Greedy Algorithms

Local optima choice at each step helps solving global optimum.
(min / max)

APPLICATIONS

1. Fractional Knapsack
2. Activity Selection / Overlapping Intervals / Meeting Rooms
3. Minimum Spanning Tree
   (Prims (PQ), Kruskal (DSU))
4. Huffman Encoding & Decoding / Optimal Merge Pattern
5. Single Source Shortest Path Algo {DIJKSTRA}
6. Job Sequencing
7. Problems based on SORTING.

Job Sequencing

Job Sequencing Problem

Medium Accuracy: 48.94% Submissions: 67146 Points: 4

Given a set of **N** jobs where each **job$_i$** has a deadline and profit associated with it.

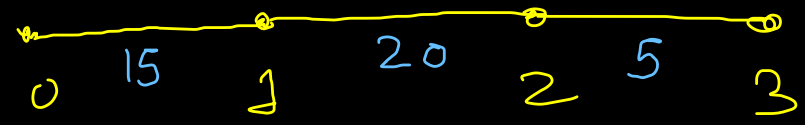
Each job takes **1** unit of time to complete and only one job can be scheduled at a time. We earn the profit associated with job if and only if the job is completed by its deadline.

Find the number of jobs done and the **maximum profit**.

Note: Jobs will be given in the form (Job$_{id}$, Deadline, Profit) associated with that Job.

(Sort as per dec order of profit)

Job	J_1	J_2	J_3	J_4	J_5
profit	20	15	10	5	1
deadline	2	2	1	3	3



Max profit = 15 + 20 + 5 = 40

We don't have to maximize the no of jobs. We just have to maximize the profit.

```java
public static class MyComparator implements Comparator<Job> {

    public int compare(Job obj1, Job obj2) {
        //decreasing order of profit
        if(obj1.profit == obj2.profit) return obj2.deadline - obj1.deadline;
        return obj2.profit - obj1.profit;
    }
}
```

```java
int[] JobScheduling(Job arr[], int n)
{
    // Your code here
    Arrays.sort(arr,new MyComparator());
    int maxDeadLine = 0;

    for(int i=0;i<n;i++) {
        maxDeadLine = Math.max(maxDeadLine,arr[i].deadline);
    }

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

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

    int[] res = new int[2];
    res[0] = jobsAllocated;
    res[1] = maxProfit;

    return res;
}
}
```

TC

$O(d * N + N \log_2 N)$

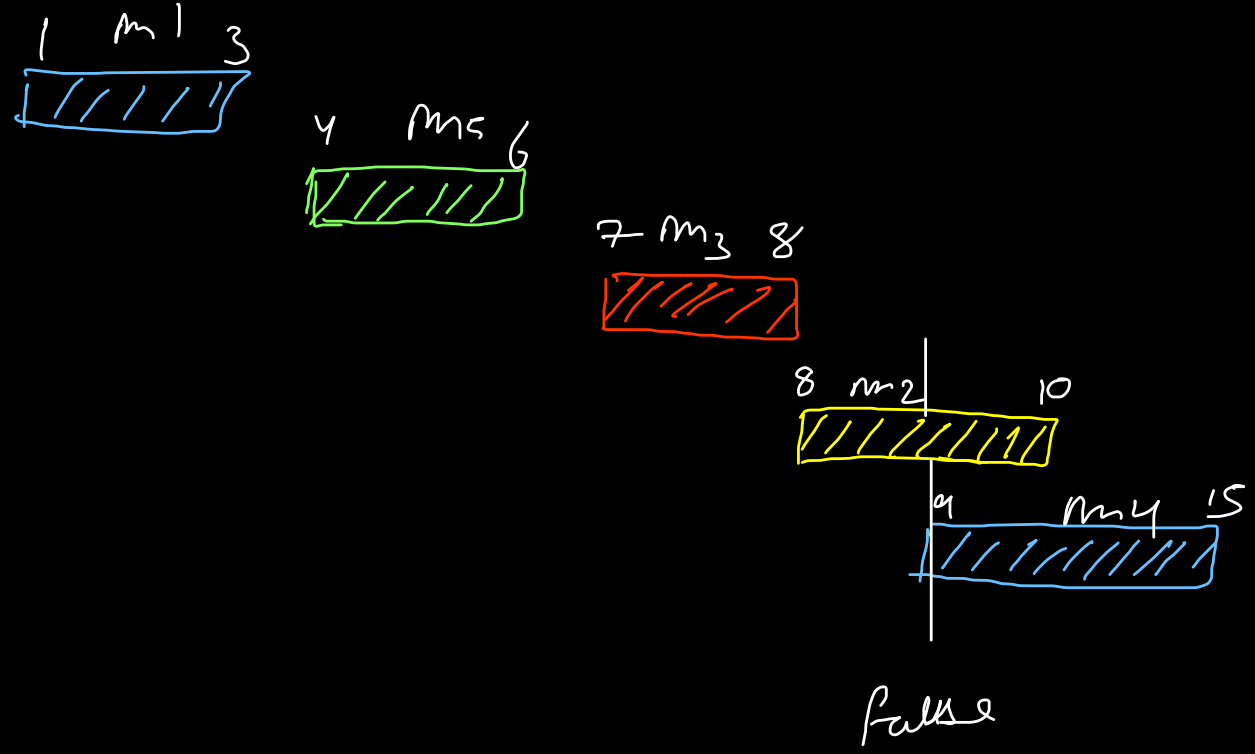
↓

Max deadline (no of the loop)

Meeting Rooms - I

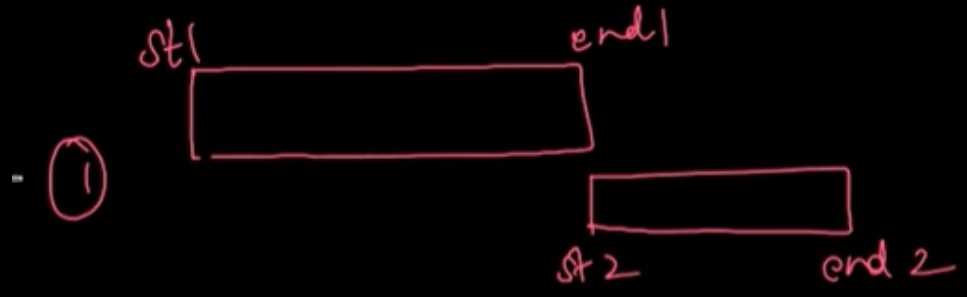
one room ⟹ one meeting at a time.

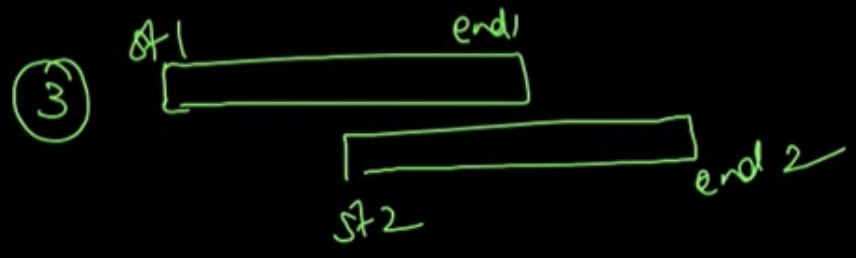
	st	et
m_1	1	3
m_2	8	10
m_3	7	8
m_4	9	15
m_5	4	6



False

Sorting on end time basis can give 3 types of cases -





if end 1 > start 2

return false

① Sort on the basis of End Time

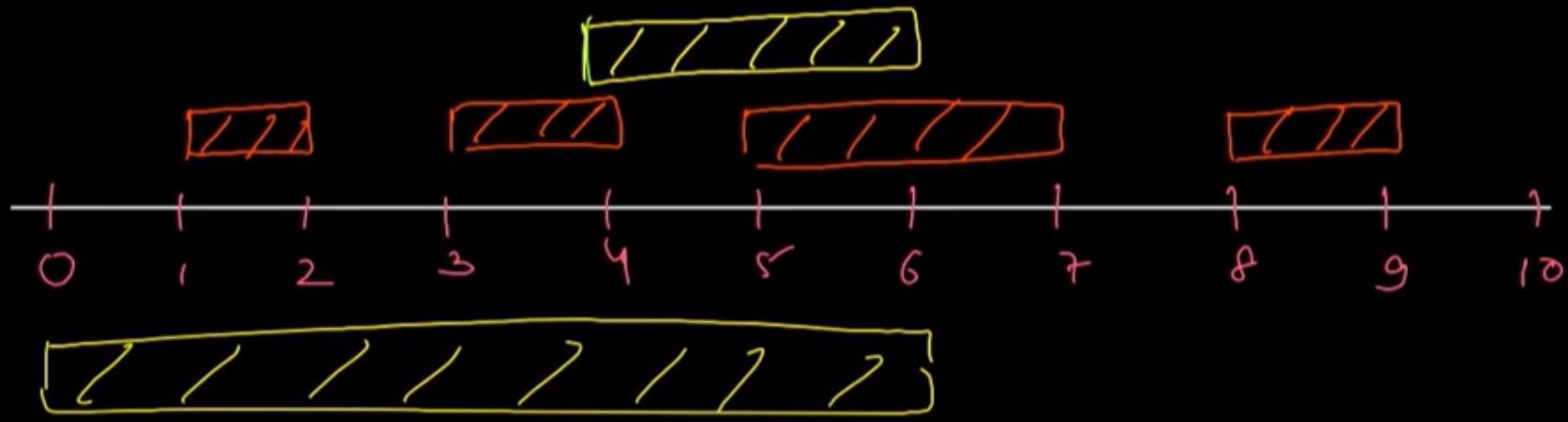
② Check overlapping intervals -

```java
public boolean canAttendMeetings(List<Interval> intervals) {
    // Write your code here
    Collections.sort(intervals,new MyComparator());

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

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

    return true;
}
```

$TC = O(N \log_2 N)$

Meeting in 1 Room {Max No of Non-Overlapping Intervals}

S[] =	1	0	3	8	5	8 4
F[] =	2	6	4	9	7	9 6
	1	2	3	4	5	6



```java
public static int maxMeetings(int start[], int end[], int n)
{
    // add your code here
    Interval[] intervals = new Interval[n];
    for(int i=0;i<n;i++) {
        intervals[i] = new Interval(start[i],end[i]);
    }

    Arrays.sort(intervals,new MyComparator());

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

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

    return count;
}
```

```java
public static class Interval {
    int start;
    int end;

    Interval() {}

    Interval(int start, int end) {
        this.start = start;
        this.end = end;
    }
}

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

435. Non-overlapping Intervals

Medium 3971 117 Add to List Share

Given an array of intervals `intervals` where `intervals[i] = [start_i, end_i]`, return *the minimum number of intervals you need to remove to make the rest of the intervals non-overlapping.*

```java
public int eraseOverlapIntervals(int[][] mat) {
    int n = mat.length;
    Interval[] intervals = new Interval[n];
    for(int i=0;i<n;i++) {
        intervals[i] = new Interval(mat[i][0],mat[i][1]);
    }

    Arrays.sort(intervals,new MyComparator());

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

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

    return n - count;
}
```

```java
class Solution {

    public static class Interval {
        int start;
        int end;

        Interval() {}

        Interval(int start, int end) {
            this.start = start;
            this.end = end;
        }
    }

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

646. Maximum Length of Pair Chain

Medium 2243 102 Add to List Share

You are given an array of `n` pairs `pairs` where `pairs[i] = [left_i, right_i]` and `left_i < right_i`.

A pair `p2 = [c, d]` **follows** a pair `p1 = [a, b]` if `b < c`. A **chain** of pairs can be formed in this fashion.

Return *the length longest chain which can be formed.*

You do not need to use up all the given intervals. You can select pairs in any order.

```java
public static class Interval {
    int start;
    int end;

    Interval() {}

    Interval(int start, int end) {
        this.start = start;
        this.end = end;
    }
}

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

 public int findLongestChain(int[][] pairs) {
     int n = pairs.length;
     Interval[] intervals = new Interval[n];
     for(int i=0;i<n;i++) {
         intervals[i] = new Interval(pairs[i][0],pairs[i][1]);
     }

     Arrays.sort(intervals,new MyComparator());

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

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

     return count;
 }
}
```

!!

Same ques as N Meetings in 1 room. (Max no of Non-Overlapping intervals)

= → non overlapping

≠ → overlapping

considering 1 intervals with end 1 = Start 2

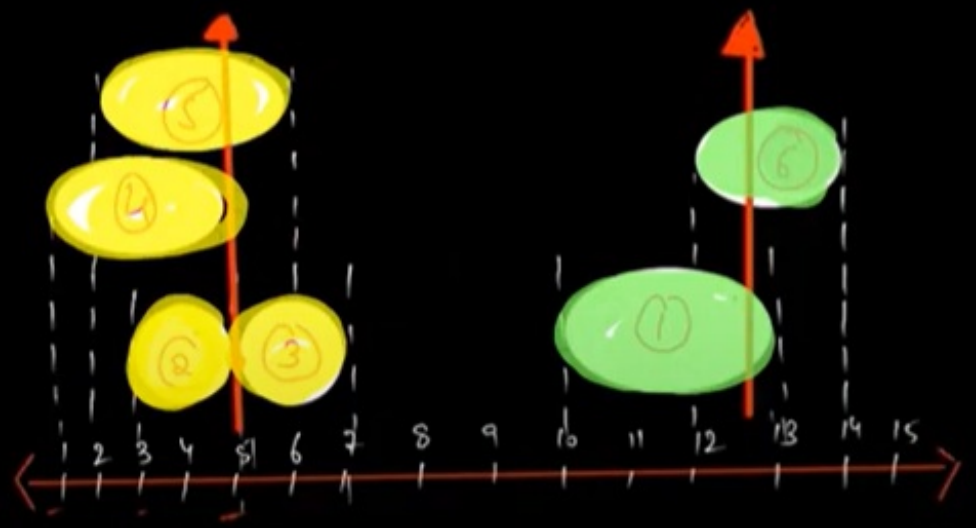
452. Minimum Number of Arrows to Burst Balloons

Medium 3427 102 Add to List Share

There are some spherical balloons taped onto a flat wall that represents the XY-plane. The balloons are represented as a 2D integer array `points` where `points[i] = [x_start, x_end]` denotes a balloon whose **horizontal diameter** stretches between `x_start` and `x_end`. You do not know the exact y-coordinates of the balloons.

Arrows can be shot up **directly vertically** (in the positive y-direction) from different points along the x-axis. A balloon with `x_start` and `x_end` is **burst** by an arrow shot at `x` if `x_start <= x <= x_end`. There is **no limit** to the number of arrows that can be shot. A shot arrow keeps traveling up infinitely, bursting any balloons in its path.

Given the array `points`, return *the **minimum** number of arrows that must be shot to burst all balloons.*



This is same as N meetings in a room.

```java
public static class Interval {
    int start;
    int end;

    Interval() {}

    Interval(int start, int end) {
        this.start = start;
        this.end = end;
    }
}

public static class MyComparator implements Comparator<Interval> {
    public int compare(Interval obj1,Interval obj2) {
        if(obj1.end >= obj2.end) return 1;
        return -1;
    }
}
```

earlier we were returning

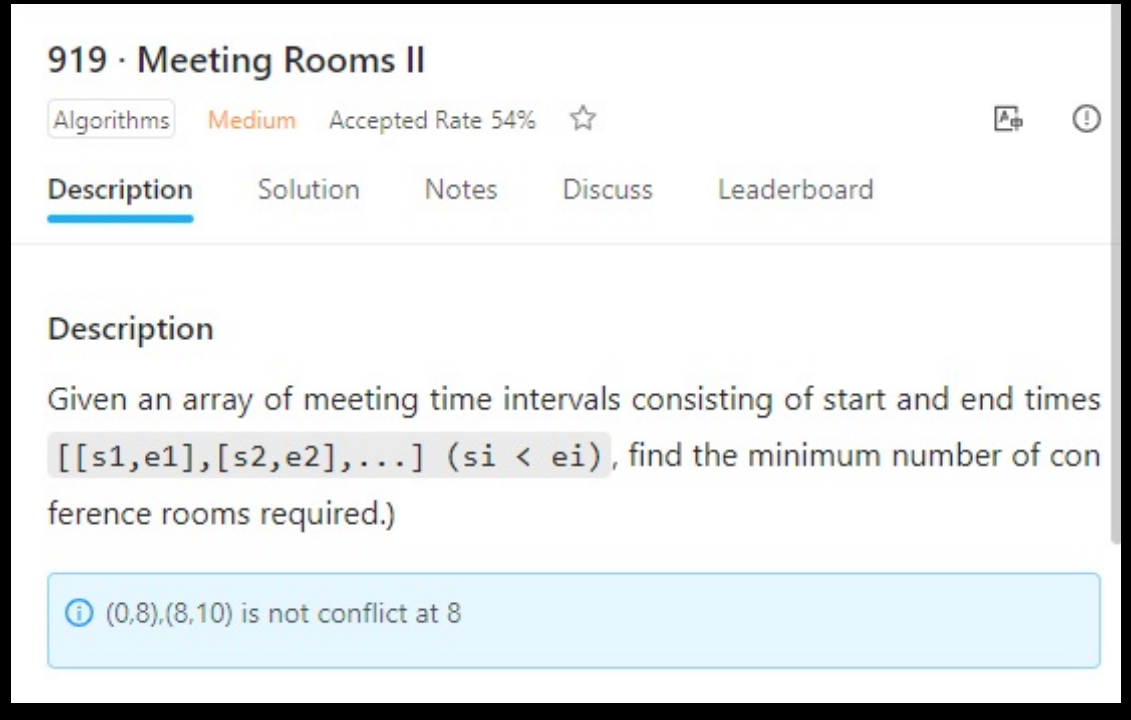
obj1.end - obj2.end

(say $-\infty$) (say ∞)

$-\infty - \infty = -2\infty$ ⇒ Very big no

```java
public int findMinArrowShots(int[][] points) {
    int n = points.length;
    Interval[] intervals = new Interval[n];
    for(int i=0;i<n;i++) {
        intervals[i] = new Interval(points[i][0],points[i][1]);
    }

    Arrays.sort(intervals,new MyComparator());

    //last interval's ending time
    int limit = intervals[0].end;
    int count = 1; //count of non overlapping intervals


    for(int i=1;i<intervals.length;i++) {
        if((int)limit < intervals[i].start) {
            count++;
            limit = intervals[i].end;
        }
    }

    return count;
}
}
```

Meeting Rooms II



919 · Meeting Rooms II

Algorithms Medium Accepted Rate 54%

Description Solution Notes Discuss Leaderboard

Description

Given an array of meeting time intervals consisting of start and end times `[[s1,e1],[s2,e2],...] (si < ei)`, find the minimum number of con ference rooms required.)

(0,8),(8,10) is not conflict at 8

In meeting rooms - 1 category, we were always given 1 room and we had to schedule down meetings in it. Here, we have to schedule all the meetings. Hence, we need to find the min no. of rooms we need to do so.

1 - 3

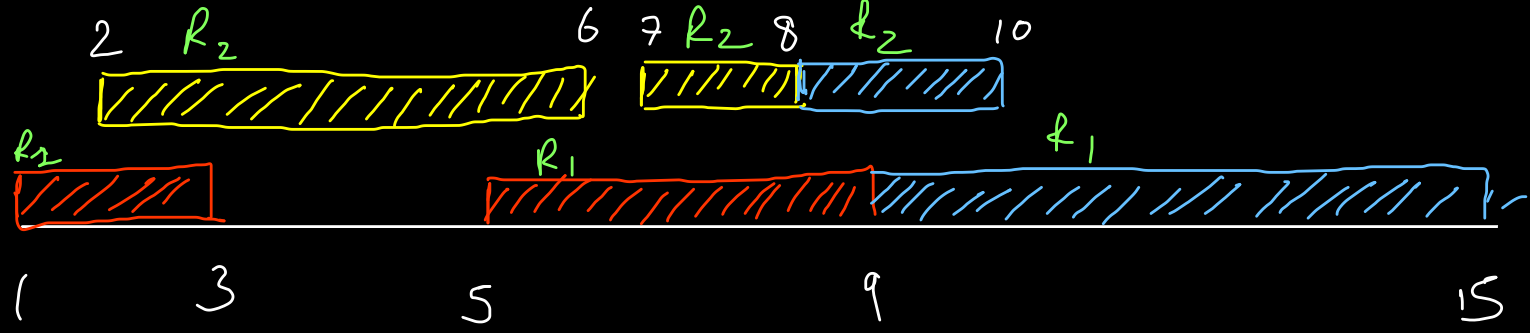
8 - 10

7 - 8

9 - 15

2 - 6

5 - 9



Ans = 2

✓ 1 - 3

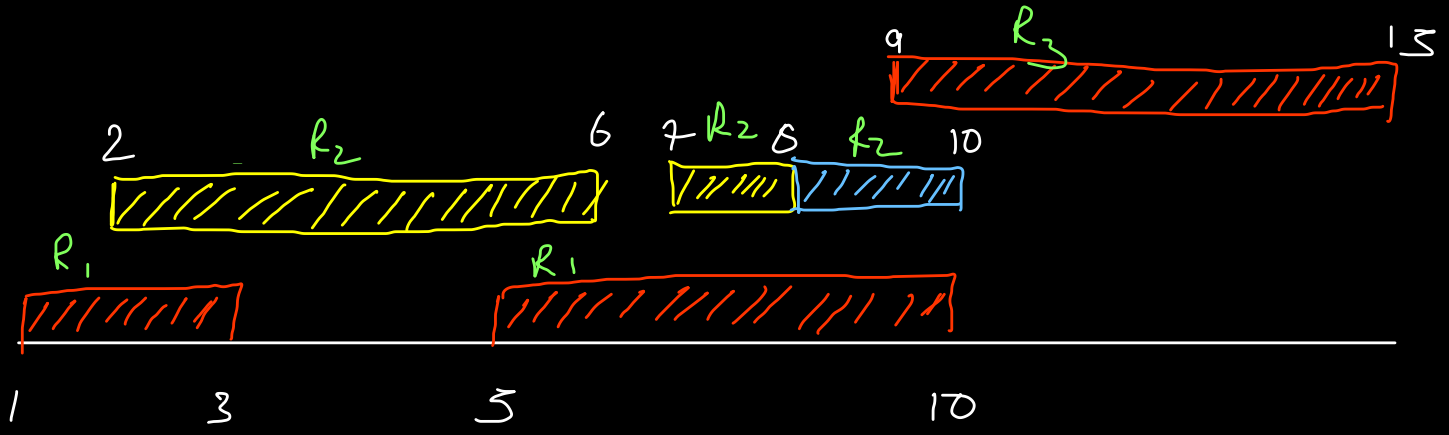
✓ 8 - 10

✓ 7 - 8

9 - 15

✓ 2 - 6

✓ 5 - 10



Ans = 3

1 - 3

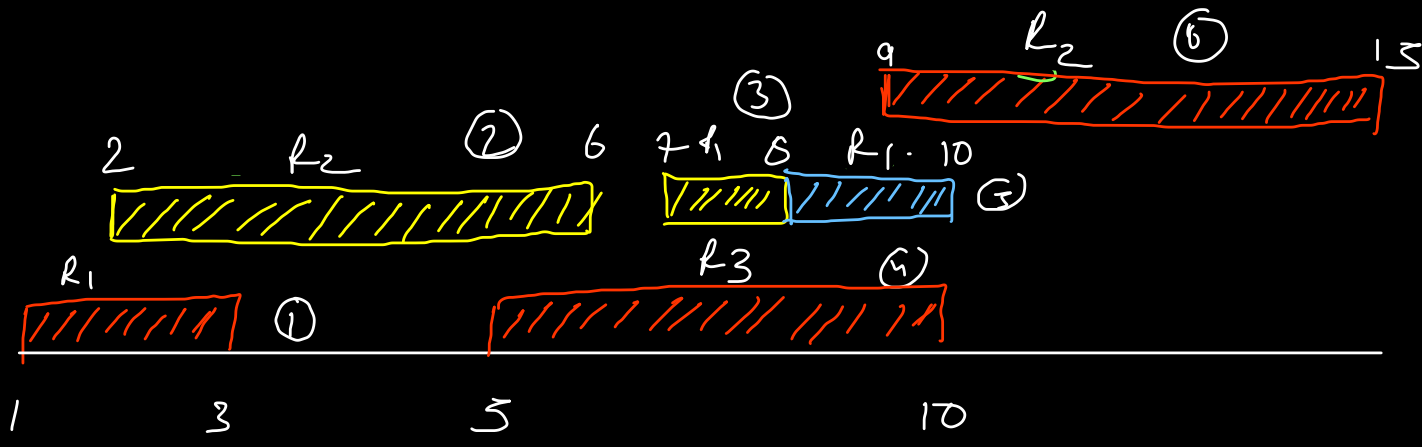
8 - 10

7 - 8

9 - 15

2 - 6

5 - 10



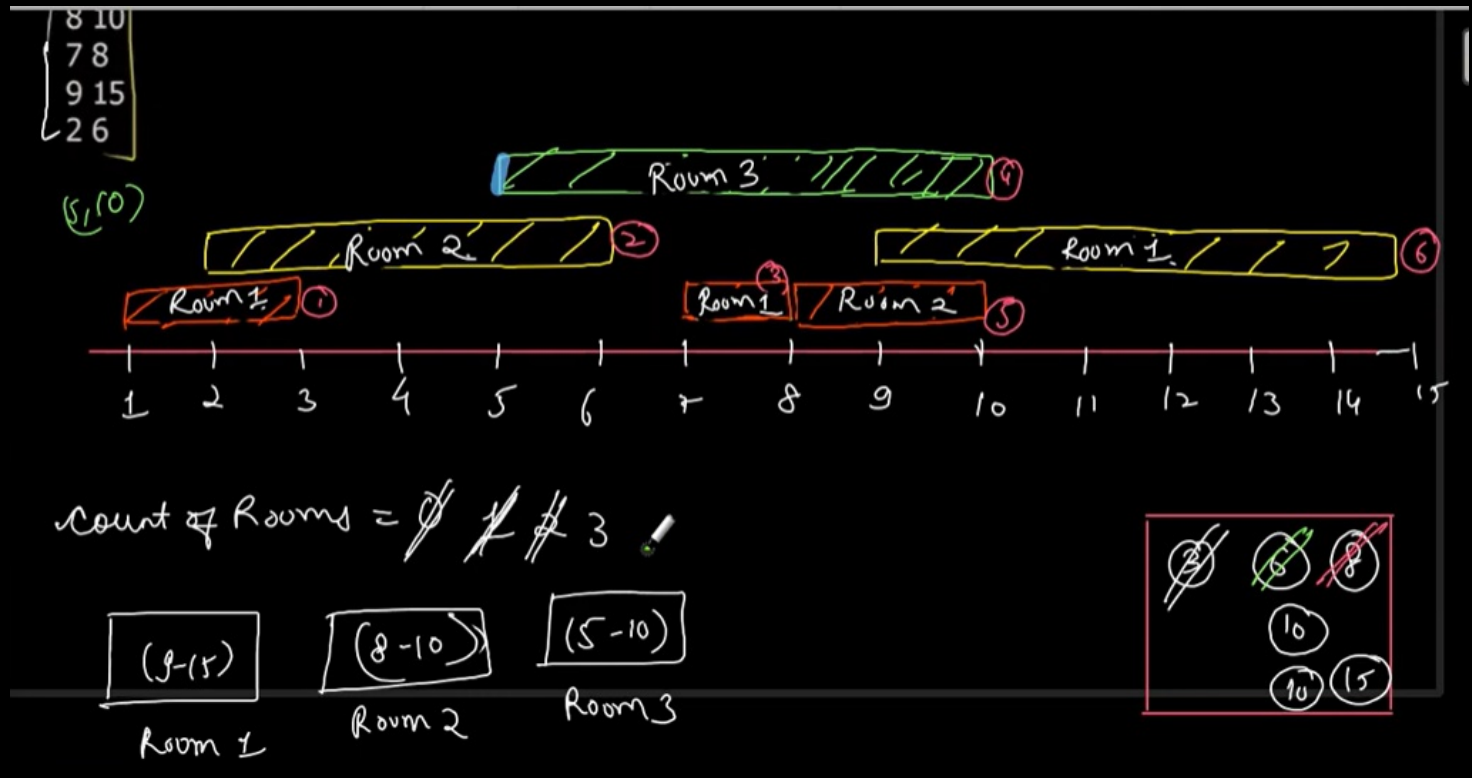
Ans = 3

① Sort on the basis of Ending Time.

A greedy choice here is to allocate that room which got empty last.

Count of Rooms = 0 1 2 3

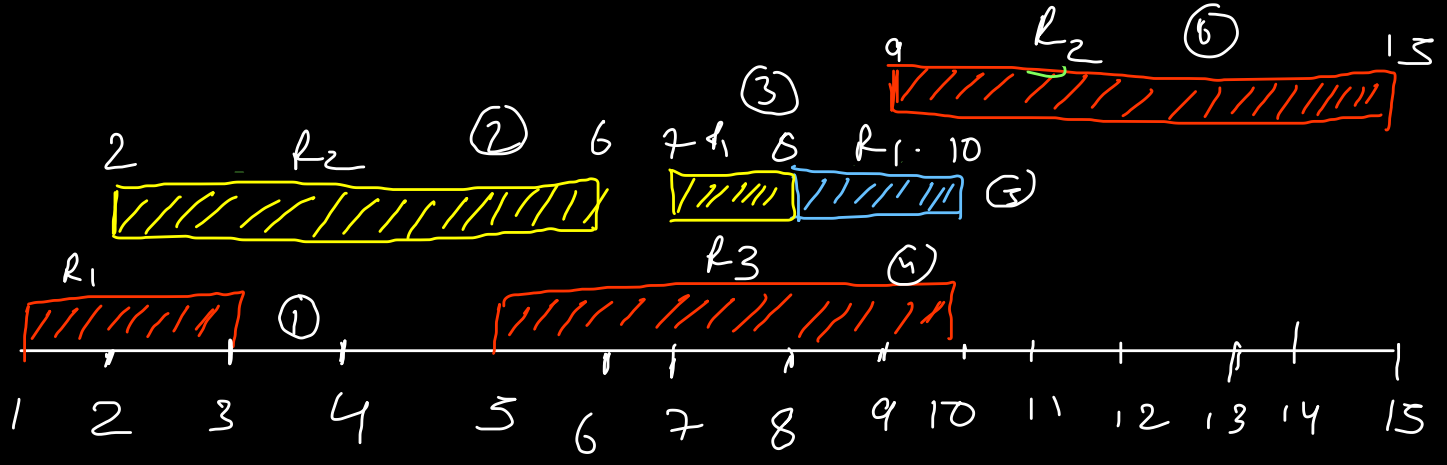
(8-10)	(9-15)	(5-10)
Room 1	Room 2	Room 3



This approach is wrong (allocating 1st empty room)

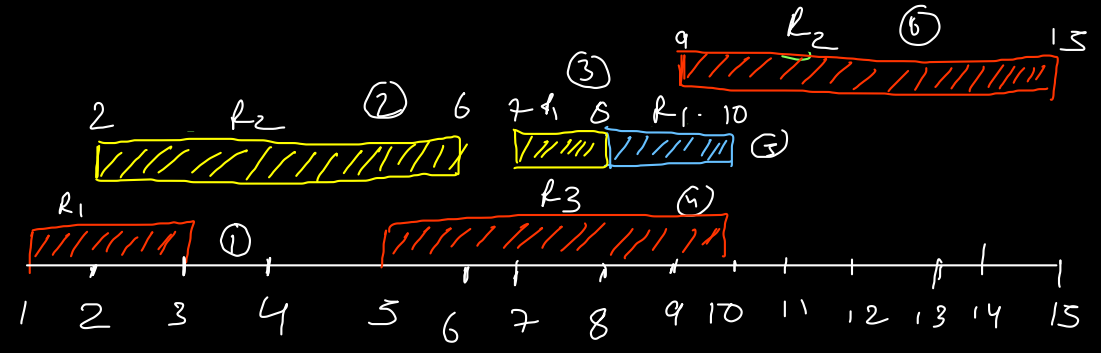
Optimized Approach

We have to look only at the transition pts (starting/ending of a meeting)



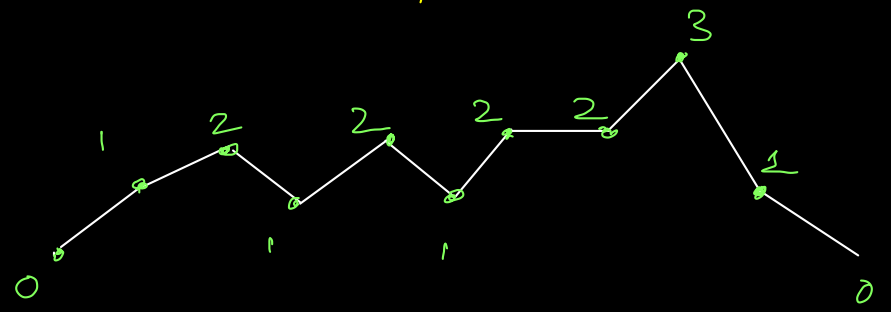
0 → 1 → 2 → 1 ——→ 2 → 1 → 2 → 2 → 3 → 1 ——→ 0

max = 3 ans



0 → 1 → 2 → 1 → 2 → 1 → 2 → 2 → 3 → 1 → 0

max = 3 ans



inc 100ms

if stpt & dec if endpt

1 2 5 7 8 9

3 6 8 10 10 15

room = 0 1 2 1 2 3

maxRoom = 0 1 2 3

Sort the 2 arrays (starting & ending time of meeting) seperately.

Apply 2 pointer similar to merge sorted arrays

* Note : After sorting the arrays, it is not necessary that start time at i^{th} idx & end time at i^{th} idx will be of same meeting.

Meeting Rooms II

```java
public int minMeetingRooms(List<Interval> intervals) {
    // Write your code here

    ArrayList<Integer> start = new ArrayList<>();
    ArrayList<Integer> end = new ArrayList<>();

    for(int i=0;i<intervals.size();i++) {
        start.add(intervals.get(i).start);
    }

    for(int i=0;i<intervals.size();i++) {
        end.add(intervals.get(i).end);
    }

    Collections.sort(start);
    Collections.sort(end);

    int rooms = 0, maxRooms = 0, i = 0, j = 0;

    while(i < start.size()) {
        if(start.get(i) < end.get(j)) {
            i++;
            rooms++;
            maxRooms = Math.max(rooms,maxRooms);
        } else if(start.get(i) > end.get(j)) {
            j++;
            rooms--;
        } else {
            i++;
            j++;
        }
    }

    return maxRooms;
}
```

N

NlogN

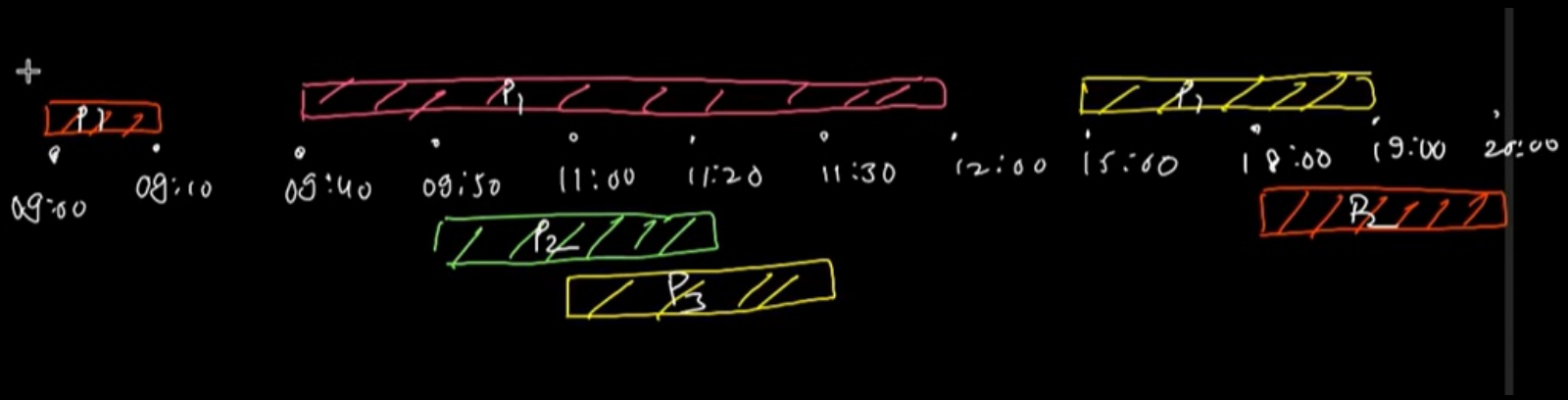
2N

Minimum Platform

Minimum Platforms

Medium Accuracy: 46.78% Submissions: 100k+ Points: 4

Given arrival and departure times of all trains that reach a railway station. Find the minimum number of platforms required for the railway station so that no train is kept waiting.
Consider that all the trains arrive on the same day and leave on the same day. Arrival and departure time can never be the same for a train but we can have arrival time of one train equal to departure time of the other. At any given instance of time, same platform can not be used for both departure of a train and arrival of another train. In such cases, we need different platforms.



```java
class Solution
{
    //Function to find the minimum number of platforms required at the
    //railway station such that no train waits.
    static int findPlatform(int arr[], int dept[], int n)
    {
        // add your code here

        Arrays.sort(arr);
        Arrays.sort(dept);

        int platforms = 0, maxPlatforms = 0, i = 0, j = 0;

        while(i < arr.length) {
            if(arr[i] <= dept[j]) {
                i++;
                platforms++;
            } else if(arr[i] > dept[j]) {
                j++;
                platforms--;
            }

          maxPlatforms = Math.max(platforms,maxPlatforms);

        }

        return maxPlatforms;
    }

}
```

1094. Car Pooling

Medium 3166 69 Add to List Share

There is a car with `capacity` empty seats. The vehicle only drives east (i.e., it cannot turn around and drive west).

You are given the integer `capacity` and an array `trips` where `trips[i] = [numPassengers_i, from_i, to_i]` indicates that the i^{th} trip has `numPassengers_i` passengers and the locations to pick them up and drop them off are `from_i` and `to_i` respectively. The locations are given as the number of kilometers due east from the car's initial location.

Return `true` *if it is possible to pick up and drop off all passengers for all the given trips, or* `false` *otherwise.*

{2, 1, 5}

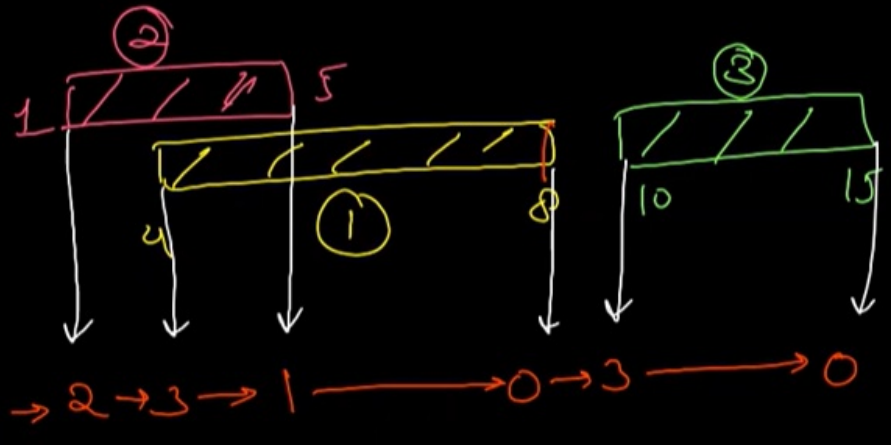
{1, 4, 8}

{3, 10, 15}

(3)

TreeMap

direction = left to right , maximum capacity = (3)



0 → 2 → 3 → 1 → 0 → 3 → 0

Self Balancing

BST { Red Black Tree }

{1, 4, 8, }

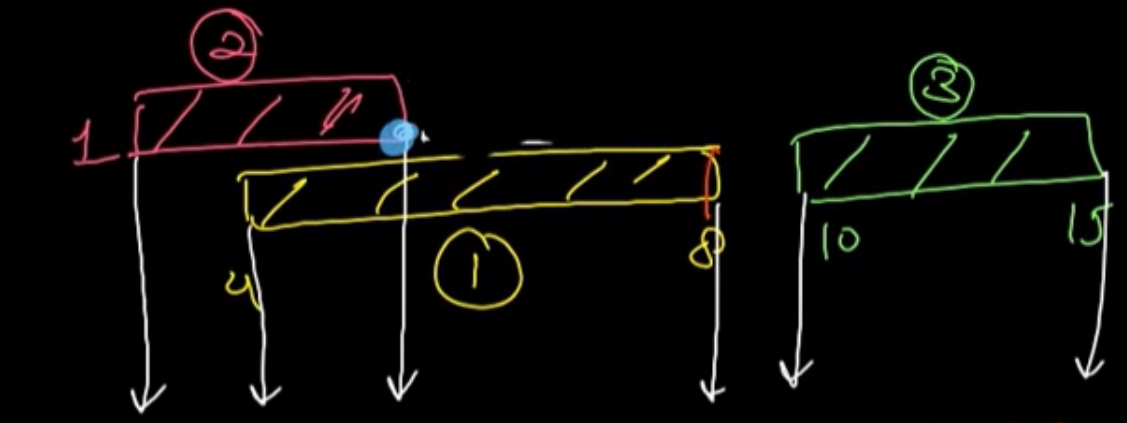
{3, 10, 15 }

(3)

HINT

TreeMap { Red Black Tree }

Ordered



1 → (+2)

4 → (+1)

8 → (-1)

10 → (+3)

15 → (-1)

```java
class Solution {
    public boolean carPooling(int[][] trips, int capacity) {
        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;
    }
}
```

TC: $O(2N \log_2 N)$

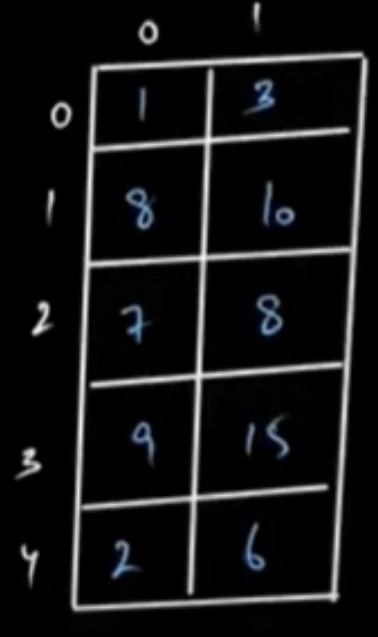
```java
class Solution {
    public boolean carPooling(int[][] trips, int capacity) {
        TreeMap<Integer,Integer> changes = new TreeMap<>();

        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;
    }
}
```

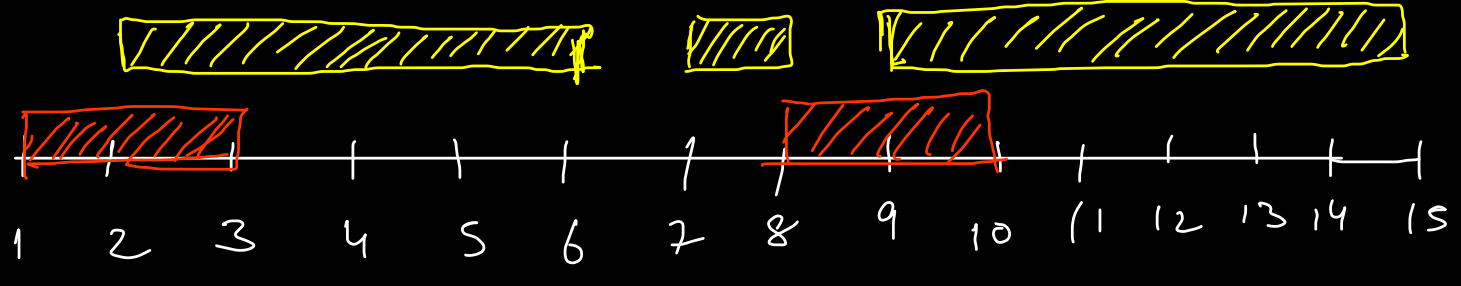
TC: $O(2N \log_2 N)$

56. Merge Intervals

Medium 13582 527 Add to List Share

Given an array of `intervals` where `intervals[i] = [start_i, end_i]`, merge all overlapping intervals, and return *an array of the non-overlapping intervals that cover all the intervals in the input.*



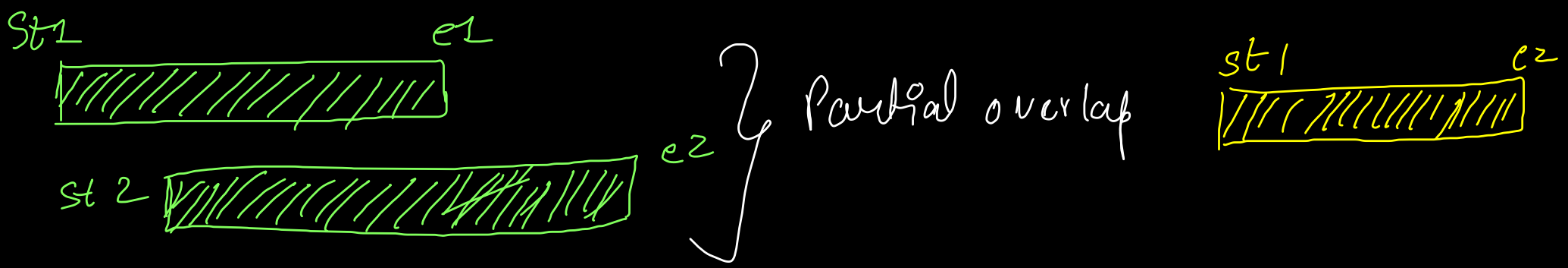


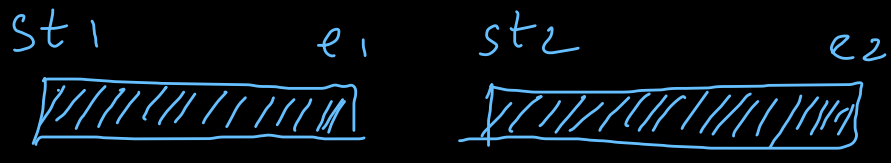
{1-6} {7-8} {8-15} → solution if

(7-8) & (8-10)

are considered non-overlapping

If we sort on the basis of starting index.





```java
class Solution {
    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];

            //curr interval start time less than end time of last interval -> can merge
            if(currInt[0] <= lastInt[1]) {
                if(lastInt[1] < currInt[1])
                    lastInt[1] = Math.max(lastInt[1],currInt[1]);
            } else { //cant merge
                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;
    }
}
```

TC
$O(N \log N)$

SC

$O(N)$ output + $O(N)$ input + $O(1)$ extra space

57. Insert Interval

Medium 4814 340 Add to List Share

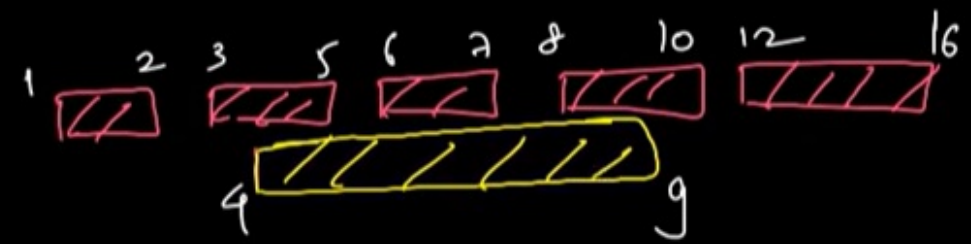
You are given an array of non-overlapping intervals `intervals` where `intervals[i] = [start_i, end_i]` represent the start and the end of the i^{th} interval and `intervals` is sorted in ascending order by `start_i`. You are also given an interval `newInterval = [start, end]` that represents the start and end of another interval.

Insert `newInterval` into `intervals` such that `intervals` is still sorted in ascending order by `start_i` and `intervals` still does not have any overlapping intervals (merge overlapping intervals if necessary).

Return `intervals` *after the insertion.*

Insert Interval

[[1,2],[3,5],[6,7],[8,10],[12,16]], 24, 93



O(N)

1. find first
2. find last
3. merge

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

```java
public int findFirst(int[][] intervals,int[] interval) {

    for(int i=0;i<intervals.length;i++) {
        if(intervals[i][1] >= interval[0]) return i;
    }

    return intervals.length;
}

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

    return -1;
}
```

```java
public int[][] insert(int[][] intervals, int[] newInterval) {
    int firstIdx = findFirst(intervals,newInterval);
    int lastIdx = findLast(intervals,newInterval);

    //non merging -> firstIdx > lastIdx


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

    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]);
        }

    }

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

    return ans;
}
```

No merging

Merging

986. Interval List Intersections

Medium 4196 86 Add to List Share

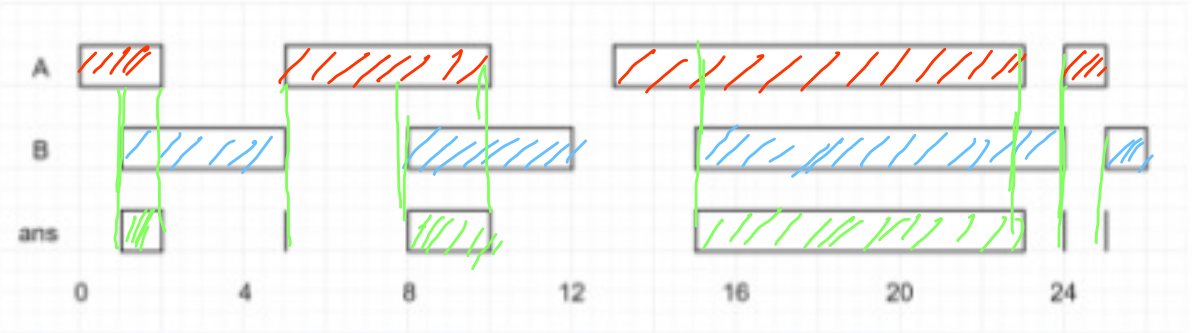
You are given two lists of closed intervals, `firstList` and `secondList`, where `firstList[i] = [start_i, end_i]` and `secondList[j] = [start_j, end_j]`. Each list of intervals is pairwise **disjoint** and in **sorted order**.

Return *the intersection of these two interval lists.*

A **closed interval** `[a, b]` (with `a <= b`) denotes the set of real numbers `x` with `a <= x <= b`.

The **intersection** of two closed intervals is a set of real numbers that are either empty or represented as a closed interval. For example, the intersection of `[1, 3]` and `[2, 4]` is `[2, 3]`.

Example 1:




```
Input: firstList = [[0,2],[5,10],[13,23],[24,25]],
secondList = [[1,5],[8,12],[15,24],[25,26]]
Output: [[1,2],[5,5],[8,10],[15,23],[24,24],
[25,25]]
```

Circular Tour { Gas Station : LC }

134. Gas Station

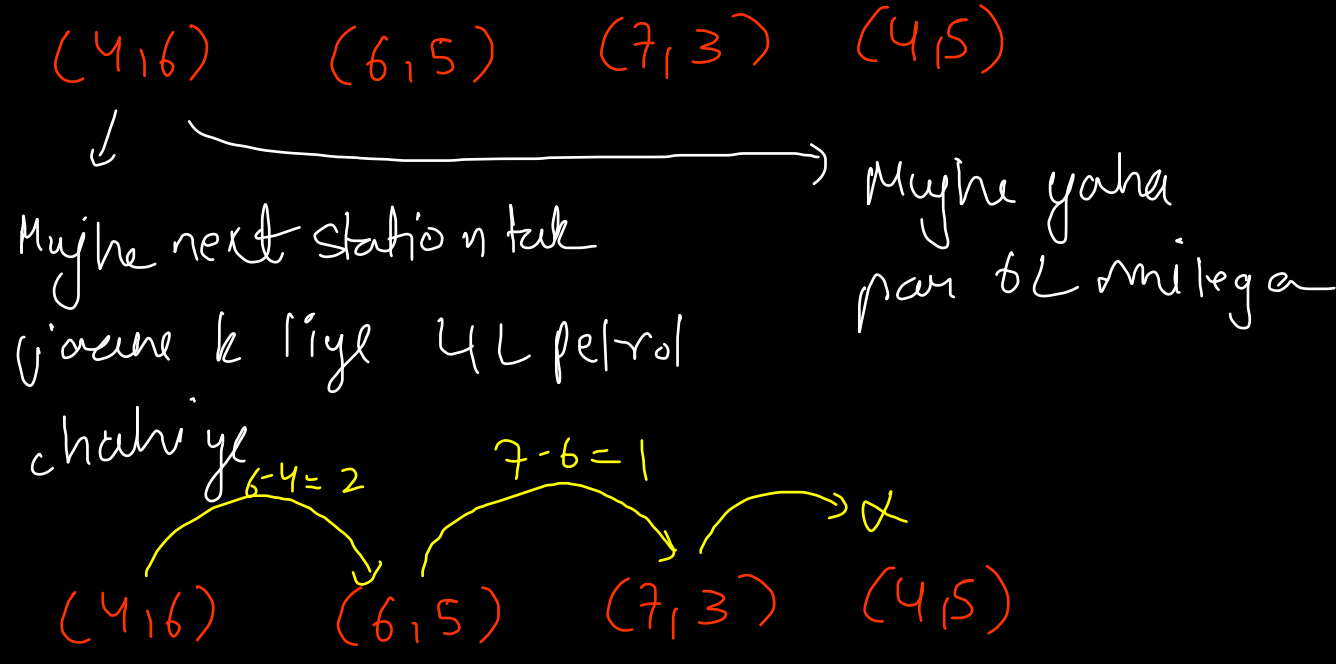
Medium 5861 605 Add to List Share

There are `n` gas stations along a circular route, where the amount of gas at the `i^th` station is `gas[i]`.

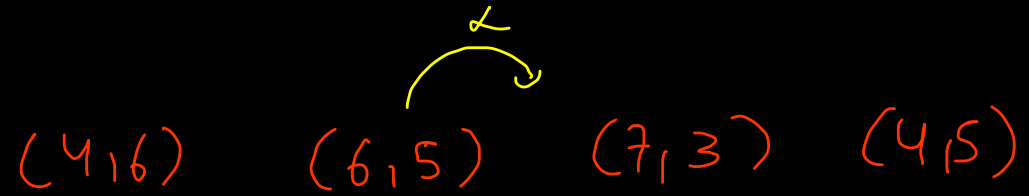
You have a car with an unlimited gas tank and it costs `cost[i]` of gas to travel from the `i^th` station to its next `(i + 1)^th` station. You begin the journey with an empty tank at one of the gas stations.

Given two integer arrays `gas` and `cost`, return *the starting gas station's index if you can travel around the circuit once in the clockwise direction, otherwise return* `-1`. If there exists a solution, it is **guaranteed** to be **unique**

{ Cost , Gas }

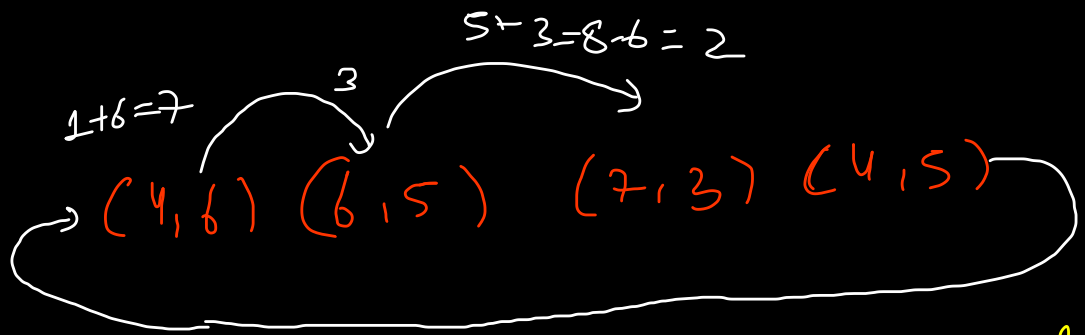


Start = 0 Hence 0 cannot be starting pt.



start = 1

start = 2



start = 3

∝ So, This can also not be our answer

Circular Tour {has Station} {Cost, gas}

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

journey : 0 : (5,0) —6-5→ (6,7) —8-6→ (7,4) —6-7=-1→ ∝

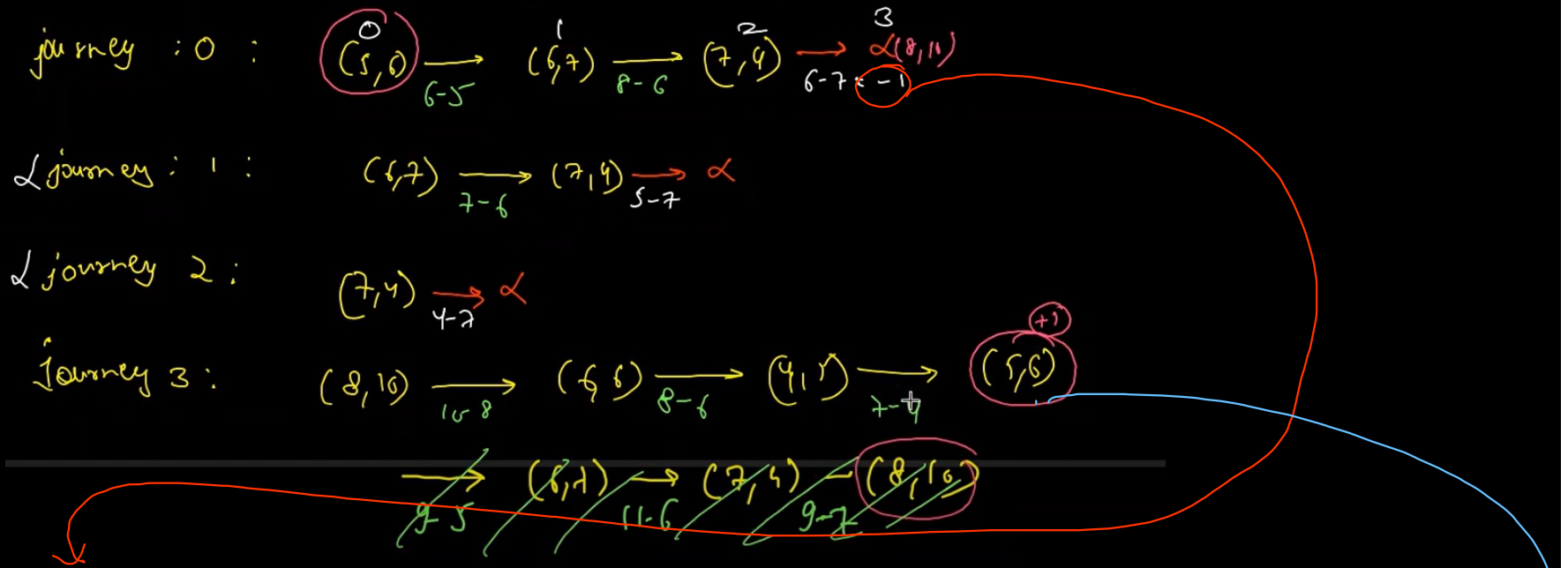
journey : 1 : (6,7) —7-6→ (7,4) —5-7→ ∝

journey 2 : (7,4) → ∝

journey 3 : (8,10) —10-8→ (6,5) —8-6→ (4,1) —7-4→ (5,6) —9-5→ (6,1) —11-6→ (7,4) —9-7→ (8,10)

} Reject 0, 1 & 2 bcz only (optimization)

This is coz agar hum kuch units of petrol lek vaha se aage nahi ja pae, so vo liye bina to kya he jaenge.



Pehle hum (5,6) par 1 unit petrol ka deficit hua tha.

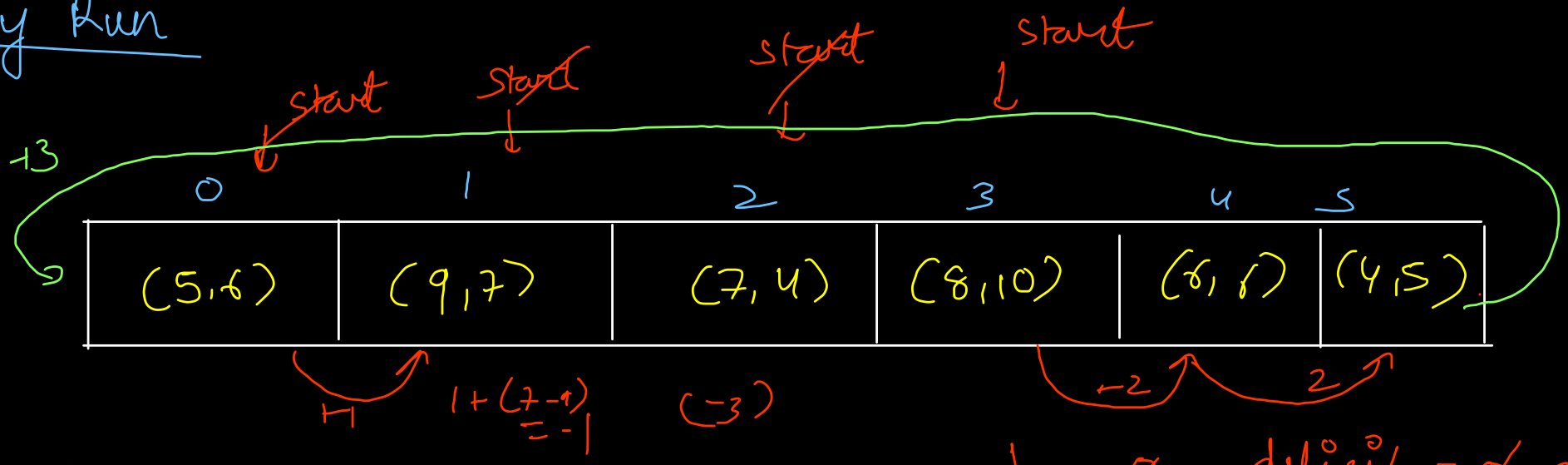
So, now if we reach (5,6) & humare pass 1 unit se zyada petrol hua, to hum journey complete kar paenge.

Hence completing the circle is not necessary.

```java
class Solution {
    public int canCompleteCircuit(int[] gas, int[] cost) {
        int n = gas.length;

        int extraGas = 0;
        int start = 0;
        int deficit = 0;

        for(int i=0;i<n;i++) {
            extraGas += (gas[i] - cost[i]);
            if(extraGas < 0) {
                start = i + 1;
                deficit -= extraGas;
                extraGas = 0;
            }
        }

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

TC: O(N)

Dry Run



extra = 0 , dyficit = 0 2+3 = 4

x 0

-3 0 ≠ 3

```java
class Solution {
    public int canCompleteCircuit(int[] gas, int[] cost) {
        int n = gas.length;

        int extraGas = 0;
        int start = 0;
        int deficit = 0;

        for(int i=0;i<n;i++) {
            extraGas += (gas[i] - cost[i]);
            if(extraGas < 0) {
                start = i + 1;
                deficit -= extraGas;
                extraGas = 0;
            }
        }

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

Chocolate Distribution Problem

Easy Accuracy: 53.25% Submissions: 47232 Points: 2

Given an array **A[ ]** of positive integers of size **N**, where each value represents the number of chocolates in a packet. Each packet can have a variable number of chocolates. There are **M** students, the task is to distribute chocolate packets among **M** students such that :

1. Each student gets **exactly** one packet.
2. The difference between maximum number of chocolates given to a student and minimum number of chocolates given to a student is minimum.

Select M out of N such that the diff b/w the max value & min value of M selected will be minimum -

{ 3, 4, 1, 9, 16, 7, 9, 12 }

Sort the array

Take a sliding window of size M now & find min diff.

```java
class Solution
{
    public long findMinDiff (ArrayList<Integer> a, int n, int m)
    {
        // your code here
        Collections.sort(a);
        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;
    }
}
```

$TC = O(N \log_2 N)$

853. Car Fleet

Medium 1403 412 Add to List Share

There are `n` cars going to the same destination along a one-lane road. The destination is `target` miles away.

You are given two integer array `position` and `speed`, both of length `n`, where `position[i]` is the position of the i^{th} car and `speed[i]` is the speed of the i^{th} car (in miles per hour).

A car can never pass another car ahead of it, but it can catch up to it and drive bumper to bumper **at the same speed**. The faster car will **slow down** to match the slower car's speed. The distance between these two cars is ignored (i.e., they are assumed to have the same position).

A **car fleet** is some non-empty set of cars driving at the same position and same speed. Note that a single car is also a car fleet.

If a car catches up to a car fleet right at the destination point, it will still be considered as one car fleet.

Return *the **number of car fleets** that will arrive at the destination.*

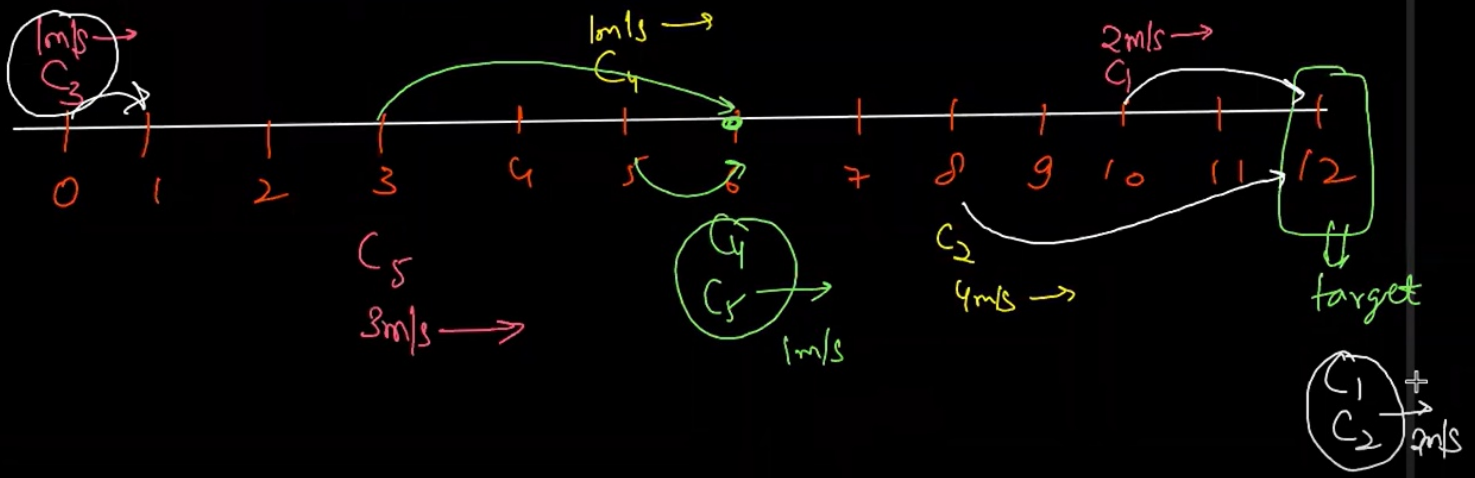
Return the no of car fleets.

→ Single car is also a car fleet.

→ fleets formed at target are also car fleets.

Car Fleet

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



① Right to left traversal (based on dist)

② Calculate time taken to reach dest & if time of a car at its left is less than or equal to our time, its the same fleet, else increment the count of fleets. (Time calculated in double)

③ max Time.

```
class Solution {

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


        Pair() {}

        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;
        }

    }

    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);

        double maxTime = 0.0;
        int countOfFleets = 0;

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

        return countOfFleets;
    }
}
```

Candy (LC) / Temple Offerings (GfG)

135. Candy

Hard 2788 251 Add to List Share

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

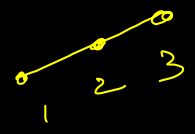
You are giving candies to these children subjected to the following requirements:

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

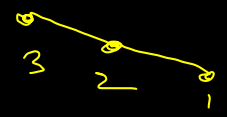
Return *the minimum number of candies you need to have to distribute the candies to the children.*

Trivial Cases

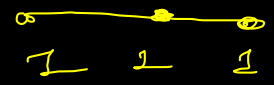
{10, 40, 60}
1 2 3



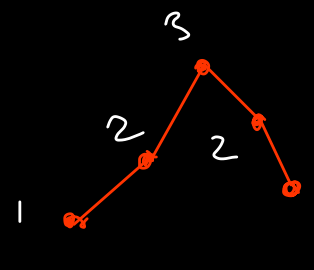
{50, 30, 20}
3 2 1



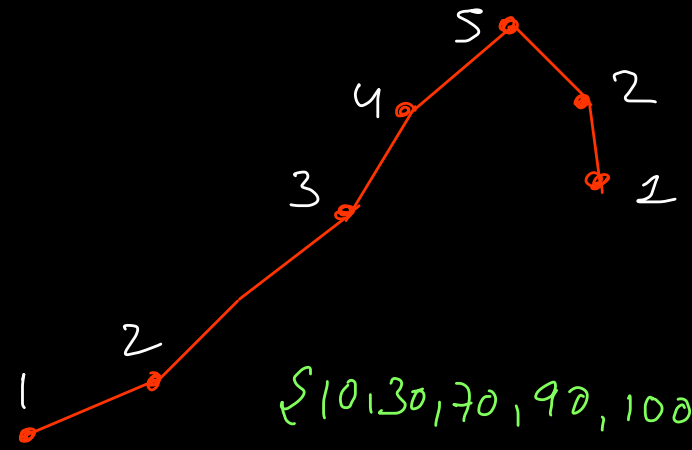
{20, 20, 20}
1 1 1



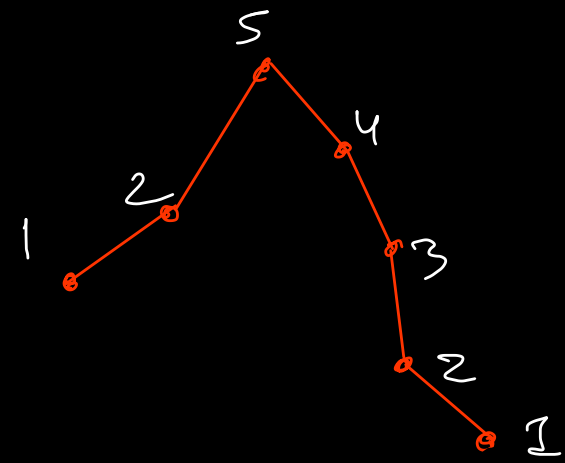
Examples



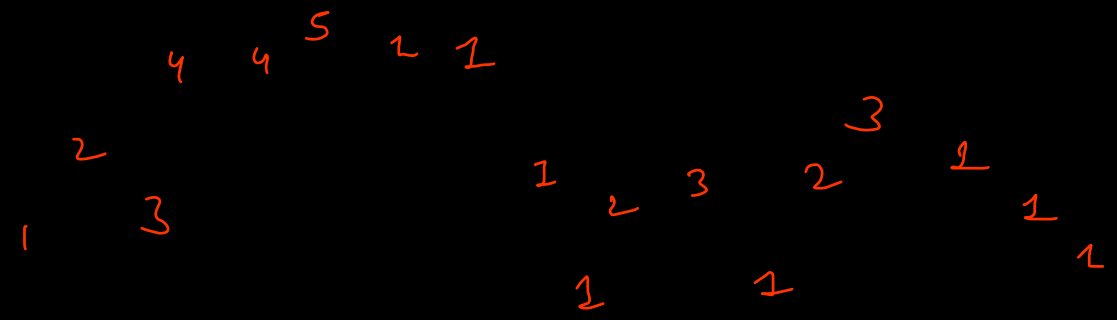
{10, 30, 70, 60, 20}
1 2 3 2 1



{10, 30, 70, 90, 100, 50, 40}
1 2 3 4 5 2 1

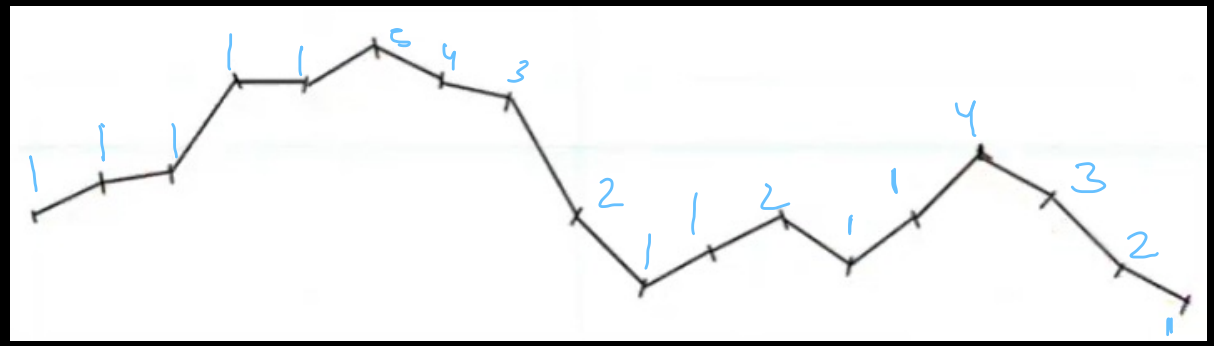


{80, 90, 100, 80, 60, 40, 30}

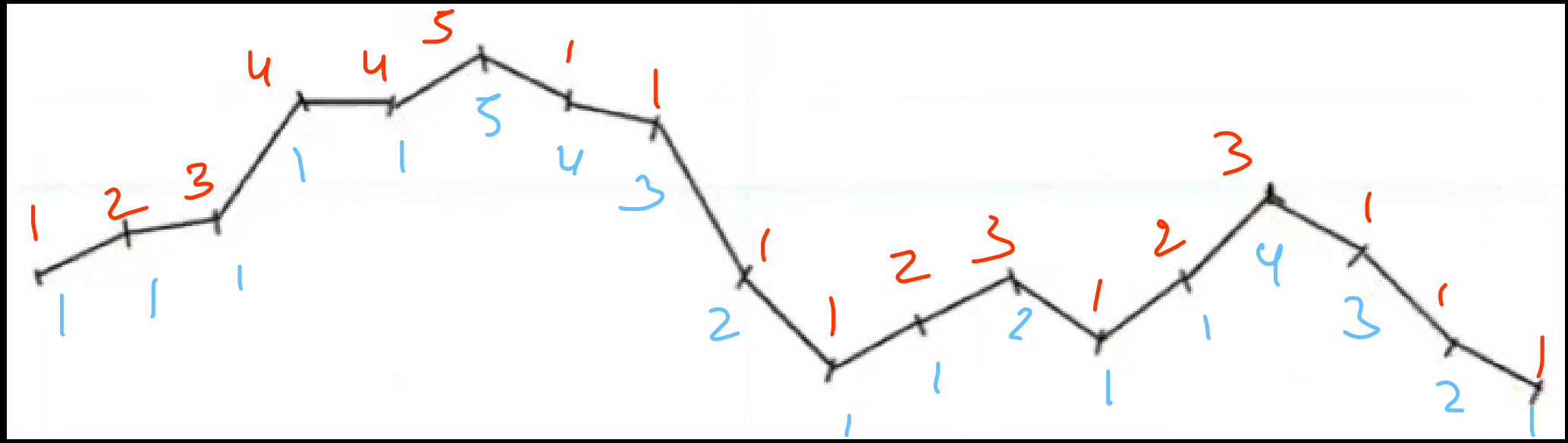


① 1^{st} iteration from left to right. See if current ele is greater than the previous ele, give it more candy as compared to prev.

If curr is less than prev, try to give it min candies.



② 2nd iteration from right to left. If curr ele is greater than the next ele (right ele), give it +1 candy as compared to right ele. If it is smaller than the right side ele, just give it min candy.



Now, the answer will be max of candies at each point.

```java
class Solution {
    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 ans=0;
        for(int i=0;i<ratings.length;i++) {
            ans += Math.max(left[i],right[i]);
        }

        return ans;
    }
}
```

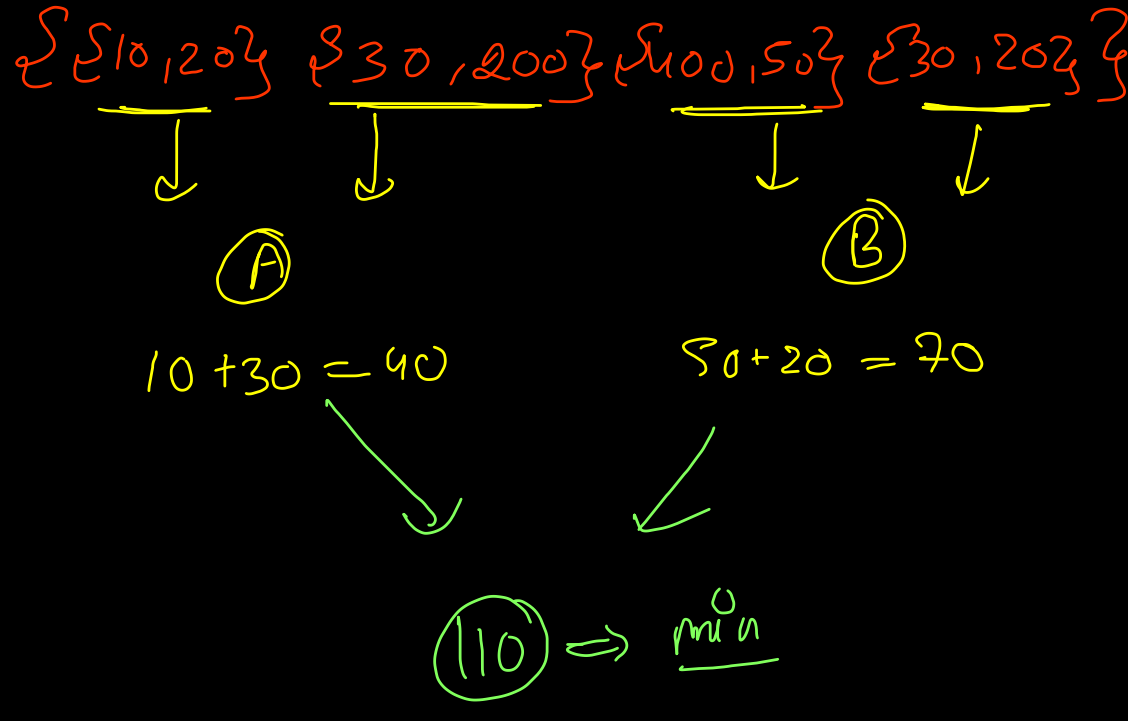
1029. Two City Scheduling

Medium 3659 276 Add to List Share

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

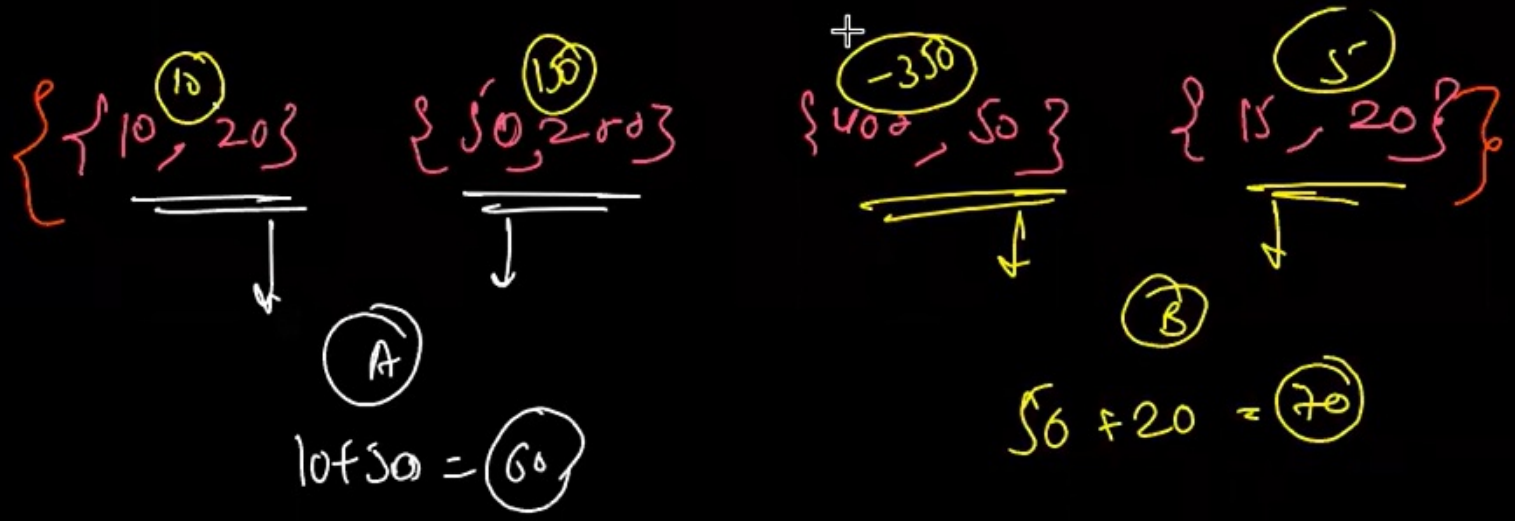
Return *the minimum cost to fly every person to a city* such that exactly `n` people arrive in each city.

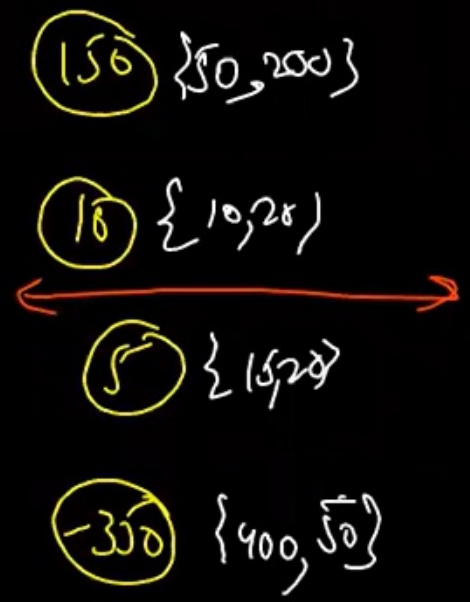
Print minimum cost out of all $2^N C_N$ combinations.



Two city Scheduling

sort on the basis of





A company is planning to interview `2n` people. Given the array `costs` where `costs[i] = [aCost[i], bCost[i]]`, the cost of flying the `i`th person to city `a` is `aCost[i]`, and the cost of flying the `i`th person to city `b` is `bCost[i]`.

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

```java
class Solution {
    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;
    }
}
```

Huffman Encoding & Decoding

Input String : B CC A BB DD A E CC BB A E DD CC

frequency

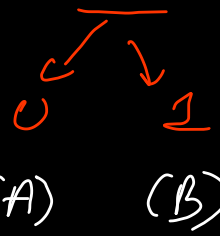
A → 3
+
B → 5
+
C → 6
+
D → 4
+
E → 2
20

Encrypted ⟶ Decrypted

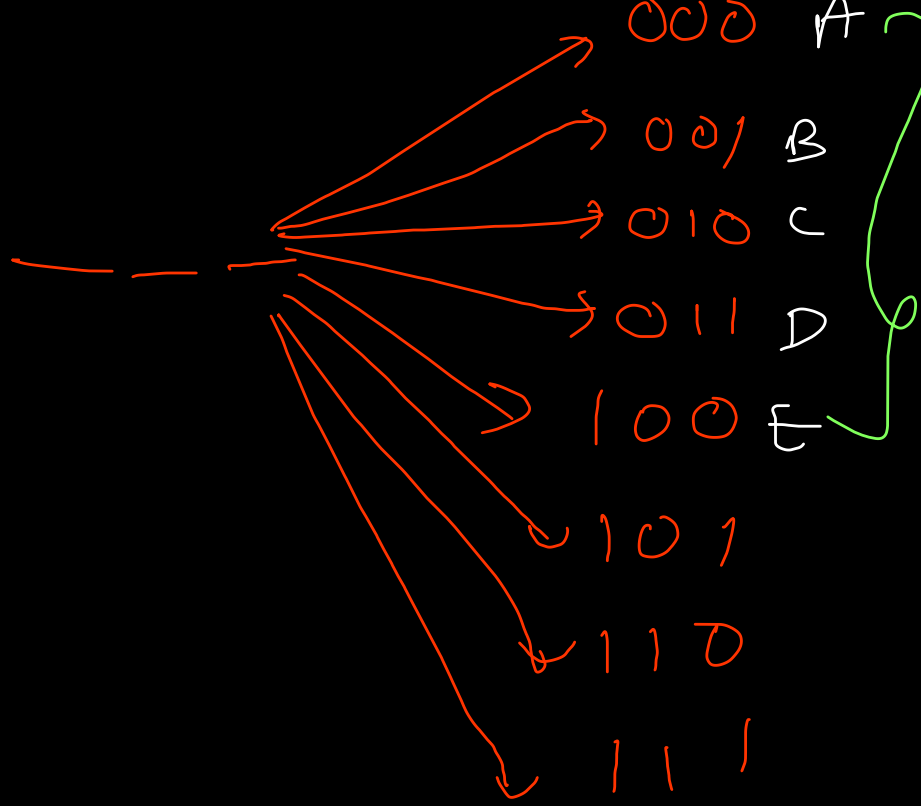
Shortest possible encoded string.

Current size of input string if characters are in ASCII = 20 x 1byte = 20 * 8bits = 160 bits.

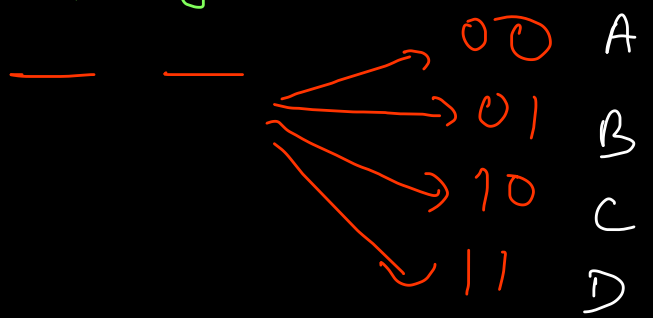
Fixed Size Encoding



(Taking only 1 bit)



Taking 2 bits



Message size = 20 * 3 = 60 bits.

Variable Size Encoding

Input String : B CC A BB DD A E CC BB A E DD CC

Frequency

A → 3
+
B → 5
+
C → 6
+
D → 4
+
E → 2
20

Sort on the basis of freq

C → 6 → assign min bits to it.

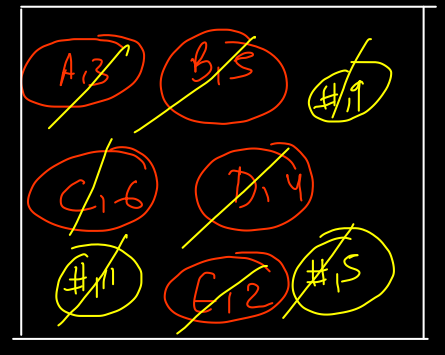
B → 5

D → 4

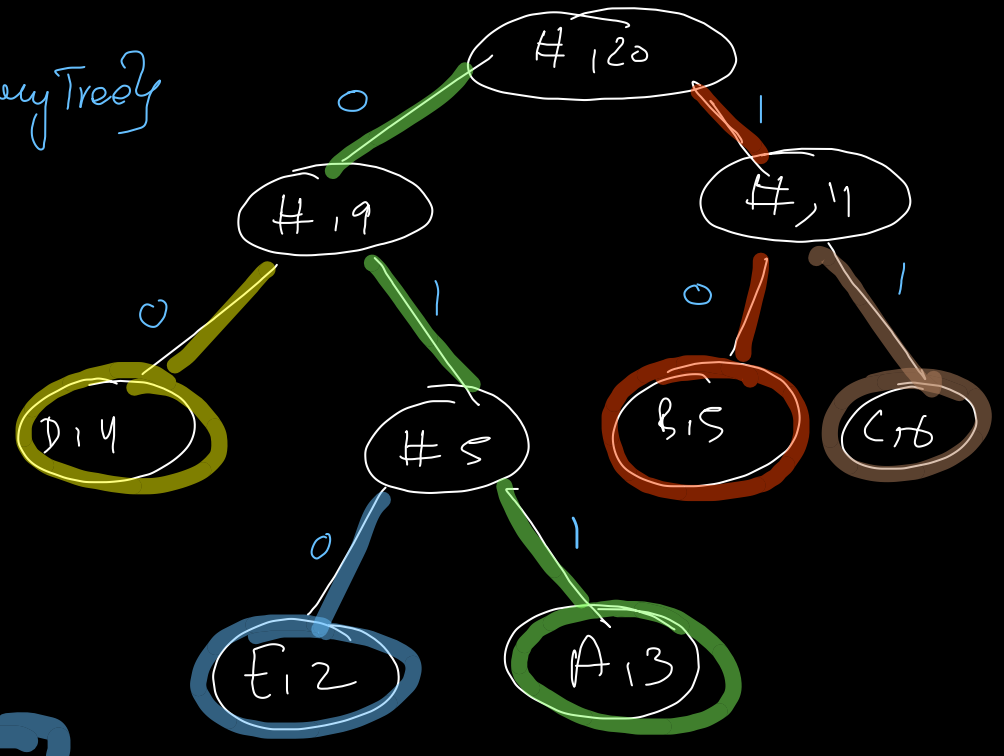
A → 3

E → 2

Preprocessing
(Encoding & Binary Tree)



Min Heap



for decryption, we need either the binary tree or the table -

A -> 0 11
B -> 01
C -> 11
D -> 00
E -> 010

$3\times3 + 5\times2 + 6\times2 + 4\times2 + 3\times2$

$= 9 + 10 + 12 + 8 + 6$

$= 45$ bits

```java
public static class Node implements Comparable<Node>{
    char ch;
    // int idx;
    int freq;
    Node left;
    Node right;

    Node(char ch, int freq){
        this.ch = ch;
        // this.idx
        this.freq = freq;
        this.left = null;
        this.right = null;
    }

    public int compareTo(Node other){
        return this.freq - other.freq;
    }
}
```

```java
ArrayList<String> encoding;
public void preorder(Node root, String str){
    if(root.left == null && root.right == null){
        encoding.add(str);
        return;
    }

    if(root.left != null)
        preorder(root.left, str + "0");

    if(root.right != null)
        preorder(root.right, str + "1");
}
```

```java
PriorityQueue<Node> q = new PriorityQueue<>();
for(int i=0; i<N; i++){
    q.add(new Node(S.charAt(i), f[i]));
}

while(q.size() > 1){
    Node left = q.remove();
    Node right = q.remove();
    Node root = new Node('#', left.freq + right.freq);
    root.left = left;
    root.right = right;
    q.add(root);
}

Node root = q.remove();
encoding = new ArrayList<>();
preorder(root, "");

return encoding;
```

O(NlogN)

O(NlogN)

O(N)

Biased Standings (SPOJ)

BAISED - Biased Standings

no tags

Usually, results of competitions are based on the scores of participants. However, we are planning a change for the next year of IPSC. During the registration each team will be able to enter a single positive integer : their preferred place in the ranklist. We would take all these preferences into account, and at the end of the competition we will simply announce a ranklist that would please all of you.

But wait... How would that ranklist look like if it won't be possible to satisfy all the requests?

Suppose that we already have a ranklist. For each team, compute the distance between their preferred place and their place in the ranklist. The sum of these distances will be called the badness of this ranklist.

Problem specification

Given team names and their preferred placements find one ranklist with the minimal possible badness.

Input specification

The first line of the input file contains an integer **T** specifying the number of test cases. Each test case is preceded by a blank line.

Each test case looks as follows: The first line contains **N** : the number of teams participating in the competition. Each of the next **N** lines contains a team name (a string of letters and numbers) and its preferred place (an integer between 1 and **N**, inclusive). No two team names will be equal.

Output specification

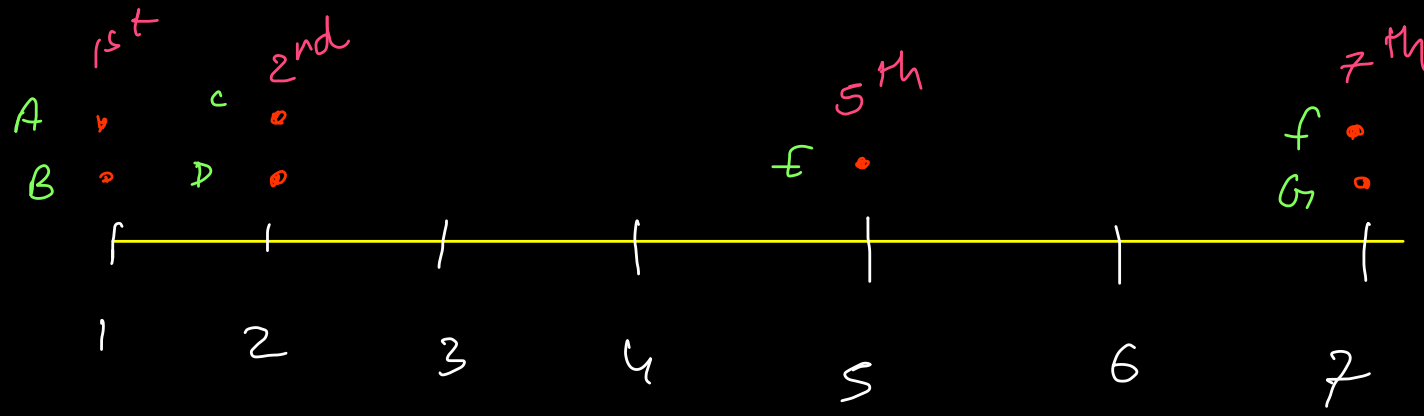
For each of the test cases output a single line with a single integer : the badness of the best ranklist for the given teams.

$1 \leq a[i].preferredRank \leq N$

```
7
noobz 1 (0)
1lamas 2 (0)
Winn3rz 2 (1)
5thwheel 1 (3)
NotoricCoders 5 (0)
StrangeCase 7 (1)
WhoKnows 7 (0)
```

$\Rightarrow$ Total badness $= 0+0+1+3+0+1+0$

$= 5$



$B \to 2$ $C \to 3$

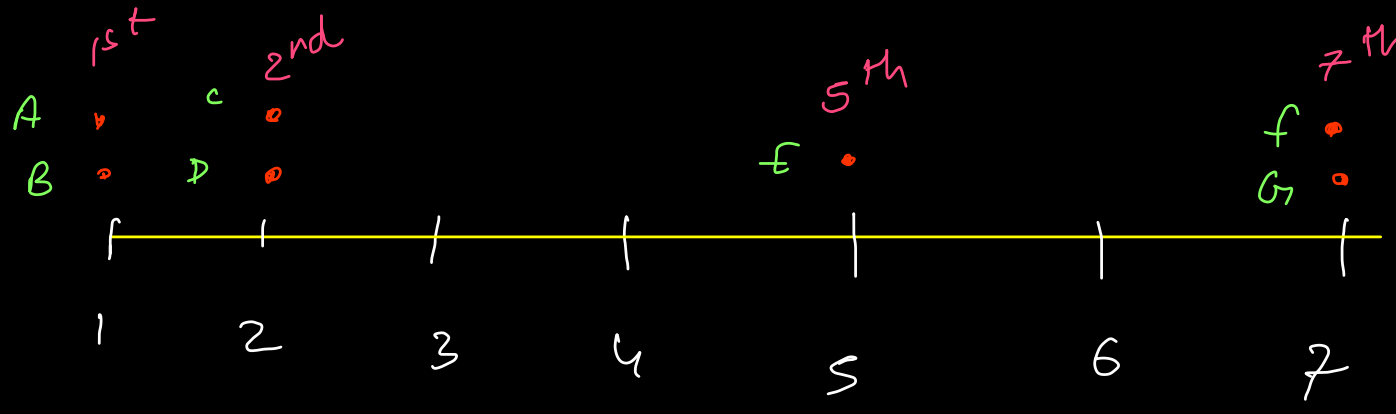
$2-1=1$ $3-2=1$

$1+1=2$

$B \to 3$ $C \to 2$

$3-1=2$ $2-2=0$

$2+0=2$



Add all the elements in an ordered set based on their preferred rank.

Now, remove one ele from ordered set & allocate it to next bucket.

B → 2 C → 3

-1 = 1 3-2 = 1

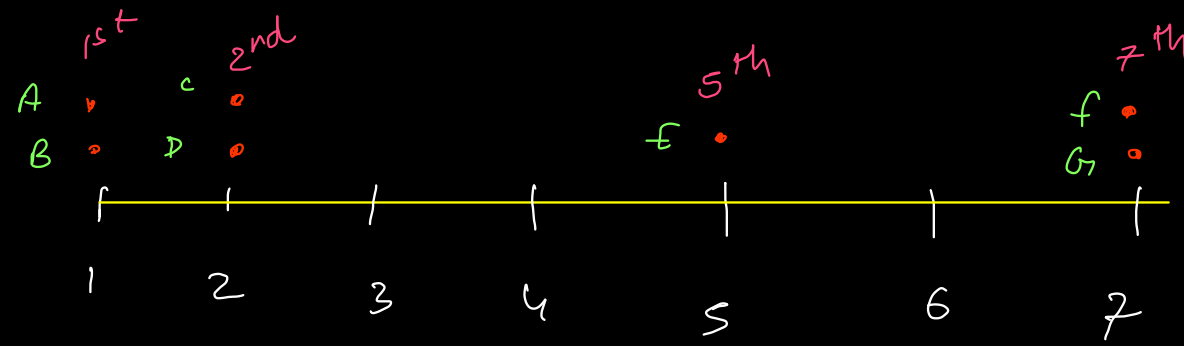
1+1 = (2)

B → 3 C → 2

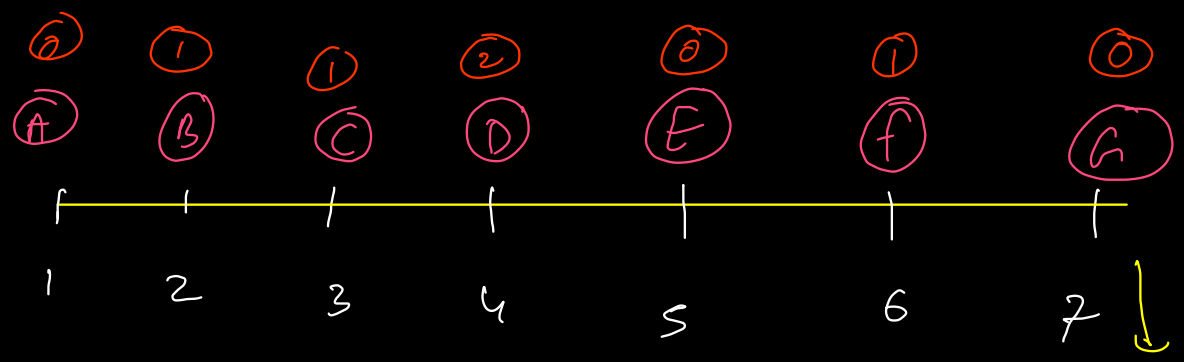
3-1 = 2 2-2 = 0

2+0 = (2)

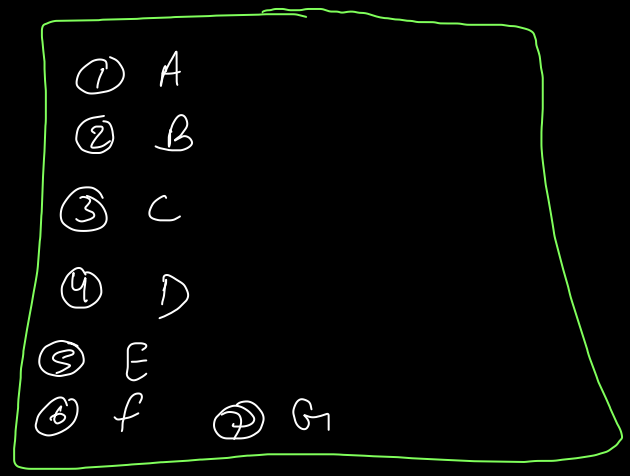
Badness remains same



→ Desired



Actual allotted



0+1+1+2+0+1+0

= 2+2+1 = 5 → min

This approach works because we don't want to assign max eles to the correct pos rather just want to have min total badness.

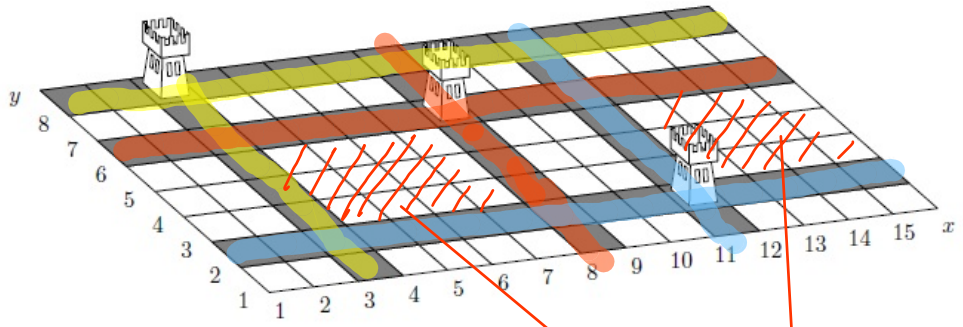
Defense of Kingdom (SPOJ)

DEFKIN - Defense of a Kingdom

no tags

Theodore implements a new strategy game "Defense of a Kingdom". On each level a player defends the Kingdom that is represented by a rectangular grid of cells. The player builds crossbow towers in some cells of the grid. The tower defends all the cells in the same row and the same column. No two towers share a row or a column.

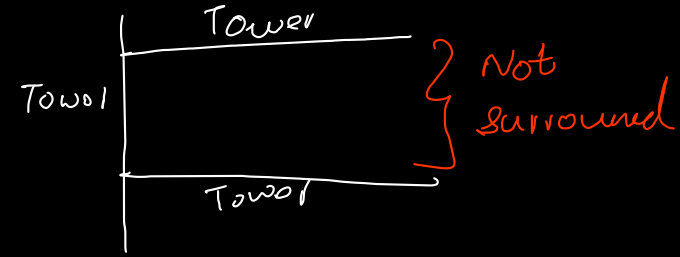
The penalty of the position is the number of cells in the largest undefended rectangle. For example, the position shown on the picture has penalty 12.



Help Theodore write a program that calculates the penalty of the given position.

Corner case

A path might not be surrounded from all the ends.



*** Rows & cols are independent

Largest undefended area (can be either of them)

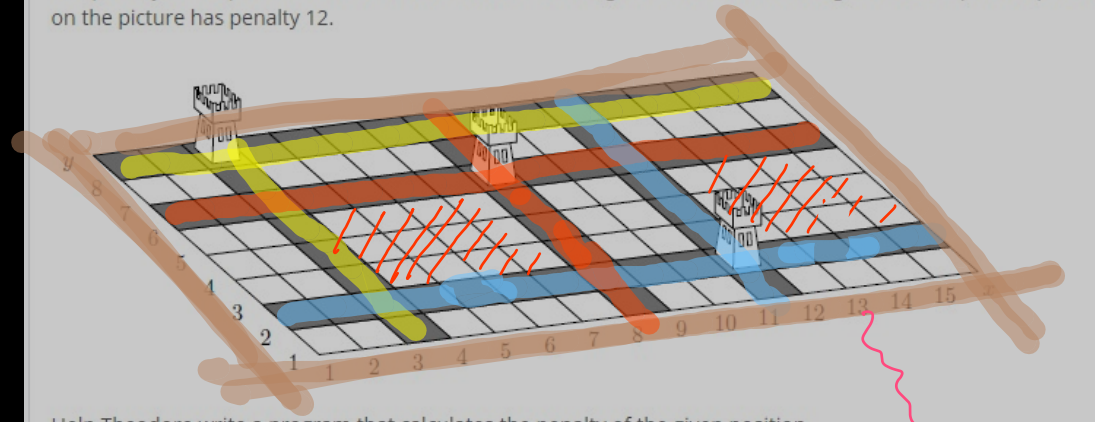
To handle the corner case, we presume that there are watch towers at all the 4 ends as shown below -

DEFKIN - Defense of a Kingdom

no tags

Theodore implements a new strategy game "Defense of a Kingdom". On each level a player defends the Kingdom that is represented by a rectangular grid of cells. The player builds crossbow towers in some cells of the grid. The tower defends all the cells in the same row and the same column. No two towers share a row or a column.

The penalty of the position is the number of cells in the largest undefended rectangle. For example, the position shown on the picture has penalty 12.



Help Theodore write a program that calculates the penalty of the given position.

Towers at all 4 ends

rows {0 ... 9}

cols {0 ... 16}

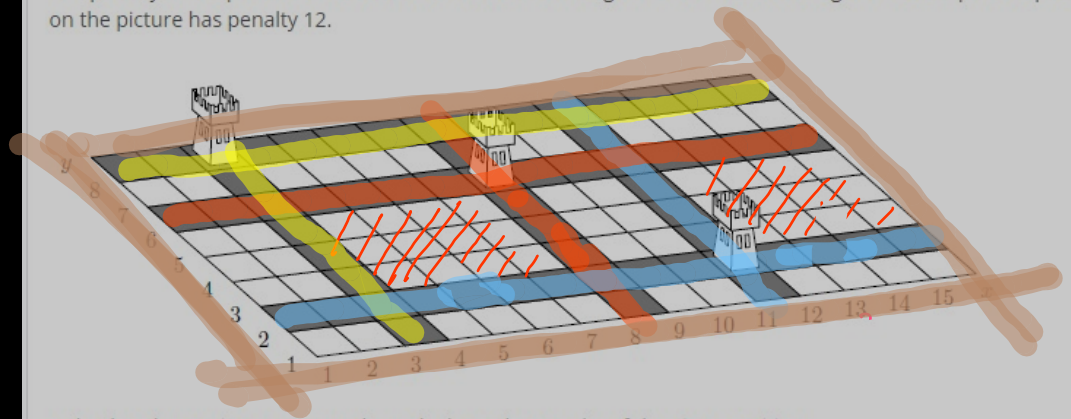
Now cal max diff blw adj rows & cols & multiply them. This will be the largest area.

DEFKIN - Defense of a Kingdom

no tags

Theodore implements a new strategy game “Defense of a Kingdom”. On each level a player defends the Kingdom that is represented by a rectangular grid of cells. The player builds crossbow towers in some cells of the grid. The tower defends all the cells in the same row and the same column. No two towers share a row or a column.

The penalty of the position is the number of cells in the largest undefended rectangle. For example, the position shown on the picture has penalty 12.



Help Theodore write a program that calculates the penalty of the given position.

maxLen

(r-l-1)

1 1 3 1 0

{0 2 6 8 9} rows

to handle corner case

{0 3 8 11 16}

2 4 1 4

→ maxBreadth

maxArea = maxLen * maxBreadth = 4 * 3 = 12

```java
Run | Debug
public static void main(String[] args) throws Exception {
    Scanner scn = new Scanner(System.in);
    int T = scn.nextInt();

    for(int t=0;t<T;t++) {
        int length = scn.nextInt();
        int breadth = scn.nextInt();
        int tower = scn.nextInt();

        ArrayList<Integer> rows = new ArrayList<>();
        rows.add(e: 0); rows.add(length + 1);
        ArrayList<Integer> cols = new ArrayList<>();
        cols.add(e: 0); cols.add(breadth + 1);

        for(int i=0;i<tower;i++) {
            int row = scn.nextInt();
            rows.add(row);
            int col = scn.nextInt();
            cols.add(col);
        }

        Collections.sort(rows);
        Collections.sort(cols);
```

```java
            int maxRow = 0;
            for(int i=1;i<rows.size();i++) {
                int curr = rows.get(i) - rows.get(i-1) - 1;
                maxRow = Math.max(maxRow,curr);
            }

            int maxCol = 0;
            for(int i=1;i<cols.size();i++) {
                int curr = cols.get(i) - cols.get(i-1) - 1;
                maxCol = Math.max(maxCol,curr);
            }

            System.out.println(maxRow * maxCol);
        }

        scn.close();
    }
}
```

$O(N \log_2 N)$

Code for Assign Mice Holes | Biased Standings

Programming / Greedy Algorithm / Assign Mice To Holes

Assign Mice to Holes

Easy 46 17

Asked In: Amazon

Problem Description

There are N Mice and N holes that are placed in a straight line. Each hole can accomodate only 1 mouse.

The positions of Mice are denoted by array A and the position of holes are denoted by array B.

A mouse can stay at his position, move one step right from x to x + 1, or move one step left from x to x − 1. Any of these moves consumes 1 minute.

Assign mice to holes so that the time when the last mouse gets inside a hole is **minimized**.

```java
public class Solution {
    public int mice(int[] A, int[] B) {
        Arrays.sort(A);
        Arrays.sort(B);

        int maxTime = 0;
        for(int i=0;i<A.length;i++) {
            maxTime = Math.max(maxTime,Math.abs(A[i] - B[i]));
        }

        return maxTime;
    }
}
```

Cutting Rectangles

Easy Accuracy: 44.13% Submissions: 1109 Points: 2

Given a rectangle of dimensions **L** x **B** find the minimum number (**N**) of identical squares of maximum side that can be cut out from that rectangle so that no residue remains in the rectangle. Also find the dimension **K** of that square.

So, for finding min rectangles, ans is GCD of L & B.

1 * 1 → 24 * 18

2 * 2 → $\frac{(L*B)}{a^2}$ = $\frac{12 \quad 9}{2 \times 2}$ = 108

3 * 3 → $\frac{24 \times 18^{6}}{3 \times 3}$ = 48

6 * 6 = $\frac{4 \quad 3}{\frac{24 \times 18}{6 \times 6}}$ = 12

```java
class Solution{


    public static long euclidean(long a, long b) {
        if(b == 0) return a;
        return euclidean(b, a%b);
    }

    static List<Long> minimumSquares(long L, long B)
    {
        // code here
        long dimen =  euclidean(L,B);
        long noOfSquares = (L*B) / (dimen * dimen);
        List<Long> res = new ArrayList<>();
        res.add(noOfSquares);
        res.add(dimen);
        return res;
    }
}
```

Code for

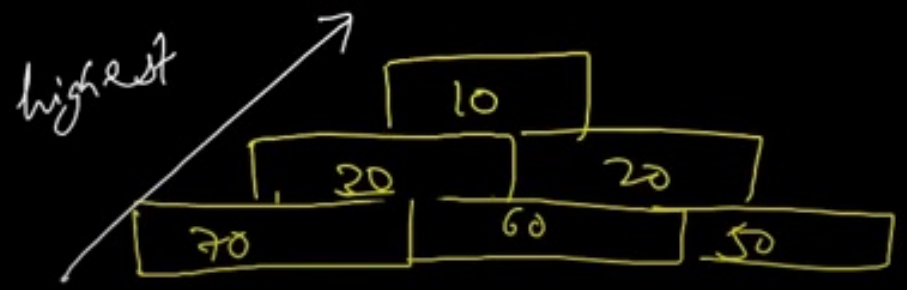
Cutting Rectangles

Highest Pyramid

(1) i^{th} level no of blocks > $(i-1)^{th}$ level of no blocks

(2) i^{th} level breadth > $(i-1)^{th}$ level breadth

```
{10, 20, 30, 50, 60, 70}
```



$\frac{x*(x+1)}{2} \leq n$ → max x that satisfies this condition is our answer.

Maximize sum(arr[i]*i) of an Array

Basic Accuracy: 38.55% Submissions: 52772 Points: 1

Given an array **A** of **N** integers. Your task is to write a program to find the maximum value of ∑**arr[i]*i**, where i = 0, 1, 2,., n 1.
You are allowed to rearrange the elements of the array.
Note: Since output could be large, hence module 10^9+7 and then print answer.

```
class Solution{
    static int m = 1000000007;
    int Maximize(int arr[], int n)
    {
        // Complete the function
        Arrays.sort(arr);
        long res = 0;

        for(int i=0;i<n;i++) {
            res  += (long)arr[i] * i;
        }

        return (int)(res % m);
    }
}
```