**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

Solution

Approach 1: Greedy.

**Greedy algorithms**

Greedy problems usually look like "Find minimum number of *something* to do *something*" or "Find maximum number of *something* to fit in *some conditions*", and typically propose an unsorted input.

The idea of greedy algorithm is to pick the *locally* optimal move at each step, that will lead to the *globally* optimal solution.

The standard solution has \mathcal{O}(N \log N)O(*N*log*N*) time complexity and consists of two parts:

* Figure out how to sort the input data (\mathcal{O}(N \log N)O(*N*log*N*) time). That could be done directly by a sorting or indirectly by a heap usage. Typically sort is better than the heap usage because of gain in space.
* Parse the sorted input to have a solution (\mathcal{O}(N)O(*N*) time).

Please notice that in case of well-sorted input one doesn't need the first part and the greedy solution could have \mathcal{O}(N)O(*N*) time complexity, [here is an example](https://leetcode.com/articles/gas-station/).

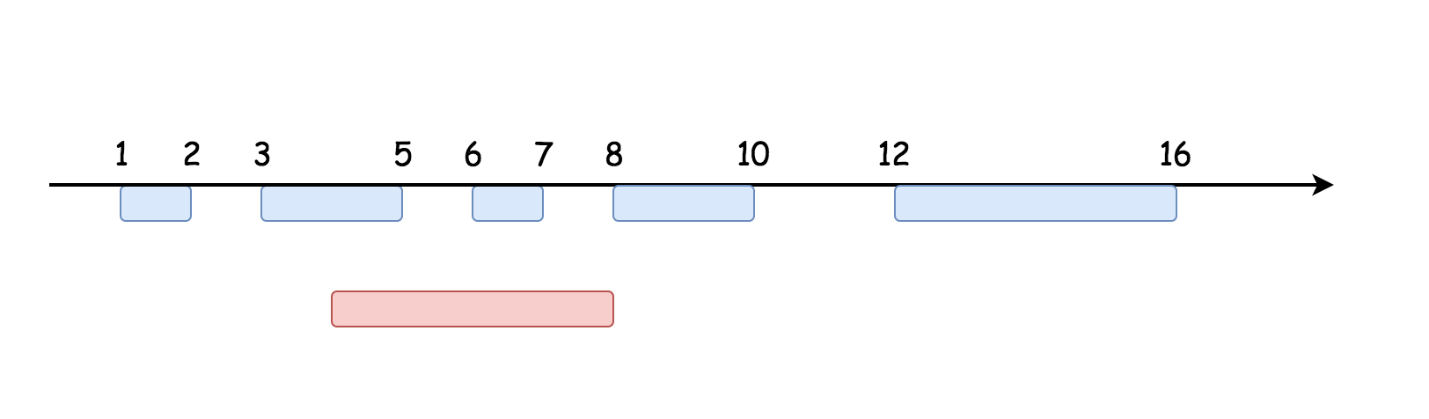
How to prove that your greedy algorithm provides globally optimal solution?

Usually you could use the [proof by contradiction](https://en.wikipedia.org/wiki/Proof_by_contradiction).

**Intuition**

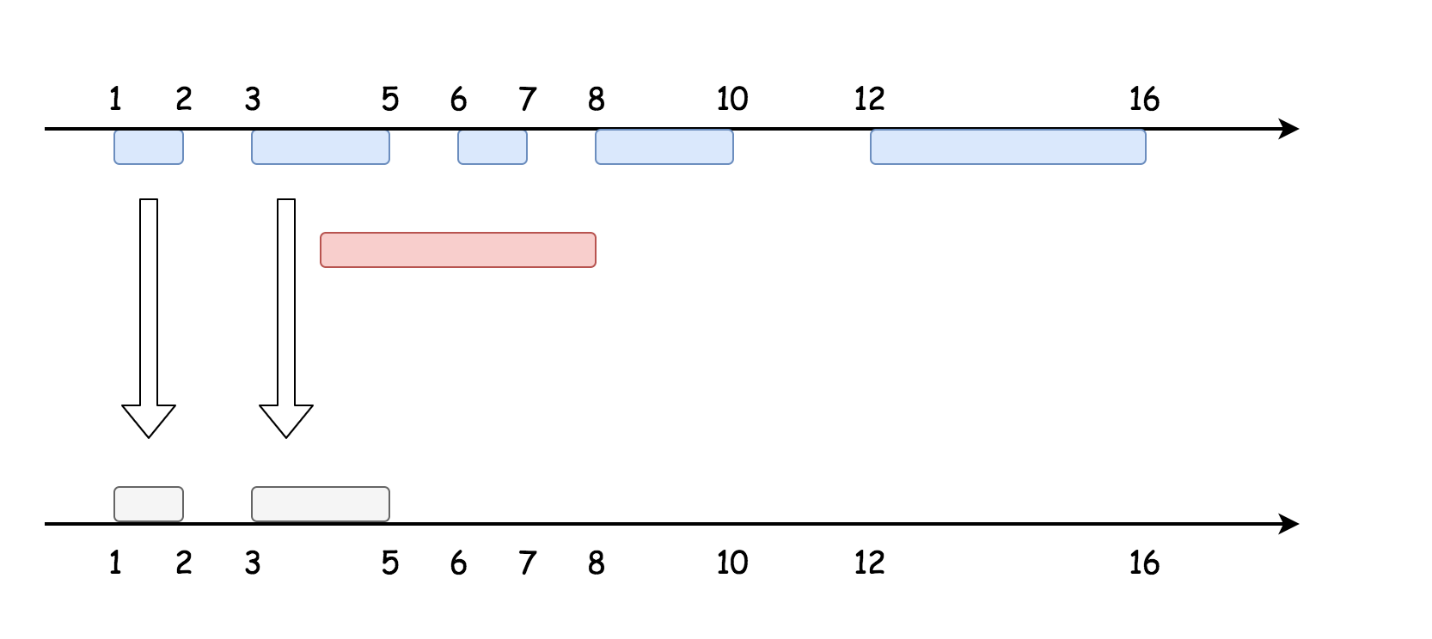
Here we have an example of a greedy problem with a well-sorted input, and hence the algorithm time complexity should be \mathcal{O}(N)O(*N*).

Let's consider the following intervals

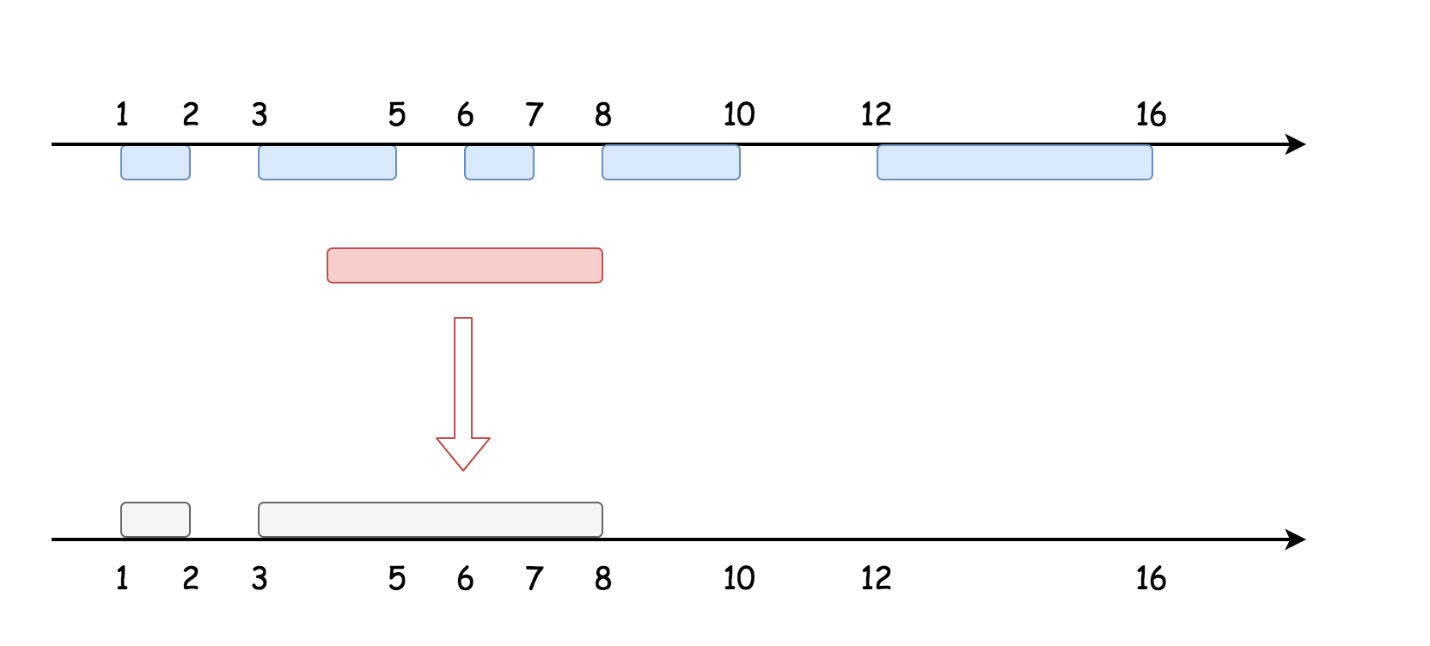


The straightforward one-pass strategy could be implemented in three steps.

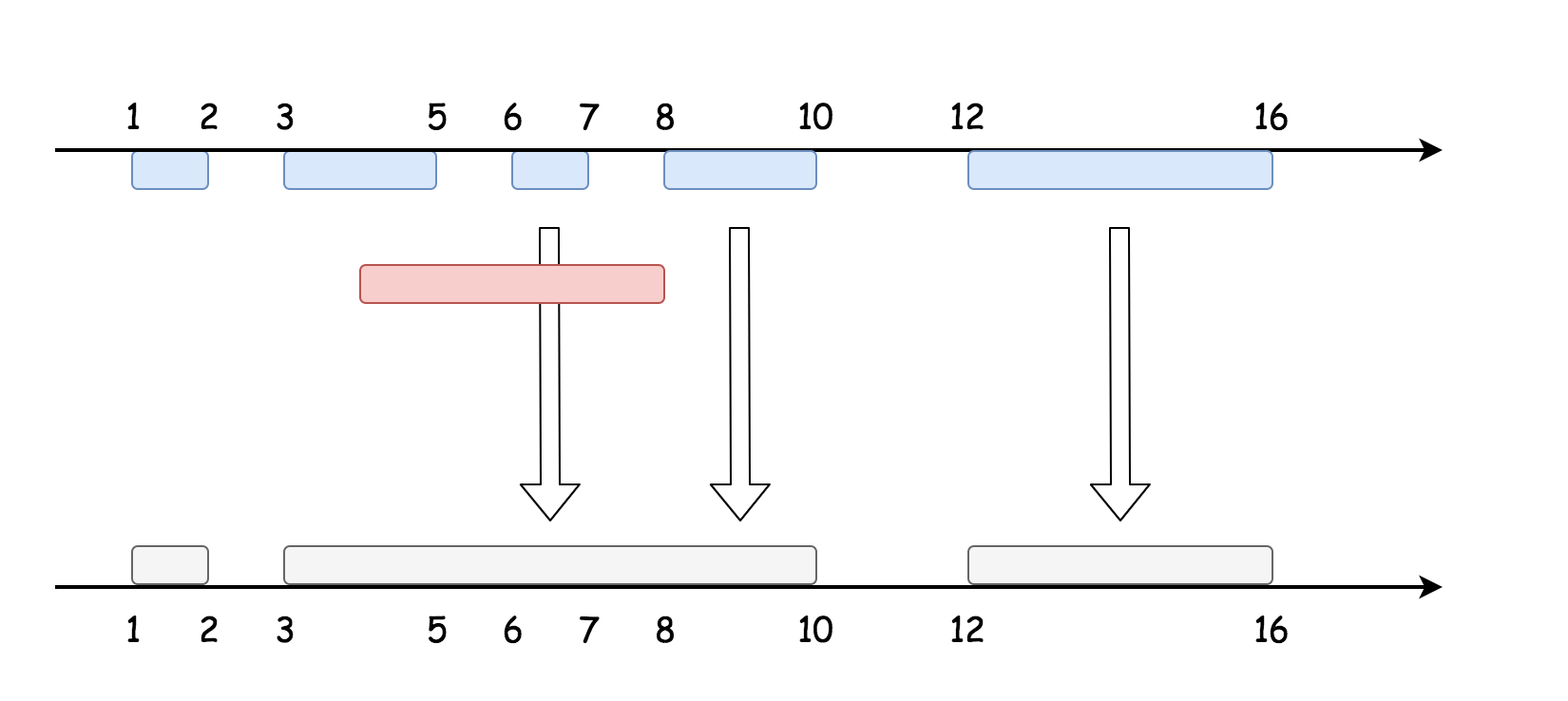
1 . Add to the output all the intervals starting before newInterval.



2 . Add to the output newInterval, merge it with the last added interval if needed.



3 . Add the next intervals one by one, merge if needed.



Basically, the same strategy [as here](https://leetcode.com/articles/merge-intervals/), with an additional care to add the new interval in its proper position in order not to destroy the well-sorted input.

**Algorithm**

Here is the algorithm :

* Add to the output all the intervals starting before newInterval.
* Add to the output newInterval. Merge it with the last added interval if newInterval starts before the last added interval.
* Add the next intervals one by one. Merge with the last added interval if the current interval starts before the last added interval.

**Implementation**

class Solution {

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

// init data

int newStart = newInterval[0], newEnd = newInterval[1];

int idx = 0, n = intervals.length;

LinkedList<int[]> output = new LinkedList<int[]>();

// add all intervals starting before newInterval

while (idx < n && newStart > intervals[idx][0])

output.add(intervals[idx++]);

// add newInterval

int[] interval = new int[2];

// if there is no overlap, just add the interval

if (output.isEmpty() || output.getLast()[1] < newStart)

output.add(newInterval);

// if there is an overlap, merge with the last interval

else {

interval = output.removeLast();

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

output.add(interval);

}

// add next intervals, merge with newInterval if needed

while (idx < n) {

interval = intervals[idx++];

int start = interval[0], end = interval[1];

// if there is no overlap, just add an interval

if (output.getLast()[1] < start) output.add(interval);

// if there is an overlap, merge with the last interval

else {

interval = output.removeLast();

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

output.add(interval);

}

}

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

}

}

**Complexity Analysis**

* Time complexity : \mathcal{O}(N)O(*N*) since it's one pass along the input array.
* Space complexity : \mathcal{O}(N)O(*N*) to keep the output.