Insert Interval (medium)

We'll cover the following

Problem Statement

Given a list of non-overlapping intervals sorted by their start time, **insert a given interval at the correct position** and merge all necessary intervals to produce a list that has only mutually exclusive intervals.

Example 1:

```
Input: Intervals=[[1,3], [5,7], [8,12]], New Interval=[4,6]
Output: [[1,3], [4,7], [8,12]]
Explanation: After insertion, since [4,6] overlaps with [5,7], we merged them into one [4,7].
```

Example 2:

```
Input: Intervals=[[1,3], [5,7], [8,12]], New Interval=[4,10]
Output: [[1,3], [4,12]]
Explanation: After insertion, since [4,10] overlaps with [5,7] & [8,12], we merged them into [4,
12].
```

Example 3:

```
Input: Intervals=[[2,3],[5,7]], New Interval=[1,4]
Output: [[1,4], [5,7]]
Explanation: After insertion, since [1,4] overlaps with [2,3], we merged them into one [1,4].
```

Try it yourself

Try solving this question here:

Java | Python3 | JS | C++

```python
def insert(intervals, new_interval):
  merged = []
  # TODO: Write your code here
  return merged


def main():
  print("Intervals after inserting the new interval: " + str(insert([[1, 3], [5, 7], [8, 12]], [4, 6])))
  print("Intervals after inserting the new interval: " + str(insert([[1, 3], [5, 7], [8, 12]], [4, 10])))
  print("Intervals after inserting the new interval: " + str(insert([[2, 3], [5, 7]], [1, 4])))


main()
```

Run | Save | Reset

Solution

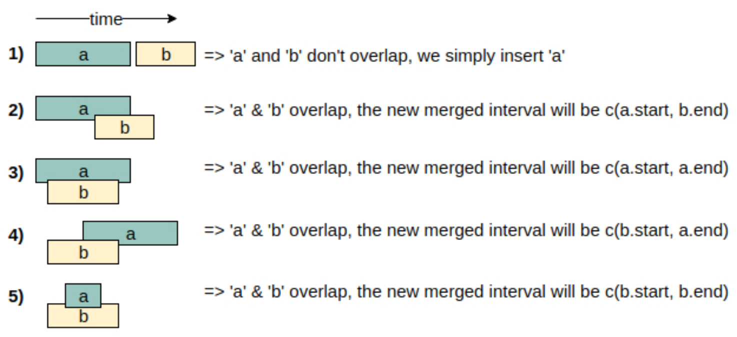
Solution

If the given list was not sorted, we could have simply appended the new interval to it and used the `merge()` function from Merge Intervals. But since the given list is sorted, we should try to come up with a solution better than $O(N * logN)$

When inserting a new interval in a sorted list, we need to first find the correct index where the new interval can be placed. In other words, we need to skip all the intervals which end before the start of the new interval. So we can iterate through the given sorted listed of intervals and skip all the intervals with the following condition:

```
intervals[i].end < newInterval.start
```

Once we have found the correct place, we can follow an approach similar to Merge Intervals to insert and/or merge the new interval. Let's call the new interval 'a' and the first interval with the above condition 'b'. There are five possibilities:


time
1) a b => 'a' and 'b' don't overlap, we simply insert 'a'
2) a b => 'a' & 'b' overlap, the new merged interval will be c(a.start, b.end)
3) a b => 'a' & 'b' overlap, the new merged interval will be c(a.start, a.end)
4) a b => 'a' & 'b' overlap, the new merged interval will be c(b.start, a.end)
5) a b => 'a' & 'b' overlap, the new merged interval will be c(b.start, b.end)

The diagram above clearly shows the merging approach. To handle all four merging scenarios, we need to do something like this:

```
    c.start = min(a.start, b.start)
    c.end = max(a.end, b.end)
```

Our overall algorithm will look like this:

1. Skip all intervals which end before the start of the new interval, i.e., skip all `intervals` with the following condition:

```
    intervals[i].end < newInterval.start
```

2. Let's call the last interval 'b' that does not satisfy the above condition. If 'b' overlaps with the new interval (a) (i.e. `b.start <= a.end`), we need to merge them into a new interval 'c':

```
    c.start = min(a.start, b.start)
    c.end = max(a.end, b.end)
```

3. We will repeat the above two steps to merge 'c' with the next overlapping interval.

Code

Here is what our algorithm will look like:

Java | Python3 | C++ | JS

```
def insert(intervals, new_interval):
  merged = []
  i, start, end = 0, 0, 1

  # skip (and add to output) all intervals that come before the 'new_interval'
  while i < len(intervals) and intervals[i][end] < new_interval[start]:
    merged.append(intervals[i])
    i += 1
```

```python
  # merge all intervals that overlap with 'new_interval'
  while i < len(intervals) and intervals[i][start] <= new_interval[end]:
    new_interval[start] = min(intervals[i][start], new_interval[start])
    new_interval[end] = max(intervals[i][end], new_interval[end])
    i += 1

  # insert the new_interval
  merged.append(new_interval)

  # add all the remaining intervals to the output
  while i < len(intervals):
    merged.append(intervals[i])
    i += 1
  return merged


def main():
  print("Intervals after inserting the new interval: " + str(insert([[1, 3], [5, 7], [8, 12]], [4, 6])))
  print("Intervals after inserting the new interval: " + str(insert([[1, 3], [5, 7], [8, 12]], [4, 10])))
  print("Intervals after inserting the new interval: " + str(insert([[2, 3], [5, 7]], [1, 4])))


main()
```

Run Save Reset

Time complexity

As we are iterating through all the intervals only once, the time complexity of the above algorithm is $O(N)$, where ‘N’ is the total number of intervals.

Space complexity

The space complexity of the above algorithm will be $O(N)$ as we need to return a list containing all the merged intervals.

Back
Merge Intervals (medium)

Next
Intervals Intersection (medium)

✓ Completed