<https://leetcode.com/problems/non-overlapping-intervals/>

Given an array of intervals intervals where intervals[i] = [starti, endi], return *the minimum number of intervals you need to remove to make the rest of the intervals non-overlapping*.

**Example 1:**

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

Output: 1

Explanation: [1,3] can be removed and the rest of the intervals are non-overlapping.

**Example 2:**

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

Output: 2

Explanation: You need to remove two [1,2] to make the rest of the intervals non-overlapping.

**Example 3:**

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

Output: 0

Explanation: You don't need to remove any of the intervals since they're already non-overlapping.

**Constraints:**

* 1 <= intervals.length <= 105
* intervals[i].length == 2
* -5 \* 104 <= starti < endi <= 5 \* 104

**Attempt 1: 2023-03-03**

**Solution 1: Sorting with interval 'start' and greedy remove overlapped interval on smaller 'end' (30 min)**

**Style 1: Use 'int prevEnd' variable to store the previous interval 'end'**

class Solution {

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

// Sort by the 'start' element in the intervals

Arrays.sort(intervals, (a, b) -> a[0] - b[0]);

int removeCount = 0;

int prevEnd = intervals[0][1];

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

// Overlapping

// Strict '>' not '>='

// e.g

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

// Output: 1

// Explanation: [1,3] can be removed and the rest of the intervals are non-overlapping, [1,2] and [2,3] not recognized as overlapping

if(prevEnd > intervals[i][0]) {

// When we encounter an overlap, we increase the count,

// and decide to remove the interval with the larger end date.

// In another word, if we choose the interval that ends earlier,

// then there is more space for other intervals

// In code, this translates to setting the "prevEnd" (the end

// time of the most recent non-deleted interval) to

// Math.min(current interval that we have encountered the

// conflict with, current prevEnd)

prevEnd = Math.min(prevEnd, intervals[i][1]);

removeCount++;

// No overlapping

} else {

prevEnd = intervals[i][1];

}

}

return removeCount;

}

}

Time Complexity:O(nlogn), sorting take nlogn time

Space Complexity:O(1)

**Style 2: Use 'int[] prev' to store previous interval**

class Solution {

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

// Sort by the 'start' element in the intervals

Arrays.sort(intervals, (a, b) -> a[0] - b[0]);

int removeCount = 0;

int[] prev = intervals[0];

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

// Overlapping

// Strict '>' not '>='

// e.g

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

// Output: 1

// Explanation: [1,3] can be removed and the rest of the intervals are non-overlapping, [1,2] and [2,3] not recognized as overlapping

if(prev[1] > intervals[i][0]) {

// When we encounter an overlap, we increase the count,

// and decide to remove the interval with the larger end date.

// In another word, if we choose the interval that ends earlier,

// then there is more space for other intervals

// In code, this translates to setting the "prevEnd" (the end

// time of the most recent non-deleted interval) to

// Math.min(current interval that we have encountered the

// conflict with, current prevEnd)

prev[1] = Math.min(prev[1], intervals[i][1]);

removeCount++;

// No overlapping

} else {

prev = intervals[i];

}

}

return removeCount;

}

}

Time Complexity:O(nlogn), sorting take nlogn time

Space Complexity:O(1)

**Refer to**

<https://leetcode.com/problems/non-overlapping-intervals/solutions/481758/easy-to-understand-java-solution/>

**Pretty straightforward. Logic is that we sort by the first element in the intervals, then when we encounter an overlap, we increase the count, and decide to remove the interval with the larger end date. In code, this translates to setting the "prevEnd" (the end time of the most recent non-deleted interval) to Math.min(current interval that we have encountered the conflict with, current prevEnd).**

class Solution {

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

if (intervals == null || intervals.length == 0) return 0;

Arrays.sort(intervals, (a, b) -> Integer.compare(a[0], b[0]));

int prevEnd = intervals[0][1];

int count = 0;

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

if (prevEnd > intervals[i][0]) {

count++;

prevEnd = Math.min(intervals[i][1], prevEnd);

} else {

prevEnd = intervals[i][1];

}

}

return count;

}

}

**Solution 2: Sorting with interval 'end' and calculate non-overlapping intervals first (30 min)**

class Solution {

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

// Sort by the 'end' element in the intervals

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

int noOverlappingCount = 1;

int[] prev = intervals[0];

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

// No overlapping

if(prev[1] <= intervals[i][0]) {

prev = intervals[i];

noOverlappingCount++;

}

}

return intervals.length - noOverlappingCount;

}

}

Time Complexity:O(nlogn), sorting take nlogn time

Space Complexity:O(1)

**Refer to**

<https://leetcode.com/problems/non-overlapping-intervals/solutions/91713/java-least-is-most/comments/413432>

**Actually, the problem is the same as "Given a collection of intervals, find the maximum number of intervals that are non-overlapping." (the classic Greedy problem: [Interval Scheduling](https://en.wikipedia.org/wiki/Interval_scheduling" \l "Interval_Scheduling_Maximization)). With the solution to that problem, guess how do we get the minimum number of intervals to remove? : )**

**Sorting Interval.end in ascending order is O(nlogn), then traverse intervals array to get the maximum number of non-overlapping intervals is O(n). Total is O(nlogn).**

public static int eraseOverlapIntervals(int[][] intervals) {

if(intervals == null || intervals.length== 0) return 0;

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

int k =0, count =1, n = intervals.length;

for(int i =1; i < n ; i++){

if (intervals[i][0] >= intervals[k][1]){

k = i;

count++;

}

}

return n-count;

}

**Refer to**

<https://leetcode.com/problems/non-overlapping-intervals/solutions/91713/java-least-is-most/comments/96271>

Here is my thinking process along with reading CLRS explanation in section 16.1 "An activity-selection problem".

1.**Why {all intervals} - {max compatible intervals} = minimum deleted intervals?** Suppose interval A in the latter max compatible set B and A causes two other intervals be deleted. If we delete A instead and insert those two deleted intervals to B can obtain a larger set, then it contradicts B is the max compatible intervals.

2.**Why sort by finish time can get max compatible intervals?** Refer to CLRS Theorem 16.1. Briefly speaking, if earliest finished is not included, we can always replace the first interval in the set with it.

public int eraseOverlapIntervals(Interval[] intervals) {

Arrays.sort(intervals, Comparator.comparingInt(i -> i.end));

int max = 0, lastend = Integer.MIN\_VALUE;

for (Interval in : intervals) {

if (lastend <= in.start) {

lastend = in.end;

max++;

}

}

return intervals.length - max;

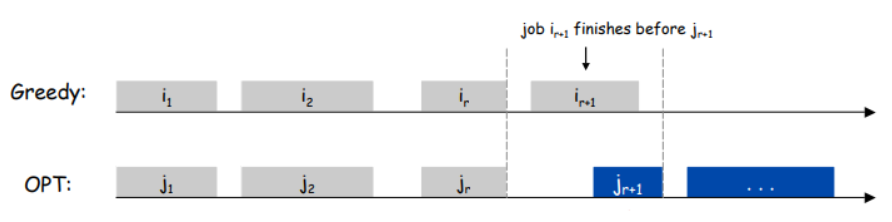
}

**Theory proof**

<https://leetcode.com/problems/non-overlapping-intervals/solutions/91713/java-least-is-most/comments/730443>

## **Why sort by finish time can get max compatible intervals?**

Let i1, i2, i3, ... ik denote the intervals selected by Greedy. Let j1, j2, j3, ... jm denote the intervals in the optimal solution. Assume the first r intervals are the same in both Greedy and Optimal, that is, i1 = j1, i2 = j2,..., ir = jr. As the graph shown below, ir+1 and jr+1 will be the first interval pair that are not the same.



Now assume greedy is not optimal, then the number of intervals selected by Greedy should less than the number of intervals in the optimal solution. But we can use ir+1 to replace jr+1 and the solution will still feasible and optimal, which is contracted with our assumption.

