-ii-greedy-c/>

Check if the current balloon can be merged with the balloon having the highest end co-ordinate.

int findMinArrowShots(vector<vector<int>>& points) {

sort(points.begin(),points.end());

priority\_queue<int> pq;

for(int i=0;i<points.size();i++)

{

if(pq.empty() || pq.top()<points[i][0])

pq.push(points[i][1]);

else

{

int top=pq.top();

pq.pop();

pq.push(min(top,points[i][1]));

}

}

return pq.size();

}