

Greedy Algorithms

we will start by
9:10

~~WHAT~~

local optimal choice at each step helps
solving global optimum (minima) maxima

APPLICATIONS

- ① Fractional knapsack

- ② Activity Selection / Overlapping Intervals / Meeting Rooms

- ③ Minimum Spanning Tree
(Prim's (PQ), Kruskal(DSU))

- ④ Huffman Encoding & Decoding /
optimal Merge Pattern

- ⑤ Single source Shortest Path
Algo { DIJKSTRA }

- ⑥ Job Sequencing

- ⑦ Problems based on $\log n$
SORTING

Lecture 1

① Job Sequencing

② Meeting Rooms - I

③ Disjoint Intervals - I

④ Disjoint Intervals - II

⑤ Maximum Chain Length

⑥ Minimum Balloon Burst

Saturday Morning

Lecture 2

① Meeting Rooms - II

② Minimum Platforms

③ Car Pooling

④ Merge Overlapping Intervals

⑤ Insert Interval

⑥ Intervals Intersection

Saturday Evening

Lecture ③

- ① Circular Tour (Salesforce)
- ② Car Fleet (Sprinklr)
- ③ Two City Scheduling
- ④ Candy/Temple Offerings
- ⑤ Chocolate Distribution
- ⑥ Queue Reconstruction Height

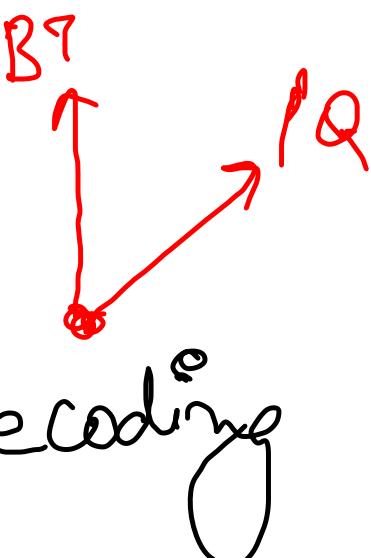
Sunday Morning

Lecture ④

- ① Biased Standings
- ② Defense Kingdom
- ③ Georgivia
- ④ Amplifiers
- ⑤ Load Balancing

Huffman Coding & Decoding

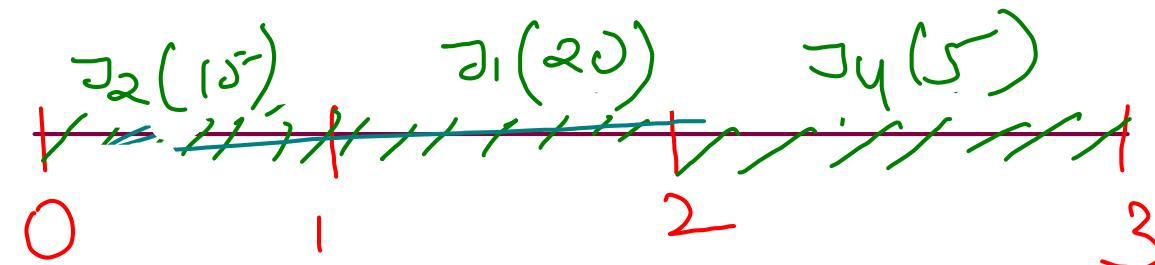
Sunday Evening



Job Sequencing

1 job \rightarrow 1 unit of time
maximum prof's

$n=5$	$i \Rightarrow 0$	1	2	3	4	5
Jobs	J_1	J_2	J_3	J_4	J_5	
profits	20	15	10	5	1	
deadlines	2	2	1	3	3	



- ① Order of picking jobs
 \Rightarrow decreasing order of profit

- ② Picked job should be placed as last as possible

$J_1 \cancel{=} J_3$

Benefit = $\cancel{20} \cancel{=} 40$

```

public static class MyComparator implements Comparator<Job>{
    public int compare(Job obj1, Job obj2){
        if(obj1.profit != obj2.profit){
            return obj2.profit - obj1.profit;
        }
        return obj2.deadline - obj1.deadline;
    }
}

//Function to find the maximum profit and the number of jobs done.
int[] Jobscheduling(Job arr[], int n)
{
    Arrays.sort(arr, new MyComparator());  $\Rightarrow$  n log n
    int maxDeadline = 0;
    for(int i=0; i<n; i++){
        maxDeadline = Math.max(arr[i].deadline, maxDeadline);
    }

    boolean[] slots = new boolean[maxDeadline];
    int maxProfit = 0;
    int jobsAllocated = 0;

    for(int i=0; i<n; i++) {
        for(int j=arr[i].deadline-1; j>=0; j--){
            if(slots[j] == false){
                slots[j] = true;
                jobsAllocated++;
                maxProfit += arr[i].profit;
                break;
            }
        }
    }

    return new int[]{jobsAllocated, maxProfit};
}

```

$O(dn + n \log n)$

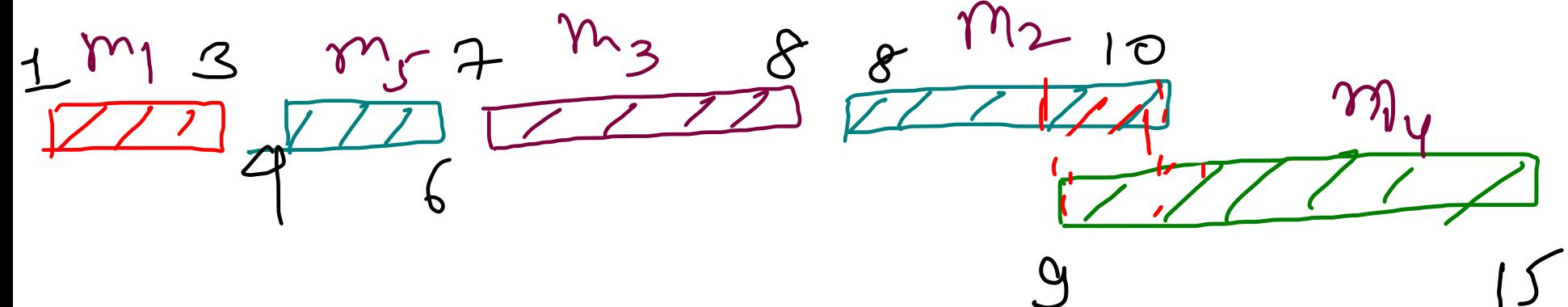
$d \times n$

max deadline

Meeting Rooms - I

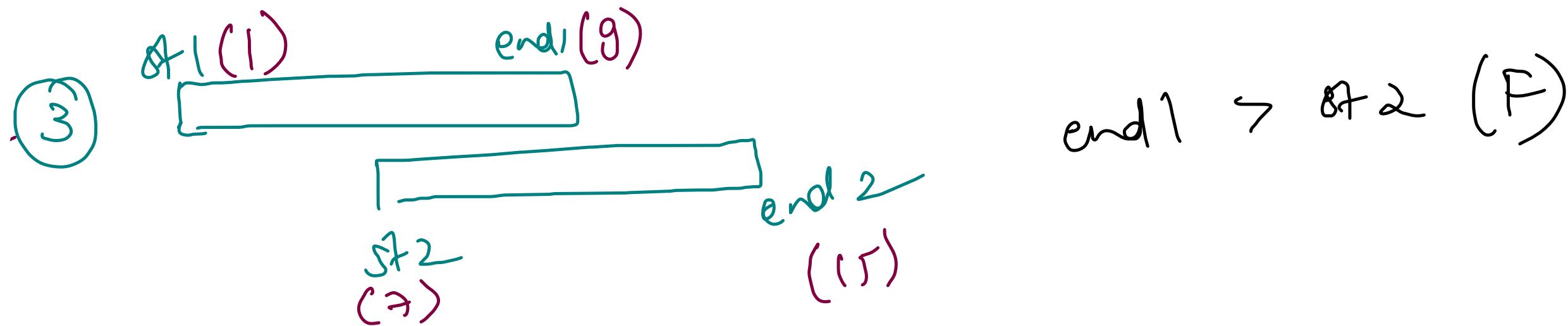
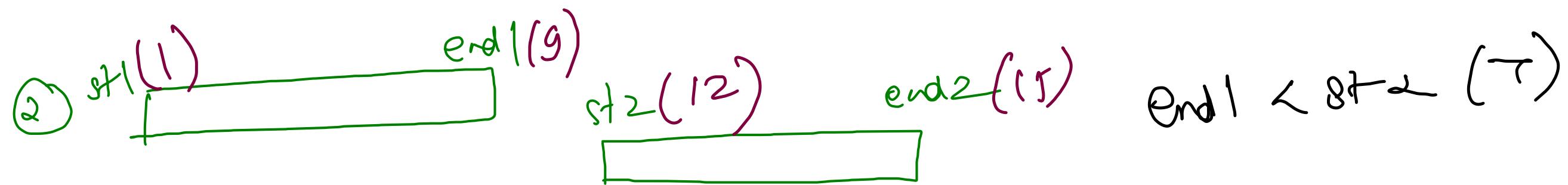
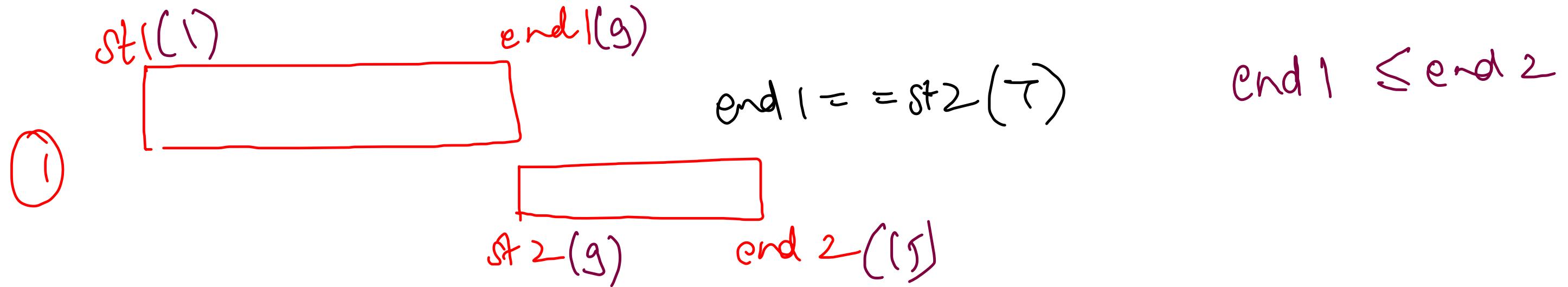
One room \Rightarrow one meeting at a time

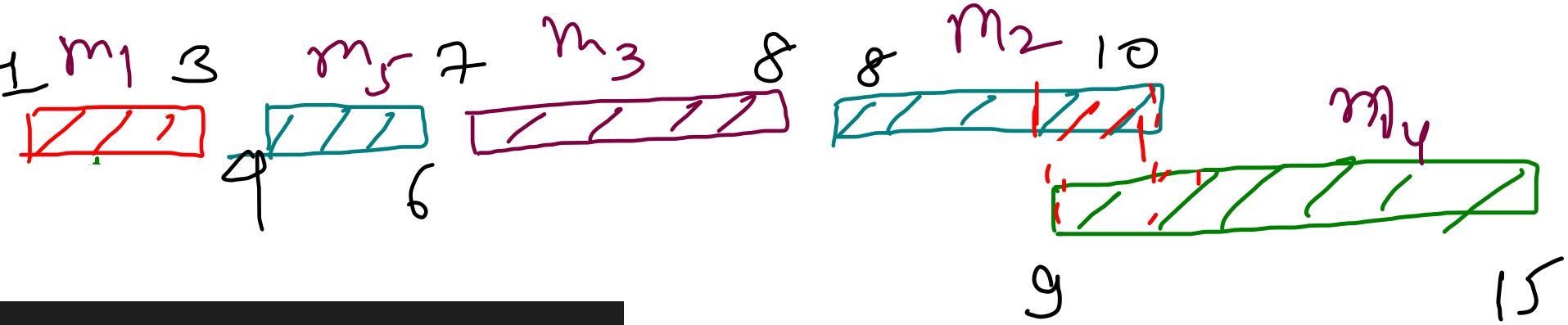
	0	1
① m_1	1	3
② m_2	8	10
③ m_3	7	8
④ m_4	9	15
⑤ m_5	4	2



- ① Sort intervals based on ending index
- ② Check overlapping with previous slot.

false





```

public class Solution {
    public static class MyComparator implements Comparator<Interval>{
        public int compare(Interval obj1, Interval obj2){
            if(obj1.end != obj2.end)
                return obj1.end - obj2.end;
            return obj1.start - obj2.start;
        }
    }

    public boolean canAttendMeetings(List<Interval> intervals) {
        Collections.sort(intervals, new MyComparator()); } O(N log N)

        int limit = Integer.MIN_VALUE; // last interval's ending time

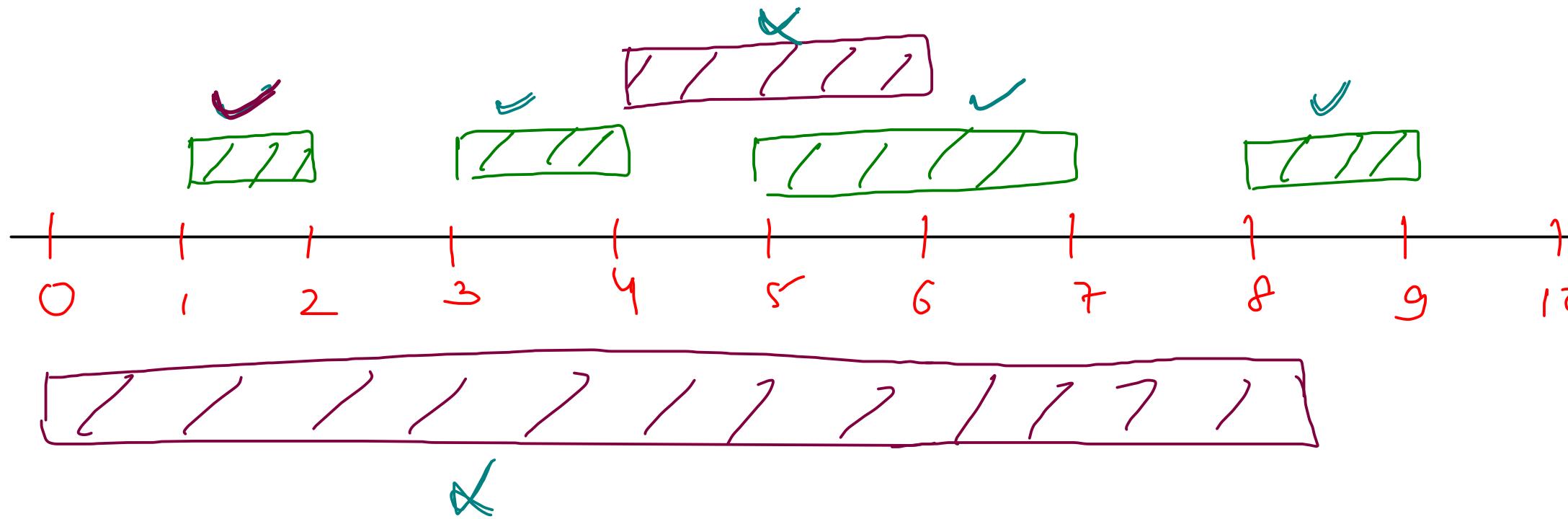
        for(int i=0; i<intervals.size(); i++){
            if(limit > intervals.get(i).start){ }  $\Rightarrow O(N)$ 
                return false;
            }
            limit = intervals.get(i).end;
        }

        return true;
    }
}

```

$s[] =$	1	0	3	8	5	8	4
$f[] =$	2	18	4	9	7	16	
	1	2	3	4	5	6	

1
2
3
4



```

int limit = Integer.MIN_VALUE; // last interval's ending time
int count = 0;

for(int i=0; i<n; i++){
    if(limit < intervals[i].start){
        count++;
        limit = intervals[i].end;
    }
}

return count;

```

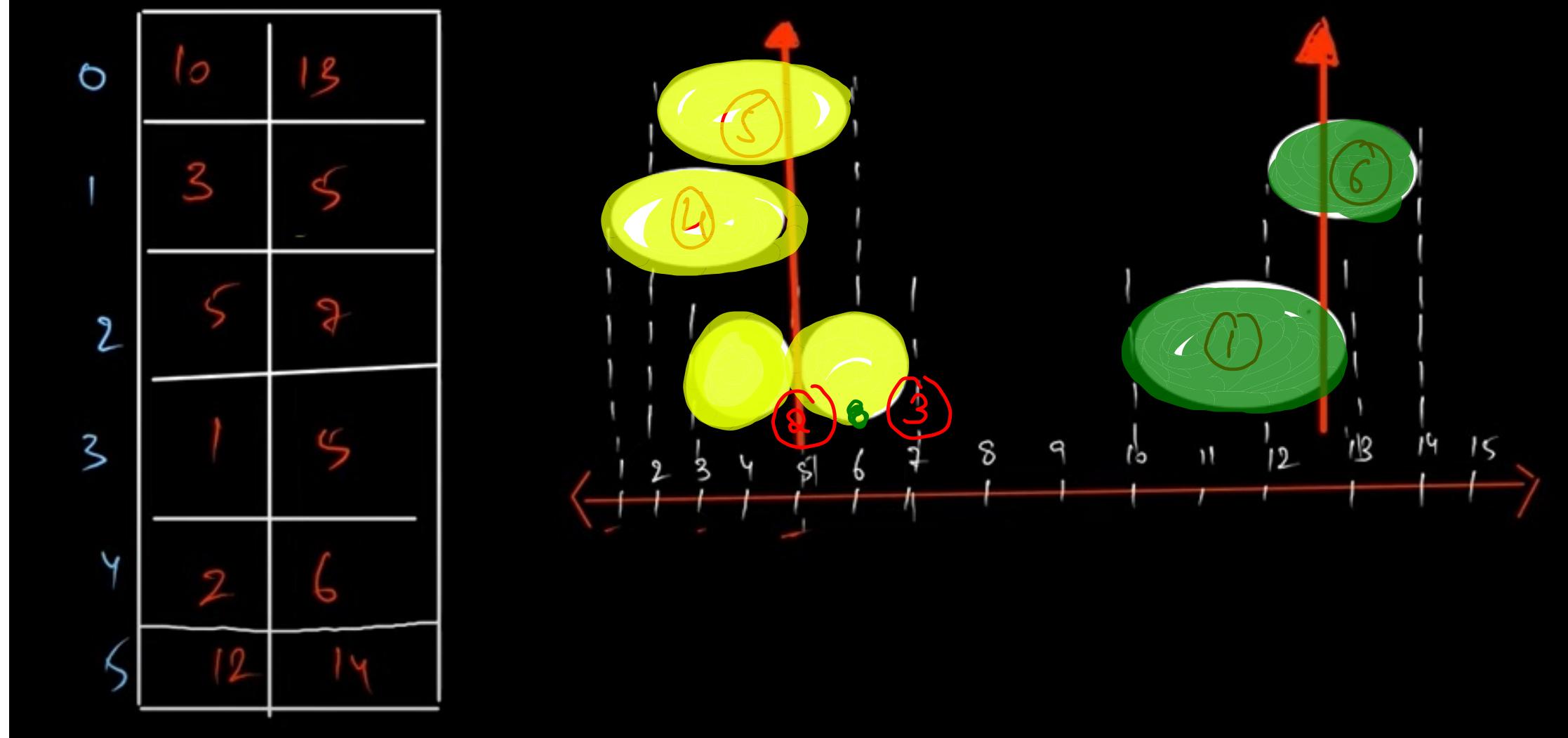
\Rightarrow non-overlapping

\Rightarrow overlapping

$\text{end}_2 = 8 + 1$

Minimum Balloon Bursts

```
coordinates[] : [[10,13],[3,5],[5,7],[1,5],[2,6],[12,14]]  
output : 2
```

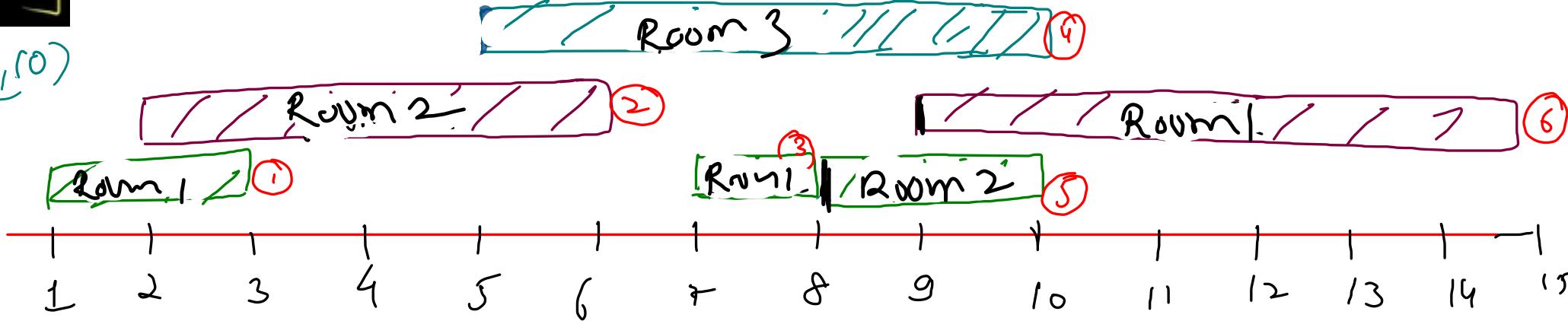


minimum
arrows
= maximum
meetings
that
can be
accommodated
in
one hour

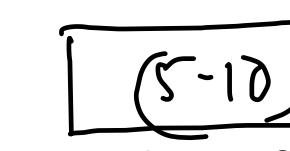
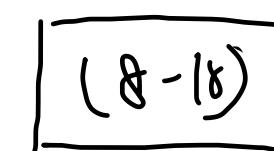
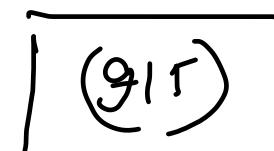
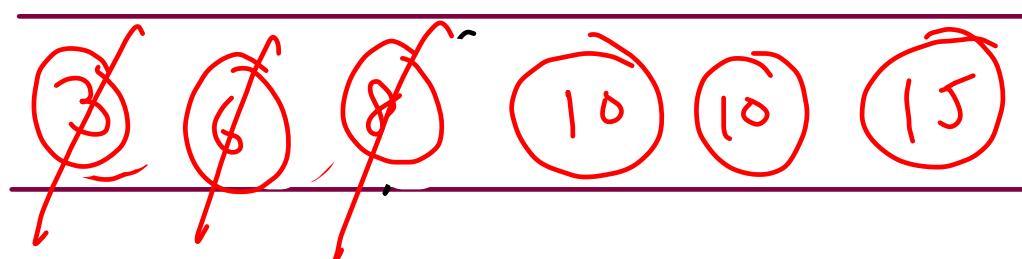
5
1 3
8 10
7 8
9 15
2 6

minimum room

(5,10)



Count of Rooms = ~~0 1 2 3~~



Room 1

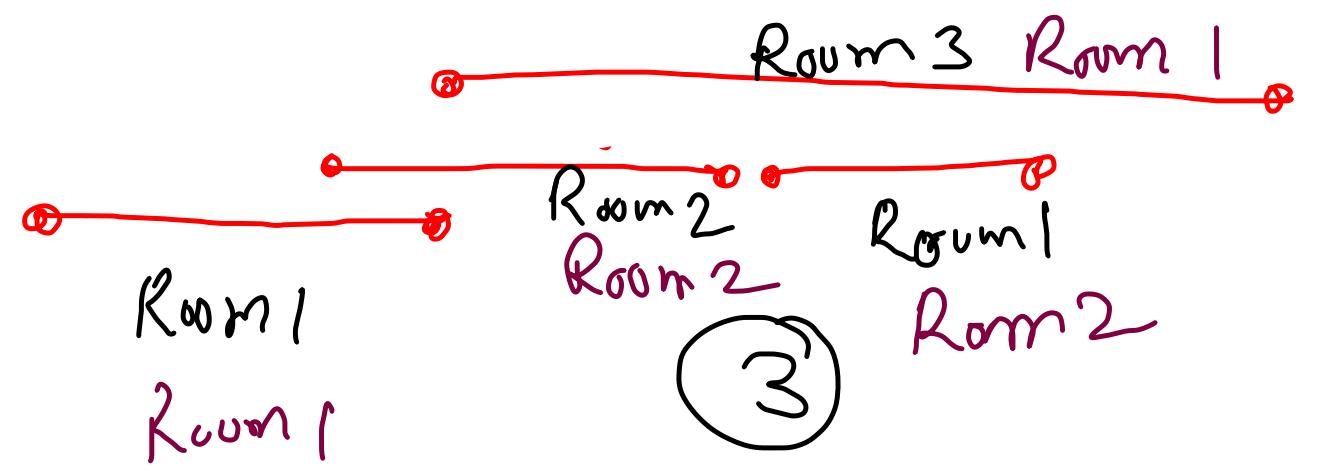
Room 2

Room 3

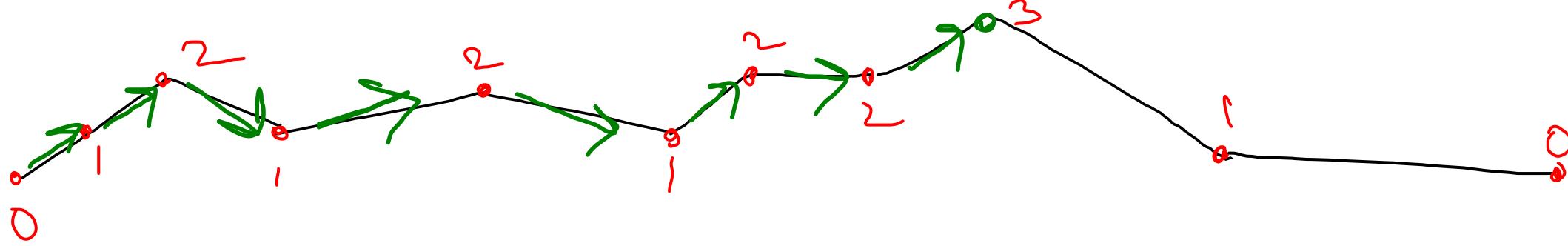
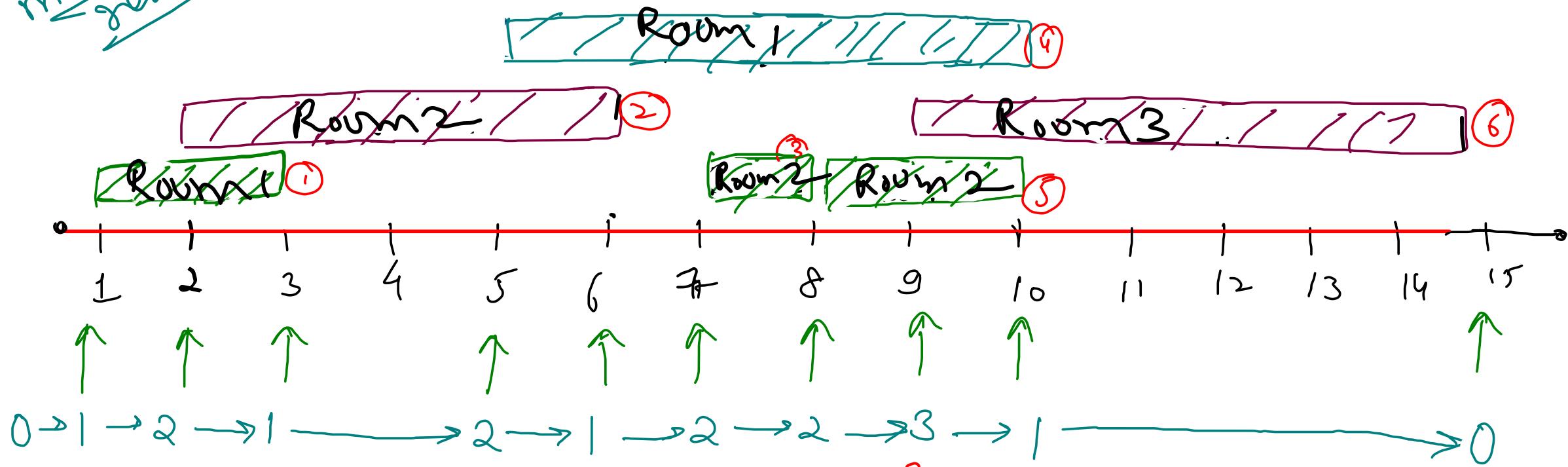
```
public static int meetingRooms(int intervals[][]){  
    int maxRooms = 0;  
    Arrays.sort(intervals, (a, b) -> a[1] - b[1]);  
  
    Queue<Integer> q = new ArrayDeque<>();  
  
    for(int i=0; i<intervals.length; i++){  
        if(q.size() == 0 || q.peek() > intervals[i][0]){  
            maxRooms++;  
        } else {  
            q.remove();  
        }  
        q.add(intervals[i][1]);  
    }  
    return maxRooms;  
}
```

Time $\rightarrow O(N \log N + N)$

Space $\rightarrow O(N)$



minimum
geoms



Room 1

Room 2

Room 3

1 2 5 8 9
3 6 8 10 15
1

```
public int minMeetingRooms(List<Interval> intervals) {  
    ArrayList<Integer> start = new ArrayList<>();  
    ArrayList<Integer> end = new ArrayList<>();  
    for(int i=0; i<intervals.size(); i++){  
        start.add(intervals.get(i).start);  
        end.add(intervals.get(i).end);  
    }  
  
    Collections.sort(start);  
    Collections.sort(end);  
  
    int currentRooms = 0, maxRooms = 0;  
    int startIdx = 0, endIdx = 0;  
  
    while(startIdx < intervals.size()){  
        if(start.get(startIdx) < end.get(endIdx)){  
            startIdx++;  
            currentRooms++;  
        } else if(end.get(endIdx) < start.get(startIdx)){  
            endIdx++;  
            currentRooms--;  
        } else{  
            startIdx++;  
            endIdx++;  
        }  
        maxRooms = Math.max(maxRooms, currentRooms);  
    }  
    return maxRooms;  
}
```

$O(N)$

$O(N \log N)$

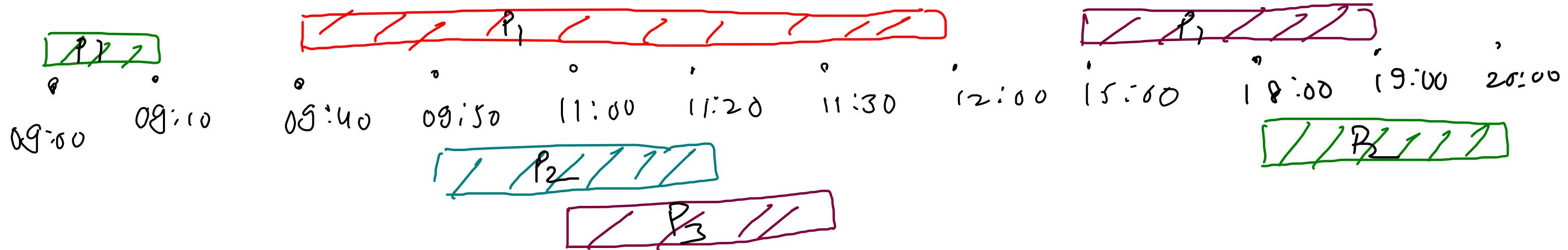
$O(2N)$

minimum Platforms

{ same as meeting Rooms - 11 }

{0900, 0940, 0950, 1100, 1500, 1800}

{0910, 1200, 1120, 1130, 1900, 2000}



Car Pooling

{2, 1, 5}

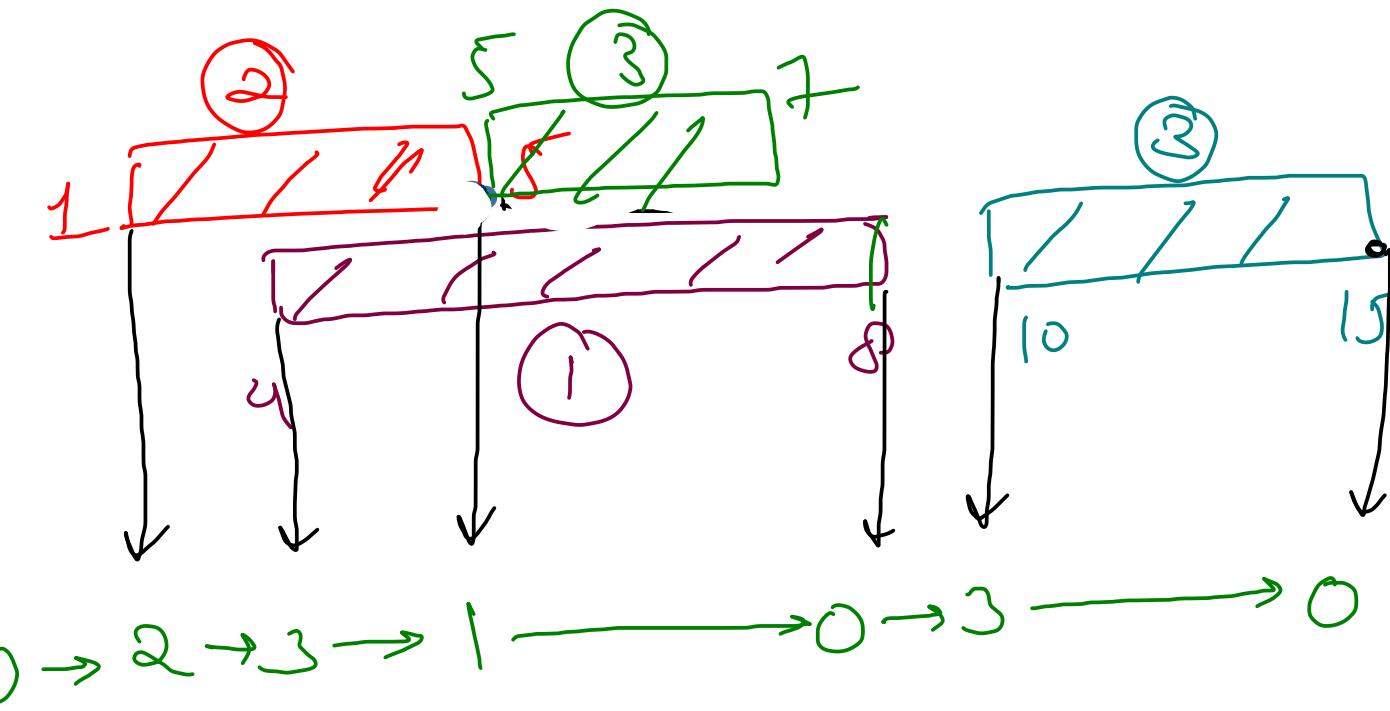
{1, 4, 8}

{3, 10, 15}

(3)

HINT
TreeMap { Red Black Tree }
→ ordered (keys)

direction → left to right , max^m capacity = ③



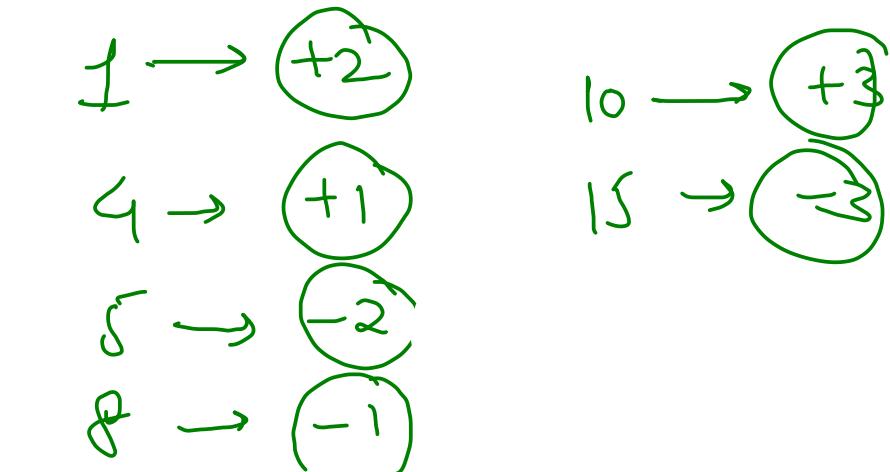
```

TreeMap<Integer, Integer> changes = new TreeMap<>();
for(int i=0; i<trips.length; i++){
    int passengers = trips[i][0];
    int start = trips[i][1];
    int end = trips[i][2];

    changes.put(start, changes.getOrDefault(start, 0) + passengers);
    changes.put(end, changes.getOrDefault(end, 0) - passengers);
}

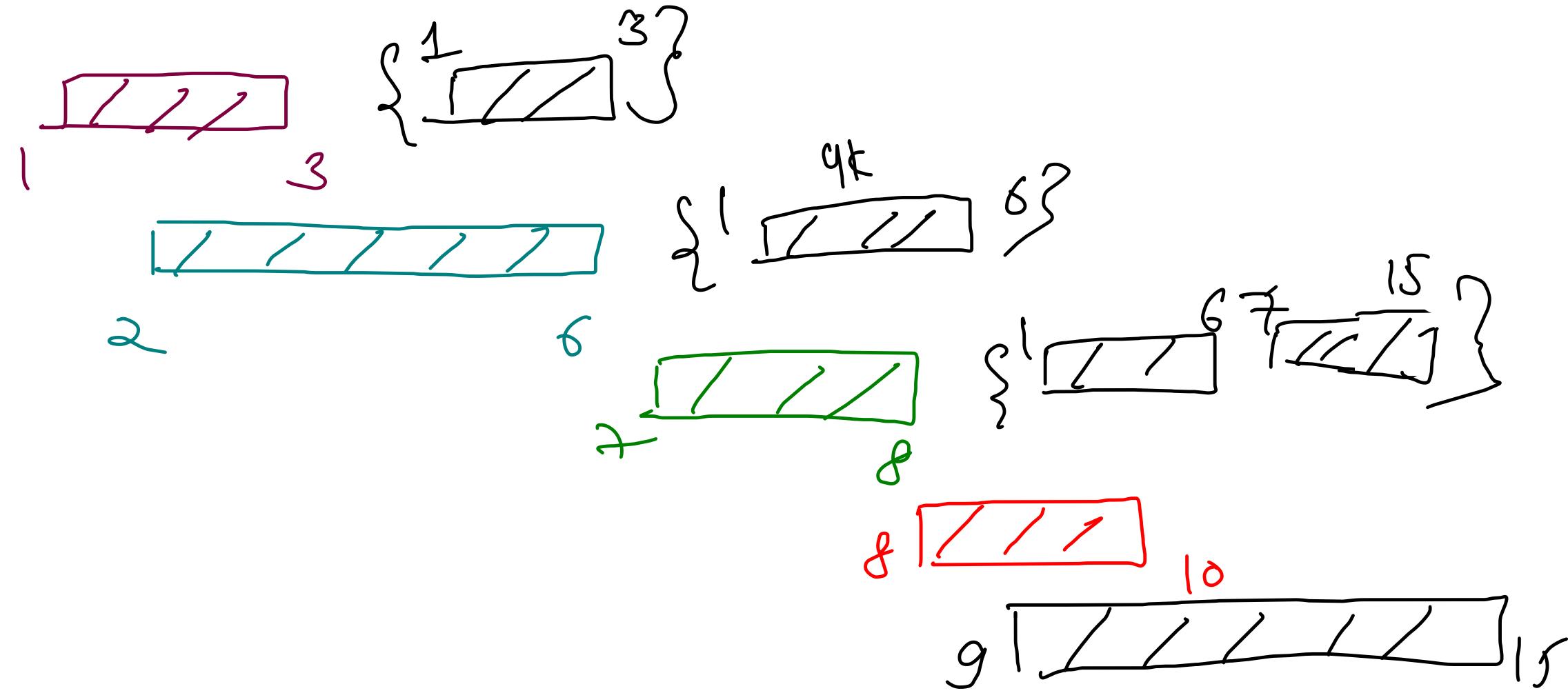
int currPassengers = 0;
for(Integer key: changes.keySet()){
    Integer val = changes.get(key);
    currPassengers += val;

    if(currPassengers > capacity) return false;
}
return true;
    
```

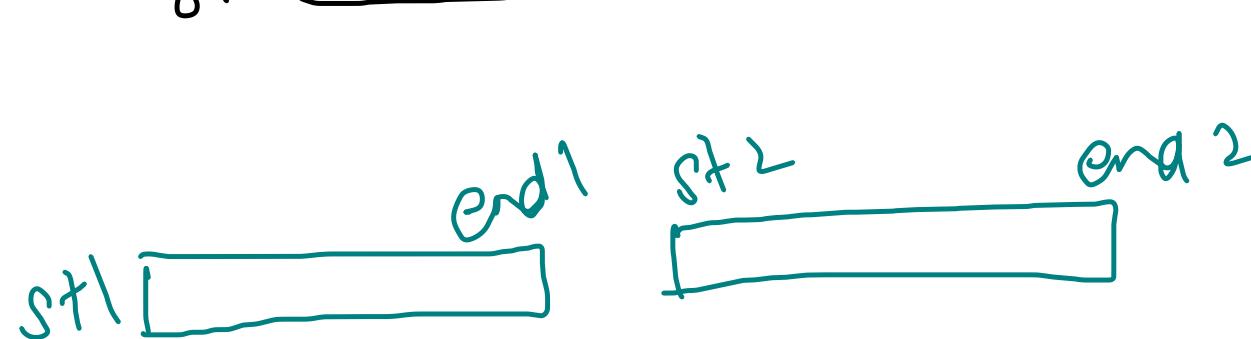
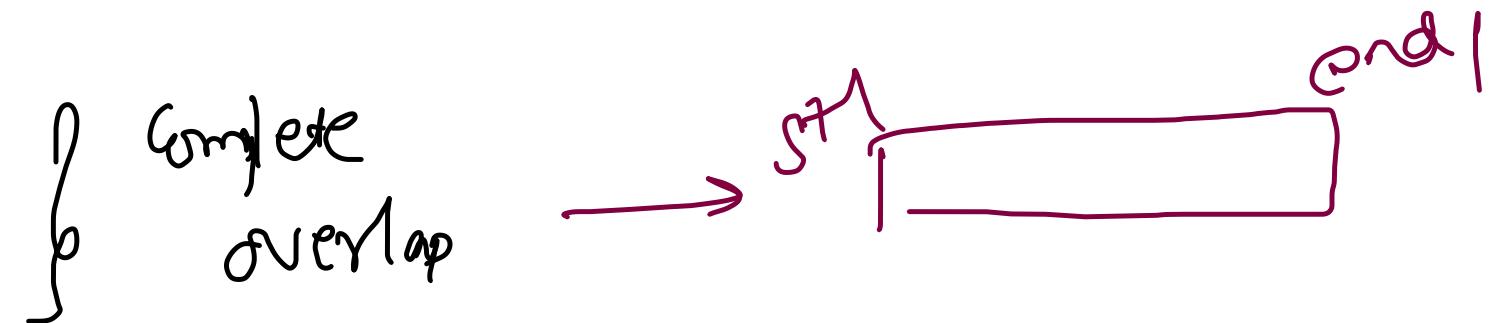


Merge Overlapping Intervals

0	1
0	1 3
1	8 10
2	7 8
3	9 15
4	2 6



Sort on basis of starting index \Rightarrow $st_1 \leq st_2$



```

public int[][] merge(int[][] intervals) {
    Arrays.sort(intervals, (a, b) -> a[0] - b[0]);
    ArrayList<int[]> merged = new ArrayList<>();
    merged.add(intervals[0]);

    for(int i=1; i<intervals.length; i++){
        int[] lastInt = merged.get(merged.size() - 1);
        int[] currInt = intervals[i];

        if(lastInt[1] >= currInt[0]){
            // merge
            lastInt[1] = Math.max(lastInt[1], currInt[1]);
        } else {
            merged.add(currInt);
        }
    }

    int[][] res = new int[merged.size()][2];
    for(int i=0; i<merged.size(); i++){
        res[i] = merged.get(i);
    }
    return res;
}

```

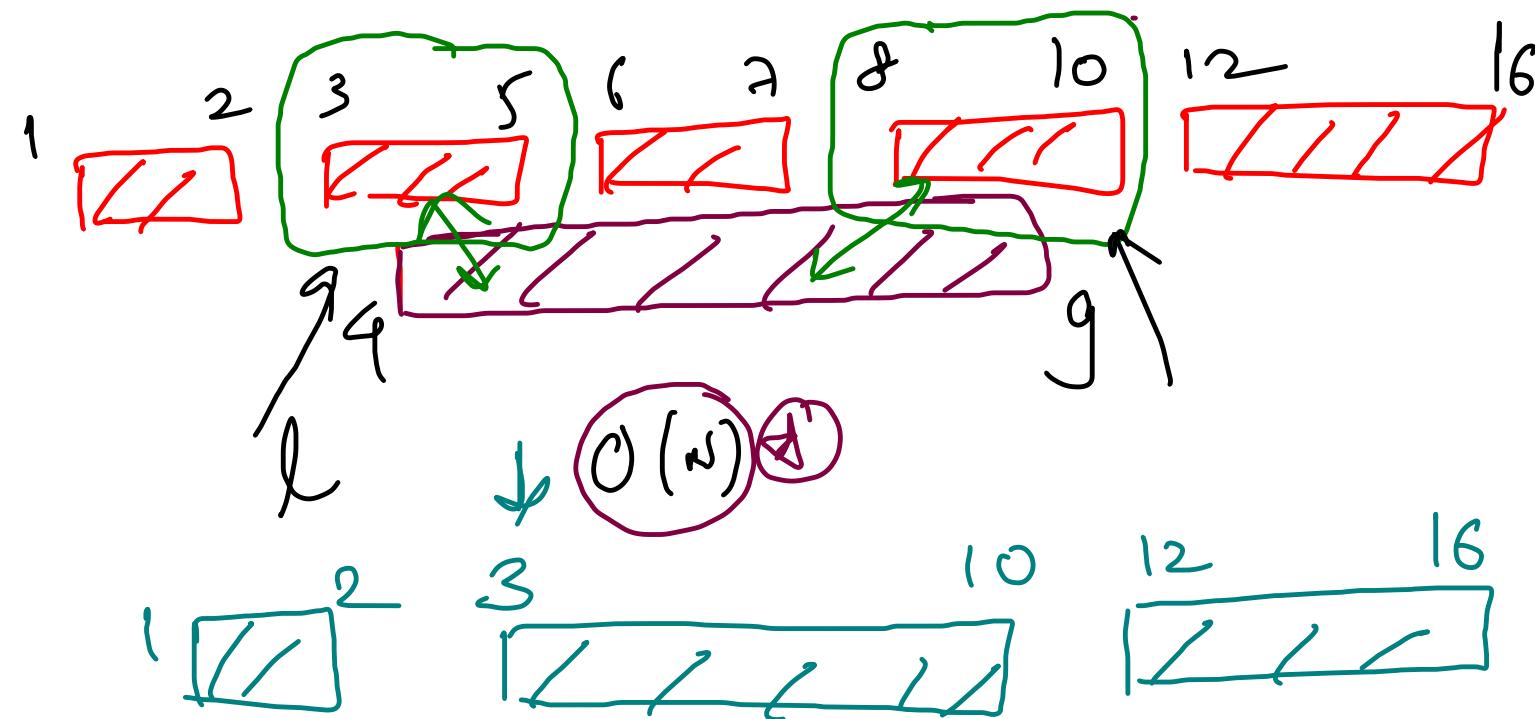
$T.C \Rightarrow O(N \log N)$

$\frac{SC}{T}$
 $O(N)$ + $O(N)$ + $O(1)$
 output Input extra space

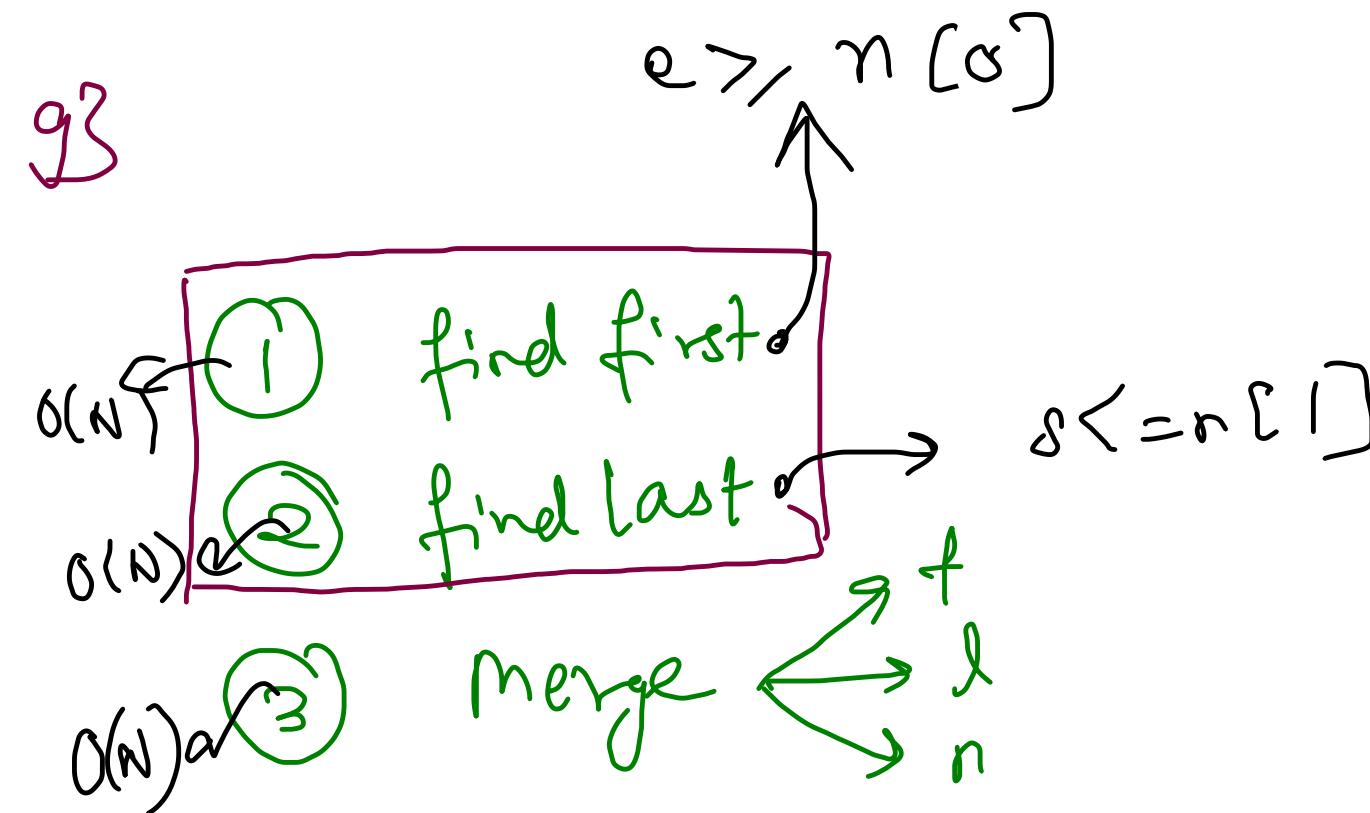
Insert Interval

`[[1, 2], [3, 5], [6, 7], [8, 10], [12, 16]]`

24, 93



```
int[] merged = new int[2];
merged[0] = Math.min(intervals[firstIdx][0], newInterval[0]);
merged[1] = Math.max(intervals[lastIdx][1], newInterval[1]);
res.add(merged);
```



$\{ \min(f[0], n[0]),$
 $\max(l[1], n[1]) \}$

```

int findFirst(int[][] intervals, int[] newInterval){
    for(int idx=0; idx<intervals.length; idx++){
        if(intervals[idx][1] >= newInterval[0]){
            return idx;
        }
    }
    return intervals.length;
}

int findLast(int[][] intervals, int[] newInterval){
    for(int idx=intervals.length-1; idx>=0; idx--){
        if(intervals[idx][0] <= newInterval[1]){
            return idx;
        }
    }
    return -1;
}

int firstIdx = findFirst(intervals, newInterval);
int lastIdx = findLast(intervals, newInterval);

ArrayList<int[]> res = new ArrayList<>();

```

```

// non merging -> firstIdx > lastIdx
if(firstIdx > lastIdx){
    for(int i=0; i<lastIdx; i++){
        res.add(intervals[i]);
    }
    res.add(newInterval);
    for(int i=firstIdx; i<intervals.length; i++){
        res.add(intervals[i]);
    }
}

```

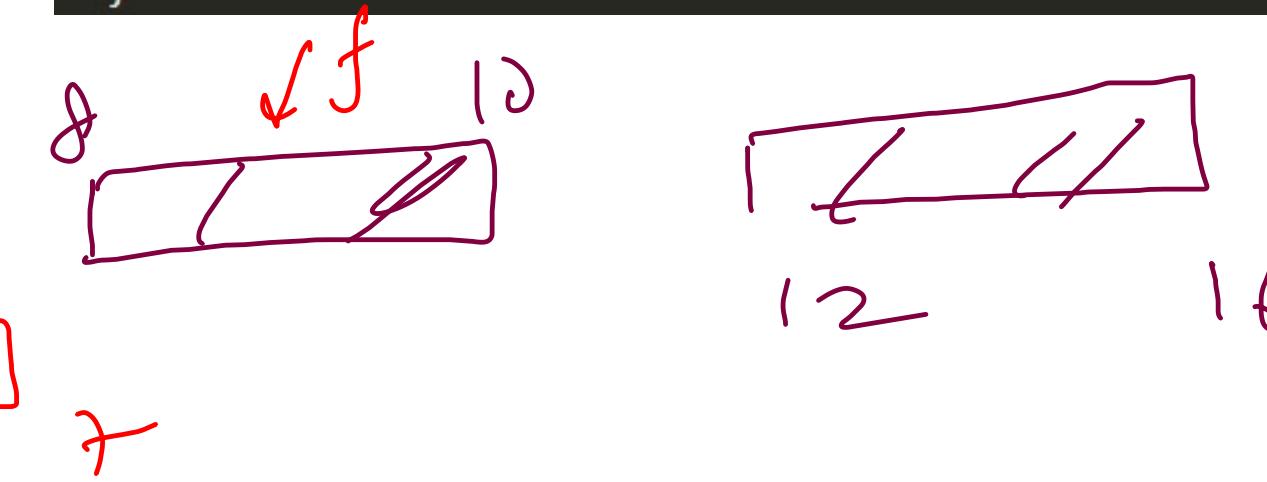
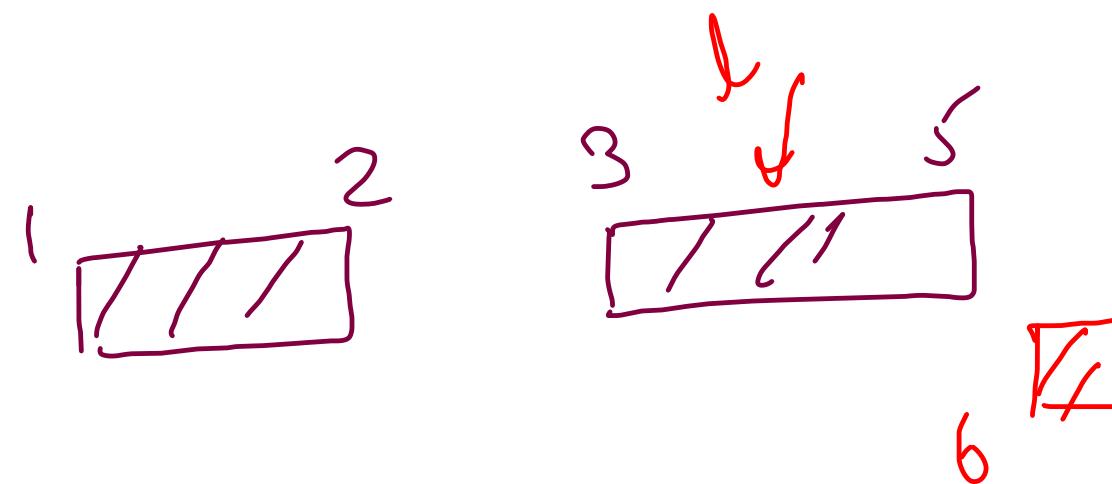
```

} else {
    for(int i=0; i<firstIdx; i++){
        res.add(intervals[i]);
    }

    int[] merged = new int[2];
    merged[0] = Math.min(intervals[firstIdx][0], newInterval[0]);
    merged[1] = Math.max(intervals[lastIdx][1], newInterval[1]);
    res.add(merged);

    for(int i=lastIdx+1; i<intervals.length; i++){
        res.add(intervals[i]);
    }
}

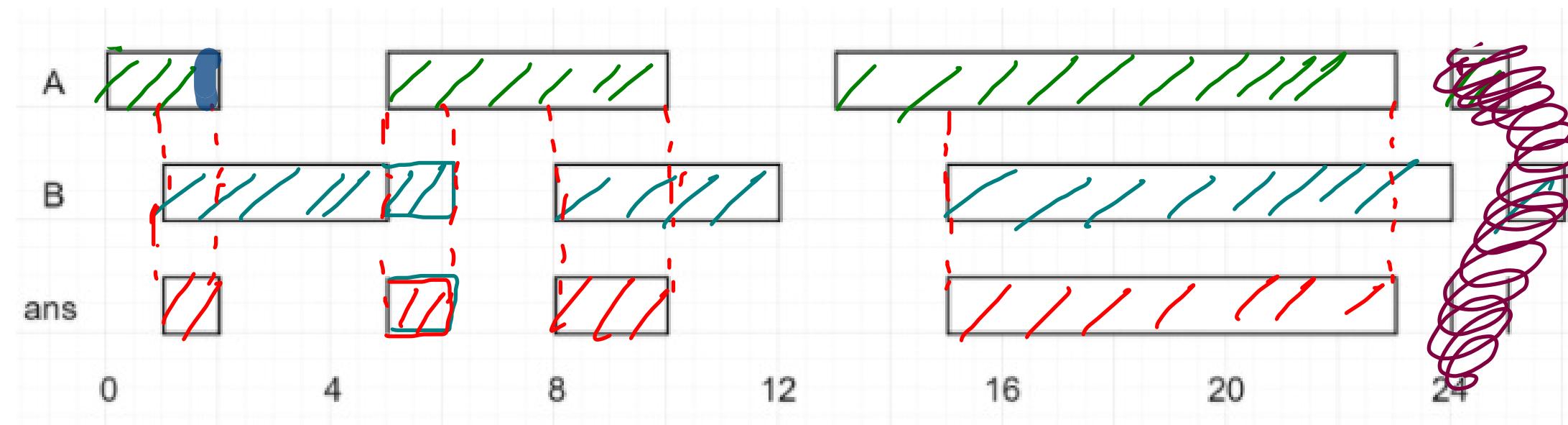
```



Intervals Intersection

non-overlapping

Each list of intervals is pairwise **disjoint** and in **sorted order**.



```
public int[][] intervalIntersection(int[][][] firstList, int[][][] secondList) {
    int firstIdx = 0, secondIdx = 0;
    ArrayList<int[]> intersect = new ArrayList<>();

    while(firstIdx < firstList.length && secondIdx < secondList.length){
        int s1 = firstList[firstIdx][0];
        int e1 = firstList[firstIdx][1];
        int s2 = secondList[secondIdx][0];
        int e2 = secondList[secondIdx][1];

        int start = Math.max(s1, s2);
        int end = Math.min(e1, e2);

        if(start <= end){
            intersect.add(new int[]{start, end});
        }

        if(e1 < e2) firstIdx++;
        else secondIdx++;
    }

    int[][] res = new int[intersect.size()][2];
    for(int i=0; i<intersect.size(); i++){
        res[i] = intersect.get(i);
    }
    return res;
}
```

- ① S&S - GitHub
- ② Notes
- ③ Recording
- ④ CB videos
- ⑤ RL R-level ②

Circular Tour {Gas Station}

{Cost, gas}

0.	1	2	3	4	5
(5, 6)	(6, 7)	(7, 4)	(8, 10)	(6, 6)	(4, 5)

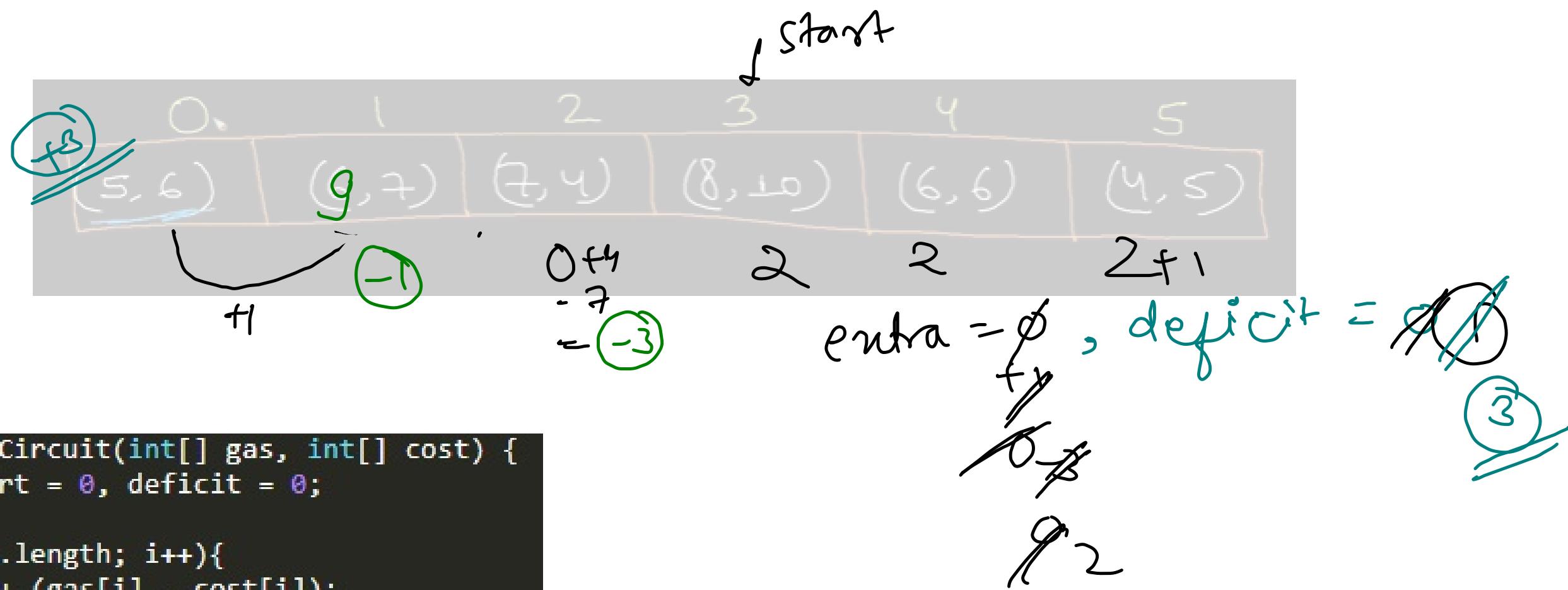
journey : 0 : $\textcircled{5,6} \xrightarrow{6-5} (6,7) \xrightarrow{8-6} (7,4) \xrightarrow{6-7 = -1} \textcircled{(8,10)}^3$

α journey : 1 : $(6,7) \xrightarrow{7-6} (7,4) \xrightarrow{5-7} \alpha$

α journey 2 : $(7,4) \xrightarrow{4-7} \alpha$

journey 3 : $(8,10) \xrightarrow{10-8} (6,7) \xrightarrow{8-6} (4,5) \xrightarrow{7-4} \textcircled{(5,6)}^+$

$\xrightarrow{9-5} (6,7) \xrightarrow{11-6} (7,4) \xrightarrow{9-7} \textcircled{(8,10)}$



```

public int canCompleteCircuit(int[] gas, int[] cost) {
    int extra = 0, start = 0, deficit = 0;

    for(int i=0; i<gas.length; i++){
        extra = extra + (gas[i] - cost[i]);

        if(extra < 0){
            start = i + 1;
            deficit -= extra;
            extra = 0;
        }
    }

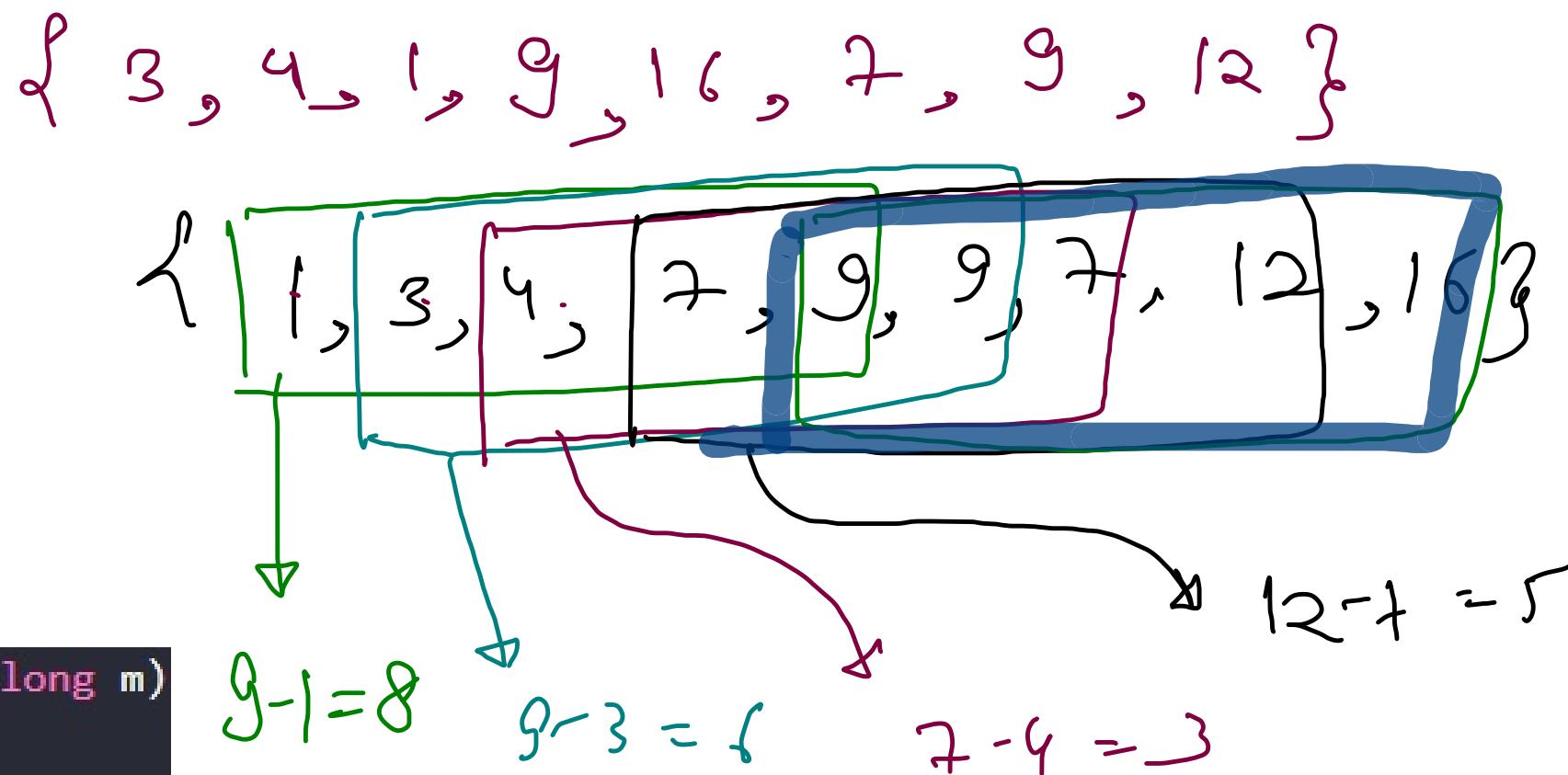
    if(extra >= deficit) return start;
    return -1;
}

```

Chocolate Distribution

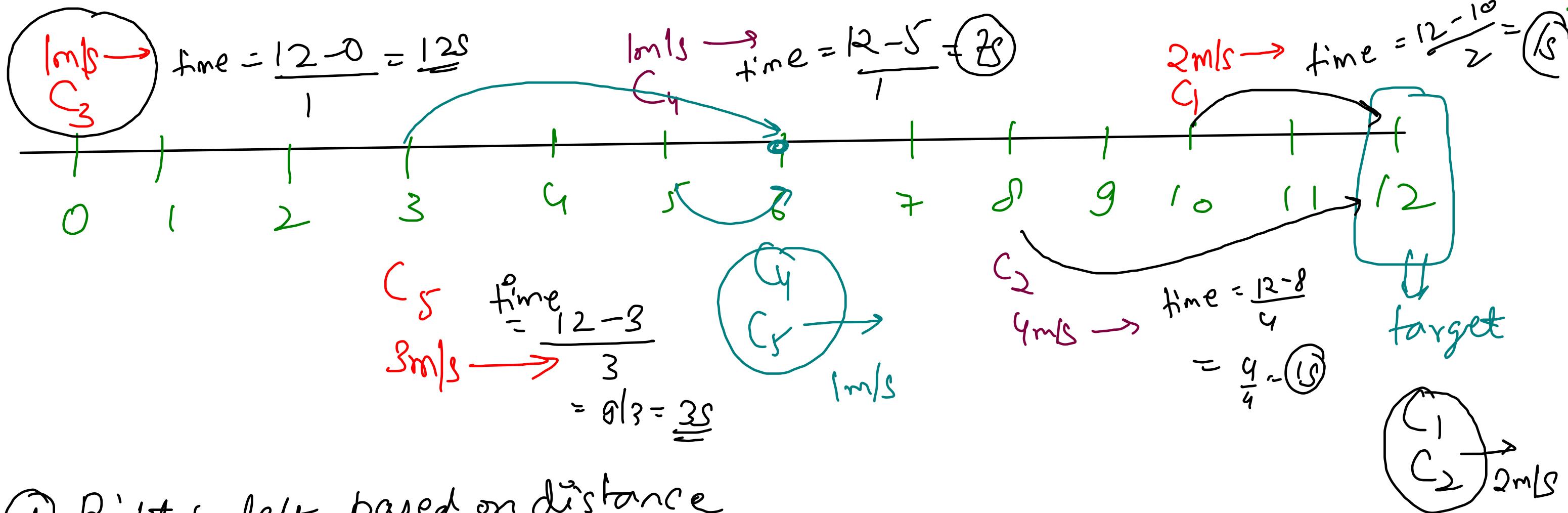
Codeforces
OA { Sorting, Searching, Greedy } 5

```
public long findMinDiff (ArrayList<Long> a, long n, long m)
{
    Collections.sort(a); } → O(n log n)
    long ans = Long.MAX_VALUE;
    for(int w=0; w<=(int)(n-m); w++){
        long Max = a.get(w + (int)m - 1);
        long Min = a.get(w);
        ans = Math.min(ans, Max - Min);
    }
    return ans;
}
```



Car Fleet

target = 12, position = [10, 8, 0, 5, 3], speed = [2, 4, 1, 1, 3]



① Right to left based on distance

② Double

③ max Time, Count of Groups

$$\text{max Time} = \cancel{12} \cancel{2} \cancel{8} / 12s$$

$$\text{Count of Group} = \cancel{1} \cancel{2} \cancel{3}$$

```

public static class Pair implements Comparable<Pair>{
    int distance;
    int speed;
    double time;

    Pair(int distance, int speed){
        this.distance = distance;
        this.speed = speed;
        this.time = ((distance * 1.0) / speed);
    }

    public int compareTo(Pair other){
        return this.distance - other.distance;
    }
}

```

break fill 11:30.

this-other → min heap
inc order

other-this → max heap
dec order

```

public int carFleet(int target, int[] position, int[] speed) {
    int n = position.length;
    Pair[] cars = new Pair[n];

    for(int i=0; i<n; i++){
        cars[i] = new Pair(target - position[i], speed[i]);
    }

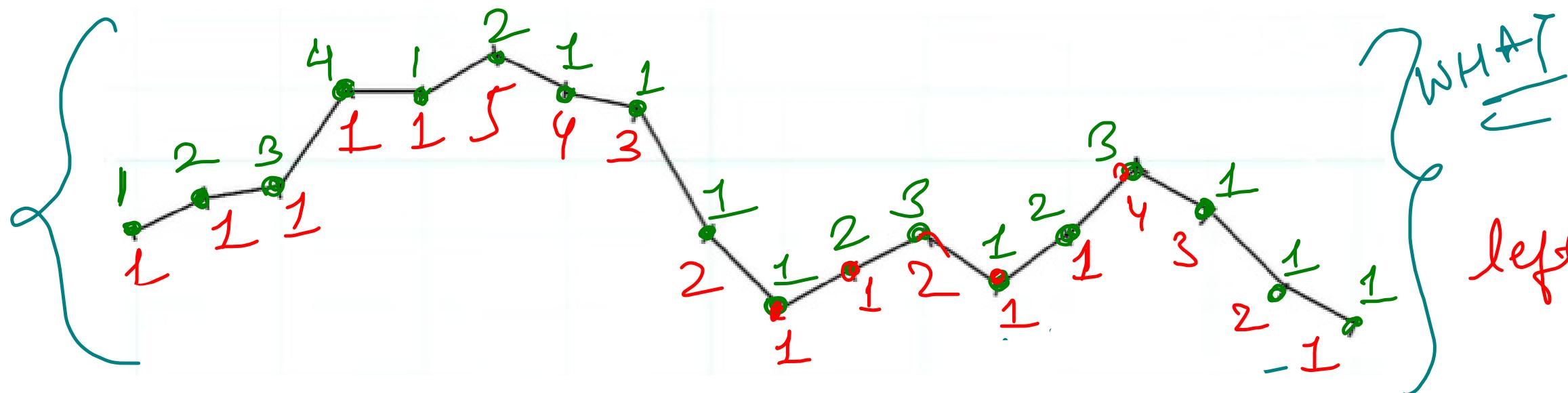
    Arrays.sort(cars); }N (N log N)

    double maxTime = 0.0;
    int countOfGroups = 0;
    for(int i=0; i<n; i++){
        if(cars[i].time > maxTime){
            countOfGroups++;
            maxTime = cars[i].time;
        }
    }
    return countOfGroups;
}

```

$O(N)$

Candy / Temple Offerings



$$1+2+3+4+1+5+4+3+2+1+2+3+1+2+4+3+2+1$$

There are n children standing in a line. Each child is assigned a rating value given in the integer array `ratings`.

You are giving candies to these children subjected to the following requirements:

- Each child must have at least one candy.
- Children with a higher rating get more candies than their neighbors.

Return the *minimum number of candies you need to have to distribute the candies to the children*.

left to right

\rightarrow left neighbour

right to left \Rightarrow
right neighbour

```
public int candy(int[] ratings) {
    int[] left = new int[ratings.length];
    int[] right = new int[ratings.length];

    left[0] = 1;
    for(int i=1; i<ratings.length; i++){
        if(ratings[i] > ratings[i - 1]){
            left[i] = left[i - 1] + 1;
        } else {
            left[i] = 1;
        }
    }

    right[ratings.length - 1] = 1;
    for(int i=ratings.length-2; i>=0; i--){
        if(ratings[i] > ratings[i + 1]){
            right[i] = right[i + 1] + 1;
        } else {
            right[i] = 1;
        }
    }

    int sum = 0;
    for(int i=0; i<ratings.length; i++){
        sum += Math.max(left[i], right[i]);
    }
    return sum;
}
```

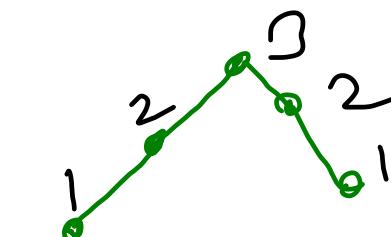
$\Rightarrow \mathcal{O}(n)$

$\Rightarrow \mathcal{O}(n)$

$\Rightarrow \mathcal{O}(n)$

$\{10, 40, 60\}$

1 2 3

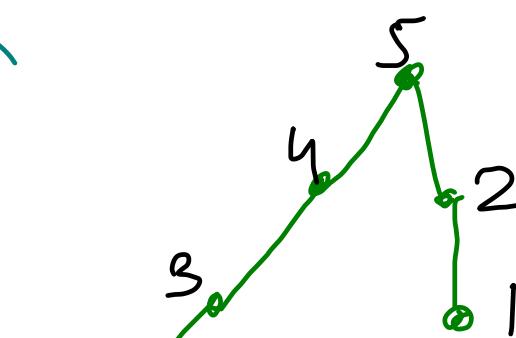


$\{10, 30, 70, 60, 20\}$

1 2 3 2 1

$\{50, 30, 20\}$

3 2 1



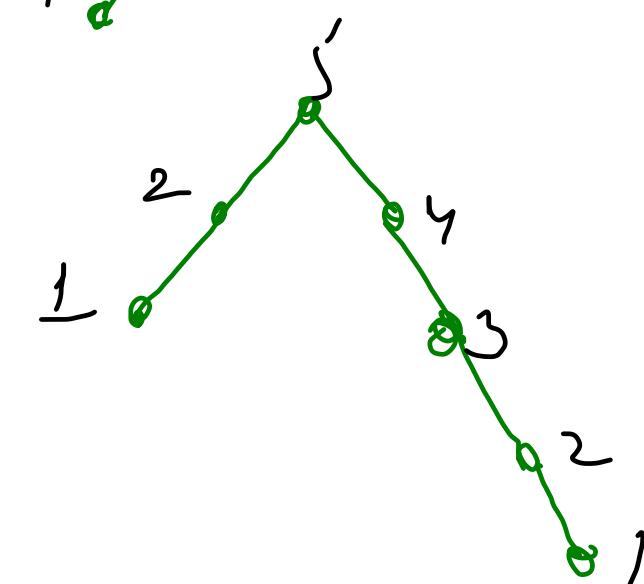
$\{10, 30, 70, 90, 60, 50, 40\}$

1 2 3 4 5 2 1

$\{20, 20, 20\}$

1 1 1

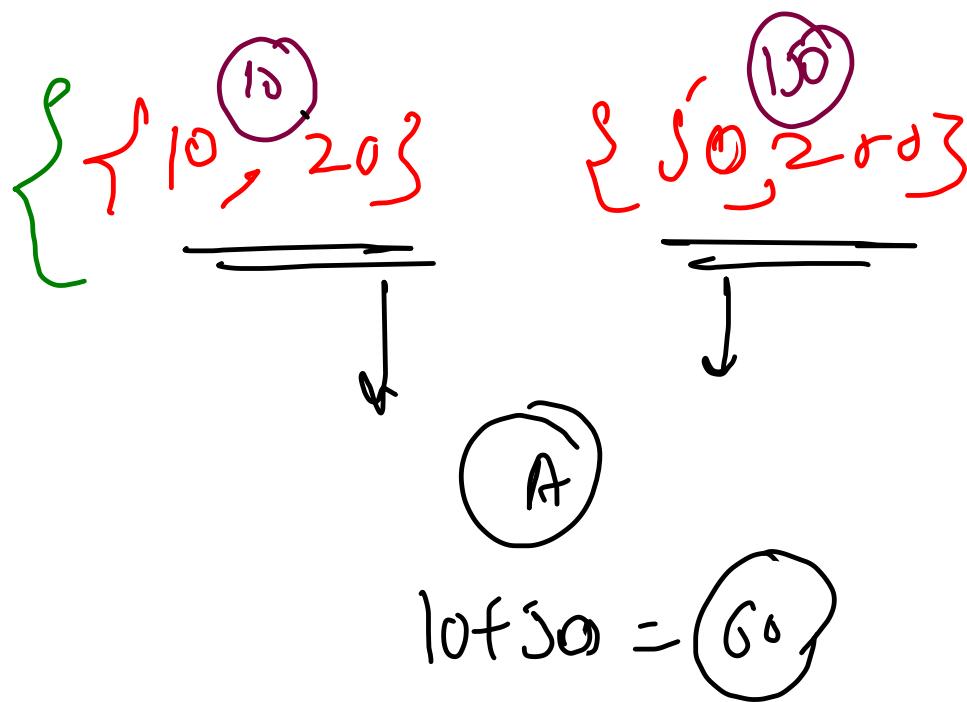
WHY



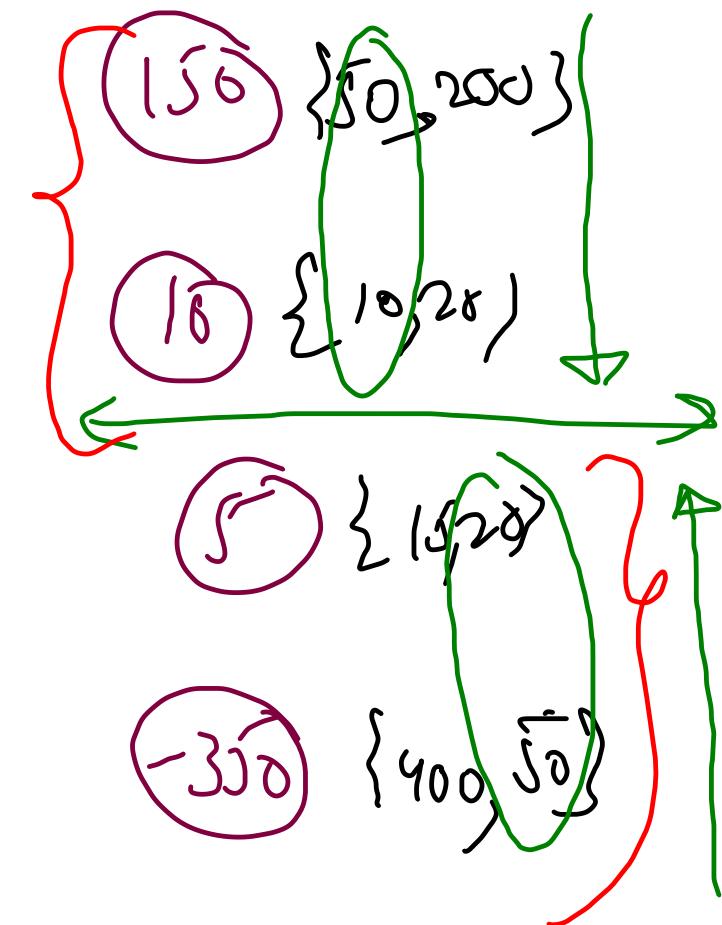
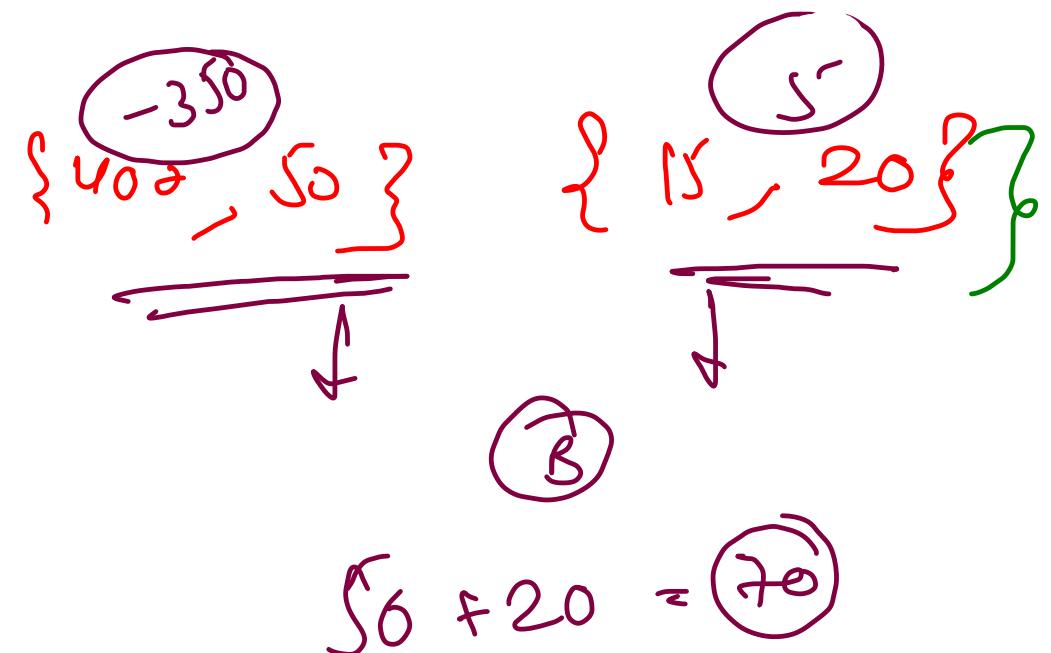
$\{80, 90, 100, 80, 60, 40, 30\}$

1 2 5 4 3 2 1

Two city Scheduling



Sort on the basis
of net gain in going
to A w.r.t B -



A company is planning to interview $2n$ people. Given the array `costs` where `costs[i] = [aCosti, bCosti]`, the cost of flying the i^{th} person to city `a` is `aCosti`, and the cost of flying the i^{th} person to city `b` is `bCosti`.

Return the *minimum cost* to fly every person to a city such that exactly n people arrive in each city.

```
public int twoCitySchedCost(int[][] costs) {
    Arrays.sort(costs, (x, y) -> ((y[1] - y[0]) - (x[1] - x[0])));
    int sum = 0;
    for(int i=0; i<costs.length/2; i++){
        sum += costs[i][0];
        sum += costs[costs.length - 1 - i][1];
    }
    return sum;
}
```

other - this

