<https://leetcode.com/problems/remove-covered-intervals/description/>

Given an array intervals where intervals[i] = [li, ri] represent the interval [li, ri), remove all intervals that are covered by another interval in the list.

The interval [a, b) is covered by the interval [c, d) if and only if c <= a and b <= d.

Return *the number of remaining intervals*.

**Example 1:**

Input: intervals = [[1,4],[3,6],[2,8]]

Output: 2

Explanation: Interval [3,6] is covered by [2,8], therefore it is removed.

**Example 2:**

Input: intervals = [[1,4],[2,3]]

Output: 1

**Constraints:**

* 1 <= intervals.length <= 1000
* intervals[i].length == 2
* 0 <= li < ri <= 10^5
* All the given intervals are **unique**.

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

**Solution 1: Sorting + Interval intersection check (10 min)**

**Similar strategy with L452.Minimum Number of Arrows to Burst Balloons**

class Solution {

public int removeCoveredIntervals(int[][] intervals) {

Arrays.sort(intervals, (a, b) -> a[0] == b[0] ? b[1] - a[1] : a[0] - b[0]);

int[] prev = intervals[0];

int count = 1;

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

if(intervals[i][1] > prev[1]) {

count++;

prev = intervals[i];

}

}

return count;

}

}

Time Complexity: O(N\*logN)

Space Complexity: O(1)

**Refer to**

<https://algo.monster/liteproblems/1288>

## Problem Description

The problem presents a scenario in which you're given a list of intervals, with each interval represented as a pair of integers [start, end). The notation [start, end) signifies that the interval includes start and goes up to but does not include end. The goal is to remove intervals that are "covered" by other intervals. An interval [a, b) is considered covered by another interval [c, d) if both c <= a and b <= d. In simple terms, if one interval lies completely within the boundary of another, the inner interval is said to be covered by the outer interval. Your task is to find out how many intervals remain after removing all the covered intervals.

## Intuition

The intuitive approach to solving this problem involves checking each interval against all others to see if it is covered by any. However, this would be inefficient, particularly for large lists of intervals. Thus, the key to an efficient solution is to sort the intervals in a way that makes it easier to identify and remove the covered intervals.

By sorting the intervals first by their start times and then by their end times in descending order, we line up intervals in such a way that once we find an interval not covered by the previous one, none of the following intervals will cover it either. This sorting strategy allows us to only compare each interval with the last one that wasn't removed.

With the sorted list, we iterate through the intervals, where the main idea is to compare the current interval's end with the previous interval's end. If the current end is greater, this interval isn't covered by the previous interval. We count this interval, and it becomes the new previous interval for subsequent comparisons. If the current end is not greater, it means this interval is covered by the previous interval, and we do not count it and proceed to the next interval. After checking all intervals, the count cnt gives the number of intervals that remain.

## Solution Approach

The solution makes use of a greedy algorithm approach, where we aim to remove the minimal number of intervals by keeping the ones that aren't covered by any other. Let's walk through the solution approach along with the algorithms, data structures, or patterns used:

* **Sorting**: We begin by sorting the intervals list using a custom sort key. This key sorts the intervals primarily by their start times in ascending order. If two intervals have the same start time, we sort them by their end times in descending order. This ensures that we have the longer intervals earlier if the start times are the same, making it easier to identify covered intervals.
* **Initializing Counters**: After sorting, we initialize our count cnt to 1. This is because we consider the first interval as not being covered by any previous one, as there are no previous intervals yet. We also initialize pre to keep track of the last interval that was not covered by the one before it.
* **Iterating Through Intervals**: We loop through the sorted intervals starting from the second one. At each iteration, we compare the current interval's end time (e[1]) with the end time of pre (pre[1]):

1. If pre[1] (previous interval's end time) is less than e[1] (current interval's end time), it indicates that the current interval is not completely covered by pre. Hence, we increment our count cnt and update pre to the current interval.
2. If pre[1] is greater than or equal to e[1], the current interval is covered by pre, and we don't increment cnt.

* **Return the Result**: After iterating through all intervals, cnt holds the count of intervals that remain after removing all that are covered by another interval.

It's important to understand that by sorting the intervals first by starting time and then by the ending time (in opposite order), we limit the comparison to only the previous interval and the current one in the loop. This greatly reduces the number of comparisons needed, resulting in a more efficient algorithm.

Below is the reference code implementation based on this approach:

class Solution:

def removeCoveredIntervals(self, intervals: List[List[int]]) -> int:

# Sort intervals by start time, and then by end time in descending order

intervals.sort(key=lambda x: (x[0], -x[1]))

cnt, pre = 1, intervals[0] # Initialize the counter and the previous interval tracker

# Iterate through each interval in the sorted list

for e in intervals[1:]:

# If the current interval's end time is greater than the previous', it's not covered

if pre[1] < e[1]:

cnt += 1 # Increment the counter, as this interval isn't covered

pre = e # Update the 'pre' interval tracker to the current interval

return cnt # Return the final count of non-covered intervals

With the combination of a clever sorting strategy and a single loop, this code neatly solves the problem in an efficient manner.

### Example Walkthrough

Let's go through a small example to illustrate the solution approach described.

Suppose we have a list of intervals: [[1,4), [2,3), [3,6)].

Following our solution approach:

**Sorting**: Applying our custom sort, the list is sorted by start times in ascending order, and for those with the same start time, by end times in descending order. But since all our example intervals have different start times, we only need to sort by the first element:

Sorted list: [[1,4), [2,3), [3,6)]

**Initializing Counters**: We initialize our count cnt to 1, assuming the first interval [1,4) is not covered. We also initialize pre with this interval.

**Iterating Through Intervals**:

We first compare [1,4) with [2,3). Here, pre[1] is 4 and e[1] is 3. Since pre[1] >= e[1], we find that [2,3) is covered by [1,4) and thus, we do not increment cnt.

Next, we compare [1,4) with [3,6). Here, pre[1] is 4 and e[1] is 6. Since pre[1] < e[1], we find that [3,6) is not covered by [1,4). So, we increment cnt to 2 and update pre to [3,6).

**Return the Result**: Having finished iterating through the intervals, we find that the count cnt is 2, which means there are two intervals that remain after removing all that are covered: [[1,4), [3,6)].

This example walk-through demonstrates the efficiency of the algorithm by effectively reducing the number of necessary comparisons and clearly exhibiting how the sorting step greatly simplifies the process of identifying covered intervals.

class Solution {

public int removeCoveredIntervals(int[][] intervals) {

// Sort the intervals. First by the start in ascending order.

// If the starts are equal, sort by the end in descending order.

Arrays.sort(intervals, (a, b) -> {

if (a[0] == b[0]) {

return b[1] - a[1];

} else {

return a[0] - b[0];

}

});

// Initialize the previous interval as the first interval

int[] previousInterval = intervals[0];

// Count the first interval

int count = 1;

// Iterate through all intervals starting from the second one

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

// If the current interval's end is greater than the previous interval's end,

// it means the current interval is not covered by the previous interval.

if (previousInterval[1] < intervals[i][1]) {

// Increment the count of non-covered intervals.

++count;

// Update the previous interval to the current interval.

previousInterval = intervals[i];

}

// If the current interval's end is not greater than the previous interval's end,

// it's covered by the previous interval and we do nothing.

}

// Return the number of non-covered intervals

return count;

}

}

## Time and Space Complexity

### Time Complexity

The time complexity of the code is dominated by two operations: the sorting of the intervals and the single pass through the sorted list.

* The sort() function in Python uses the Timsort algorithm, which has a time complexity of O(nlogn) where n is the number of intervals.
* After sorting, the code performs a single pass through the list to count the number of non-covered intervals, which has a time complexity of O(n).

Combining these two steps, the overall time complexity is O(nlogn + n). Simplified, it remains O(nlogn) because the nlogn term is dominant.

### Space Complexity

The space complexity refers to the amount of extra space or temporary storage that an algorithm requires.

* Sorting the list is done in-place, which means it doesn't require additional space proportional to the input size. Therefore, the space complexity due to sorting is constant, O(1).
* Aside from the sorted list, the algorithm only uses a fixed number of variables (cnt, pre, and e) which also take up constant space.

Hence, the overall space complexity is O(1), because no additional space that scales with the size of the input is used.