INTERVAL

No.	Problem Statement	Solution	Time complexity	Space complexity
1	**Meeting Rooms**			
	Given array of time 'intervals', determine if can attend all meetings. Ex. intervals = [[0,30],[5,10],[15,20]] -> false	- Sort 'intervals' by start time, check adjacent meetings, if overlap return false `if(intervals[i][1] > intervals[i + 1][0]) return false`	O(NlogN)	O(1)
2	**Meeting Rooms II**			

	Given array of time 'intervals', determine min # of meeting rooms required.
	Ex. intervals = [[0,30],[5,10],[15,20]] -> 2
		- Idea: Use a **Priority Queue** to keep track of all the meetings that require a **seperate** room
		- Store **end time** for the meeting --> indicating when the meeting will end in the current room
		`// sort intervals by start time`
		`sort(intervals.begin(), intervals.end());`
		`priority_queue<int, vector<int>, greater<int>> pq;`
		`pq.push(intervals[0][1]);`
		`for (int i = 1; i < intervals.size(); i++)`
		`// if no overlap then schedule the current meeting in this room`
		`if (intervals[i][0] >= pq.top()) pq.pop();`
		`// add the end time for the scheduled meeting`
		`pq.push(intervals[i][1]);`
		`return pq.size();`
			O(NlogN)	O(N)

3	**Insert Interval**			
	Given an array of non-overlapping intervals, 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).	- Suppose newInterval = (x,y) and currentInterval = (a,b) 1) Add all the intervals that do not overlap with the newInterval i.e **x > b** 2) Merge intervals that overlap with the newInterval **if(a<=y && b>=x)** x = min(x, a) y = max(y, b) 3) Add all the remaining intervals that do not overlap with the newInterval	O(N)	O(1)
4	**Merge Intervals**			

	Given an array of intervals, merge all overlapping intervals and return an array of the non-overlapping intervals.	- Sort 'intervals' by start time
		`x = intervals[0][0]`
		`y = intervals[0][1]`
		`for(int i=1; i<n; i++){`
		`if(intervals[i][0] <= y)`
		`x = min(x, intervals[i][0]);`
		`y = max(y, intervals[i][1]);`
		`else`
		`ans.push_back({x, y})`
		`x = intervals[i][0]`
		`y = intervals[i][1]`
		`ans.push_back({x, y});`
			O(NlogN)	O(1)

5	**Non-overlapping Intervals**			

	Given array of intervals, return min # of intervals to remove to make the rest of the intervals non-overlapping.	- Idea: **Remove interval with longer end point, since it will always overlap more compared to the shorter one**
		- Sort 'intervals' by start time
		1) Initialize 'end_point' : intervals[0][1]
		2) Iterate through the 'intervals' array --> i: 1 -- > n-1
		- `if(intervals[i][0] < end_point)` // Current interval overlaps
		`end_point = min(end_point, intervals[i][1])`
		`cnt++`
		-`else end_point = intervals[i][1]`
			O(NlogN)	O(1)

6	**Minimum Interval to Include Each Query**			

INTERVAL

No.	Problem Statement	Solution	Time complexity	Space complexity

	Given 'intervals' array & 'queries' array. The answer to the jth query is the size of the smallest interval i. Size of an interval 'i' is righti - lefti + 1.
	Ex. intervals = [[1,4],[2,4],[3,6],[4,4]], queries = [2,3,4,5] -> [3,3,1,4]
		- **Priority_queue<pair<int, int>> pq** --> **{size of interval, end of interval}**
		pq.top() will have the size of the minimum interval for current query 'q'
		- **Unordered_map<int, int> mp** --> **{query, size of interval}**
		- **Vector<int> sorted_quereis** --> **Sorted version of the 'queries' array**
		- Traverse the array sorted_queries: i: 0--> n-1
		1. Add all the 'intervals' in the 'pq' that contain 'q'
		`while(j<intervals.size() && q >= intervals[j][0])`
		2. Remove all the intervals from 'pq' that don't contain 'q', as the 'queries' are sorted and if the interval doesn't contain 'q' then it won't contain q+1,q+2,....
		`while(!pq.empty() && pq.top().second < q) pq.pop()`
			)(N*logN + M*logM)
			
			N=intervals.size()
			M=queries.size()
				O(N+M)