① Meeting Rooms - 2
② Minimum Platforms
③ Car Pooling

① Approach

make 2 different arrays : starting Point & ending point.

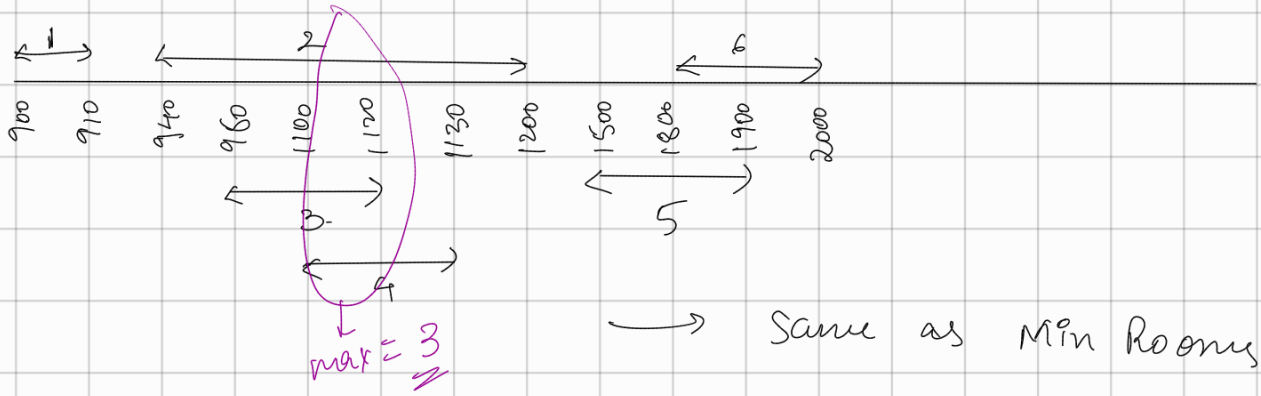
& use 2 pointer approach from i=0 & j=0 as follows

```
int i=0, j=0, Rooms=0, maxRooms=0;
while (i<n && j<n){
    if (startArr[i] < endArr[j]){
        Rooms++;
        i++;
    }else{
        Rooms--;
        j++;
    }
    maxRooms = Math.max(Rooms, maxRooms);
}
return maxRooms;
```

TC: O(N + N logN) , SC : O(N)

② Minimum Platforms

900	940	0950	1100	1500	1800
910	1200	1120	1130	1900	2000
1	2	3	4	5	6

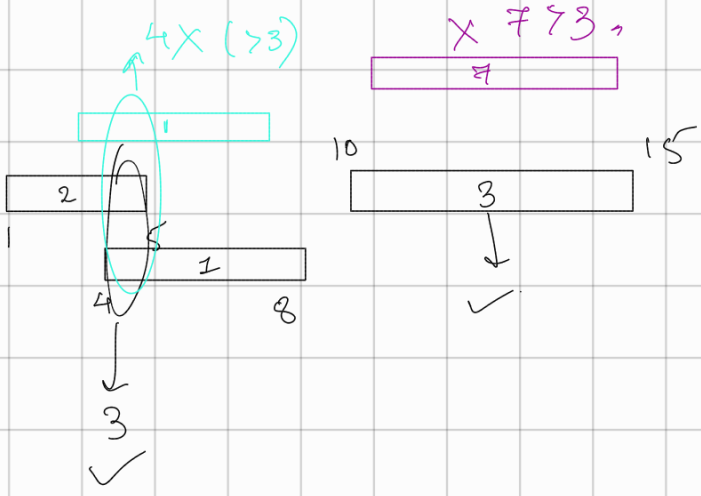


→ Same as Min Rooms ↑ ①

③ Car Pooling

Cap = 3

{2, 1, 5},
{1, 4, 8},
{3, 10, 15},



① Check if single capacity > max capacity ⇒ return false
② If trips.size() == 0 → return true

Tree map → implemented using Red Black or self balancing tree

$O(2n \log n + n) \Rightarrow TC = O(N \log N)$

↓ insertions

$SC = O(N)$

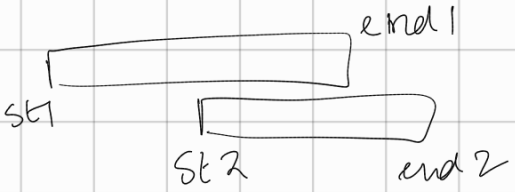
④ Merge Intervals.

Case 1 : st1 == st2 & end1 < end2

Case 2 : st1 == st2 & end1 > end2

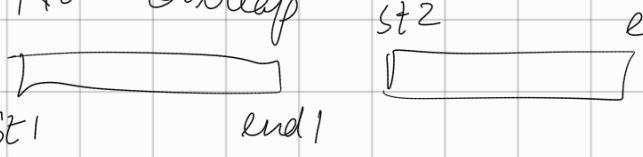
} complete overlap

Case 3 :



} Partial overlap

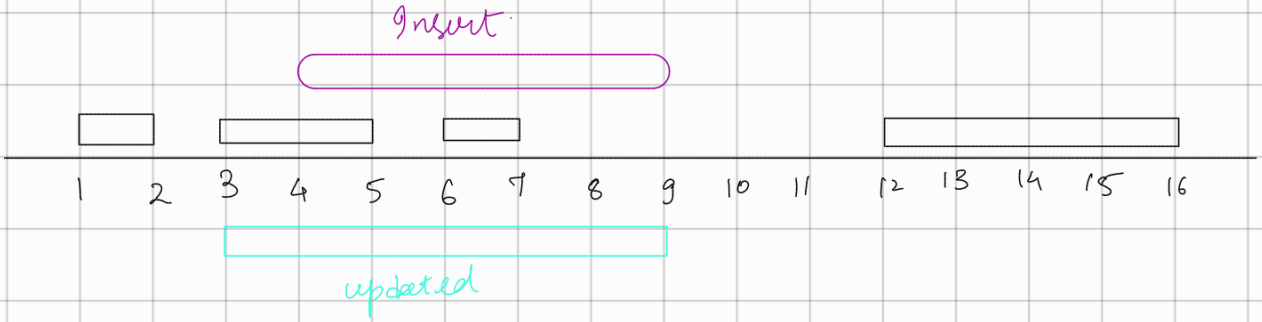
Case 4 : No overlap



} No overlap

⑤ Insert Intervals.

Eg: [[1,2] [3,5], [6,7], [12,16]] , [4,9]



Approach

1) Search first interval for end time <= insert start time

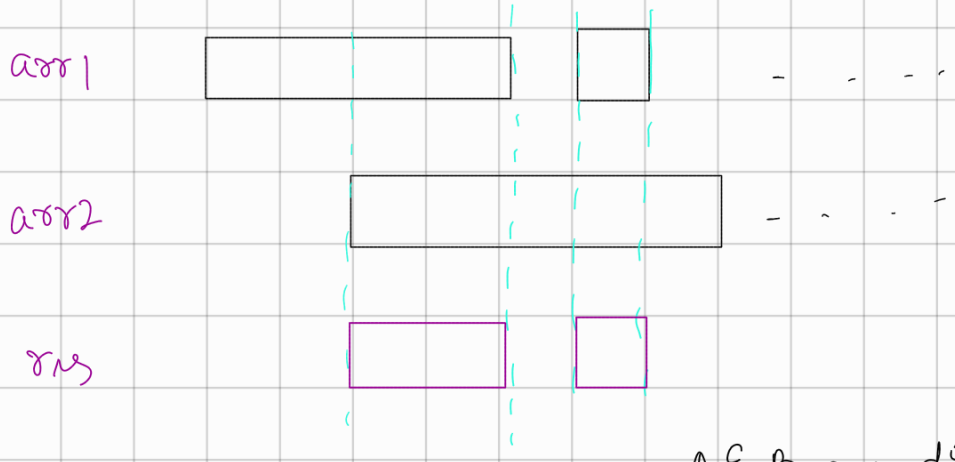
2) Search last interval for "max" start time <= insert end time

3) Update the merged intervals

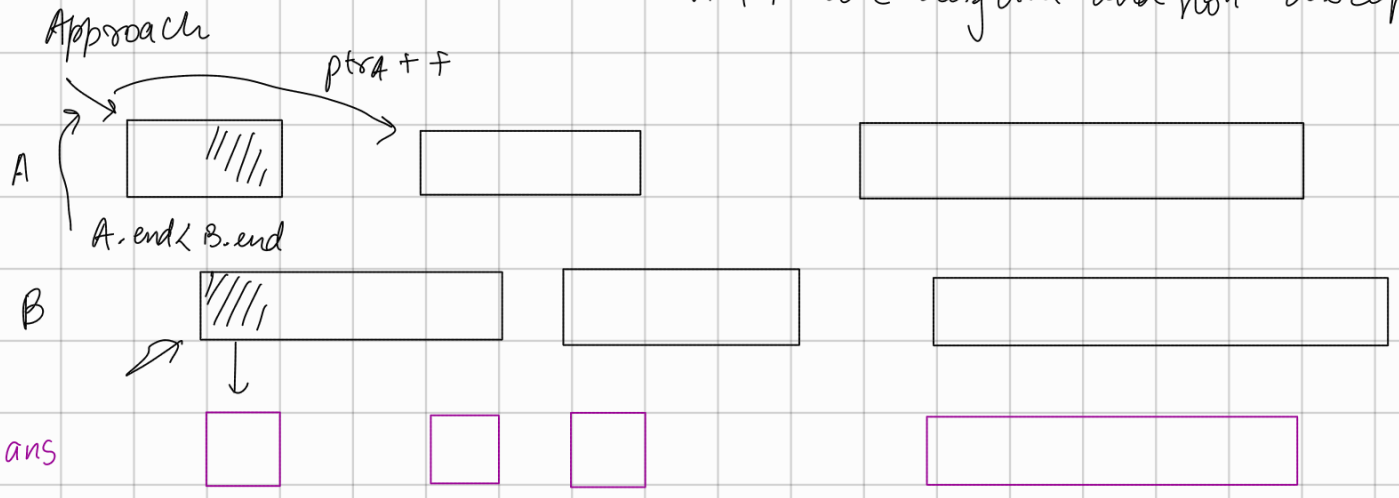
start of merged = min(first.start, insert.start);
end of merged = max(last.end, insert.end);

* Corner case: no merging.

⑥ Intervals Intersection



A & B are disjoint and non-overlapping



Increment the pointer that is ending prior