<https://leetcode.com/problems/insert-interval/>

You are given an array of non-overlapping intervals intervals where intervals[i] = [starti, endi] represent the start and the end of the ith interval and intervals is sorted in ascending order by starti. You are also given an interval newInterval = [start, end] that represents the start and end of another interval.

Insert newInterval into intervals such that intervals is still sorted in ascending order by starti and intervals still does not have any overlapping intervals (merge overlapping intervals if necessary).

Return intervals *after the insertion*.

**Example 1:**

Input: intervals = [[1,3],[6,9]], newInterval = [2,5]

Output: [[1,5],[6,9]]

**Example 2:**

Input: intervals = [[1,2],[3,5],[6,7],[8,10],[12,16]], newInterval = [4,8]

Output: [[1,2],[3,10],[12,16]]

Explanation: Because the new interval [4,8] overlaps with [3,5],[6,7],[8,10].

**Constraints:**

* 0 <= intervals.length <= 104
* intervals[i].length == 2
* 0 <= starti <= endi <= 105
* intervals is sorted by starti in **ascending** order.
* newInterval.length == 2
* 0 <= start <= end <= 105

**Attempt 1: 2023-03-01**

**Solution 1: Linear scan intervals (30 min)**

class Solution {

public int[][] insert(int[][] intervals, int[] newInterval) {

List<int[]> result = new ArrayList<int[]>();

int len = intervals.length;

int i = 0;

// Add all the intervals ending before newInterval starts

// Strict '<' means when intervals[i][1] == newInterval[0]

// we still go with merge logic

// e.g

// Input: intervals = [[1,2],[3,5],[6,7],[8,10],[12,16]], newInterval = [4,8]

// Output: [[1,2],[3,10],[12,16]]

// Explanation: Because the new interval [4,8] overlaps with [3,5],[6,7],[8,10]

// In this example [4,8] overlap [8,10]

while(i < len && intervals[i][1] < newInterval[0]) {

result.add(intervals[i]);

i++;

}

// Merge all overlapping intervals to one considering newInterval

// '<=' means when intervals[i][0] == newInterval[1]

// we still go with merge logic

// e.g same example as above one

while(i < len && intervals[i][0] <= newInterval[1]) {

// Mutate newInterval looply

newInterval[0] = Math.min(intervals[i][0], newInterval[0]);

newInterval[1] = Math.max(intervals[i][1], newInterval[1]);

i++;

}

// Add the union of intervals we got

result.add(newInterval);

// Add all the rest

while(i < len) {

result.add(intervals[i]);

i++;

}

return result.toArray(new int[result.size()][]);

}

}

Time Complexity:O(n)

Space Complexity:O(1)

**Refer to**

<https://leetcode.com/problems/insert-interval/solutions/21602/short-and-straight-forward-java-solution/comments/332527>

public int[][] insert(int[][] intervals, int[] newInterval) {

List<int[]> result = new LinkedList<>();

int i = 0;

// add all the intervals ending before newInterval starts

while (i < intervals.length && intervals[i][1] < newInterval[0]){

result.add(intervals[i]);

i++;

}

// merge all overlapping intervals to one considering newInterval

while (i < intervals.length && intervals[i][0] <= newInterval[1]) {

// we could mutate newInterval here also

newInterval[0] = Math.min(newInterval[0], intervals[i][0]);

newInterval[1] = Math.max(newInterval[1], intervals[i][1]);

i++;

}

// add the union of intervals we got

result.add(newInterval);

// add all the rest

while (i < intervals.length){

result.add(intervals[i]);

i++;

}

return result.toArray(new int[result.size()][]);

}

**Solution 2: Binary Search for potential insert index of new interval by using Find Lower Boundary template (30 min)**

class Solution {

public int[][] insert(int[][] intervals, int[] newInterval) {

List<int[]> list = new ArrayList<int[]>(Arrays.asList(intervals));

// Use "Find Lower Boundary" Binary Search template to get the

// first interval whose 'start' no less than newInterval's 'start'

int index = findLowerBoundary(intervals, newInterval);

// Insert newInterval at tail or at the 'index' position in middle

// of list then shift all elements "at && after" that 'index'

// position one step rightwards

if(index == intervals.length) {

list.add(newInterval);

} else {

list.add(index, newInterval);

}

// Merge Intervals (Same as how L56 conduct)

List<int[]> result = new ArrayList<int[]>();

int[] prev = null;

for(int[] interval : list) {

if(prev == null || prev[1] < interval[0]) {

result.add(interval);

prev = interval;

} else {

prev[1] = Math.max(interval[1], prev[1]);

}

}

return result.toArray(new int[result.size()][]);

}

private int findLowerBoundary(int[][] intervals, int[] newInterval) {

int left = 0;

int right = intervals.length - 1;

while(left <= right) {

int mid = left + (right - left) / 2;

if(intervals[mid][0] >= newInterval[0]) {

right = mid - 1;

} else {

left = mid + 1;

}

}

return left;

}

}

Time complexity: O(N)

Binary search will take O(log⁡N) time, but inserting into the list at the returned position will take O(N) time. Then iterating over the intervals and merging them with intervals ahead of it will take another O(N) time. Hence, the total time complexity will equal O(N)

Space complexity: O(1)

Inserting an interval into the list will take O(1) space. Therefore, apart from the list we return, the total space complexity would be constant.

**Refer to**

<https://leetcode.com/problems/insert-interval/editorial/>

#### **Approach 2: Binary Search**

**Intuition**

The only difference with this approach would be that instead of using linear search to find the suitable position of newInterval, we can use binary search as the list of intervals is sorted in order of their start time.

We need to find the first interval in the list intervals having a start value no less than the start value of newInterval.

Apart from this change, the logic remains the same for this approach; we insert the interval at its place using binary search and then merge the overlapping intervals using the same algorithm we used previously.

**Algorithm**

Insert the newInterval into the given list intervals using binary search. Find the index using binary search and if it's equal to the size of the list, then add the interval to the end of the list; otherwise, insert it at the respective position.

Iterate over the intervals in the list intervals; for each interval currInterval

Iterate over the intervals ahead of it in the list (including itself), and if the two interval overlaps, update currInterval to the merged interval of these two intervals and move on to the next interval.

Decrement the loop counter variable, as it will be incremented again in the outer loop, and if we don't decrement it here, the next interval will be missed.

Insert the interval currInterval in the list answer.

Return answer.

<https://leetcode.com/problems/insert-interval/solutions/3057241/java-c-100-solution-insert-interval/>

class Solution {

boolean isOverlap(int[] a, int[] b) {

// "criss-crossing" intersection

// Case 1:

// b[0] b[1]

// -----------------------

// a[0] a[1]

// ---------------

// Case 2:

// b[0] b[1]

// -----------------------

// a[0] a[1]

// ---------------

// Two styles:

// return Math.min(a[1], b[1]) - Math.max(a[0], b[0]) >= 0;

return b[0] <= a[1] && a[0] <= b[1];

}

int[] merge(int[] a, int[] b) {

int[] newInterval = {Math.min(a[0], b[0]), Math.max(a[1], b[1])};

return newInterval;

}

int lowerBound(int[][] intervals, int[] newInterval) {

if(intervals.length == 0) {

return 0;

}

int start = 0, end = intervals.length - 1;

while(start <= end) {

int mid = (start + end) / 2;

if(intervals[mid][0] > newInterval[0]) {

end = mid - 1;

} else {

start = mid + 1;

}

}

return start;

}

public int[][] insert(int[][] intervals, int[] newInterval) {

List<int[]> list = new ArrayList<>(Arrays.asList(intervals));

int index = lowerBound(intervals, newInterval);

if (index != intervals.length)

list.add(index, newInterval);

else

list.add(newInterval);

intervals = list.toArray(new int[list.size()][2]);

List<int[]> answer = new ArrayList<>();

for (int i = 0; i < intervals.length; i++) {

int[] temp = {intervals[i][0], intervals[i][1]};

while (i < intervals.length && isOverlap(temp, intervals[i]))

temp = merge(temp, intervals[i++]);

i--;

answer.add(temp);

}

return answer.toArray(new int[answer.size()][2]);

}

}

**Complexity Analysis**

Here N is the number of intervals in the list.

* Time complexity: O(N)
* Binary search will take O(log⁡N) time, but inserting into the list at the returned position will take O(N) time. Then iterating over the intervals and merging them with intervals ahead of it will take another O(N) time. Hence, the total time complexity will equal O(N).
* Space complexity: O(1).
* Inserting an interval into the list will take O(1) space. Therefore, apart from the list we return, the total space complexity would be constant.