PW SKILLS

Lesson Plan

Greedy Algorithms

PW SKILLS

Today's Checklist:

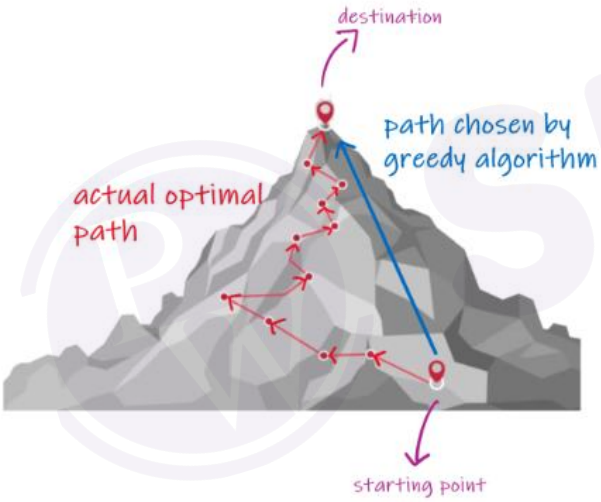
1. Algorithmic Paradigms
2. Fractional knapsack
3. Leetcode Practice Questions

Algorithmic Paradigms

A graph can be defined as group of vertices and edges that are used to connect these vertices. A graph can be seen as a cyclic tree, where the vertices (Nodes) maintain any complex relationship among them instead of having parent child relationship.

A graph G can be defined as an ordered set G(V, E) where V(G) represents the set of vertices and E(G) represents the set of edges which are used to connect these vertices.

A Graph G(V, E) with 5 vertices (A, B, C, D, E) and six edges ((A,B), (B,C), (C,E), (E,D), (D,B), (D,A)) is shown in the following figure.



Greedy algorithms make the best local choice at each step, aiming for a globally optimal solution. They prioritize immediate gains without reassessing previous decisions. Though simple and efficient, they may not always yield the optimal solution for every problem.

Maximize sum of Array After Negations [Leetcode - 1005]

Problem:

Given an integer array nums and an integer k, modify the array in the following way:
choose an index i and replace nums[i] with -nums[i].
You should apply this process exactly k times. You may choose the same index i multiple times.
Return the largest possible sum of the array after modifying it in this way.

Code:

```
class Main{
       public int largestSumAfterKNegations(int[] A, int K) {
        Arrays.sort(A);
        for (int i = 0; K > 0 && i < A.length && A[i] < 0; ++i, --
K)
            A[i] = -A[i];
        int res = 0, min = Integer.MAX_VALUE;
        for (int a : A) {
            res += a;
            min = Math.min(min, a);
        }
        return res - (K % 2) * min * 2;
    }
}
```

Explanation:

This Java code defines a Solution class with a method largestSumAfterKNegations, which takes an array of integers A and an integer K. The method first sorts the array and then negates the first K negative elements. Next, it calculates the sum of the modified array and identifies the minimum element. The final result is obtained by subtracting twice the minimum element multiplied by the remaining negations (K % 2 times) from the sum. The code effectively finds the largest possible sum after at most K negations on the array elements.

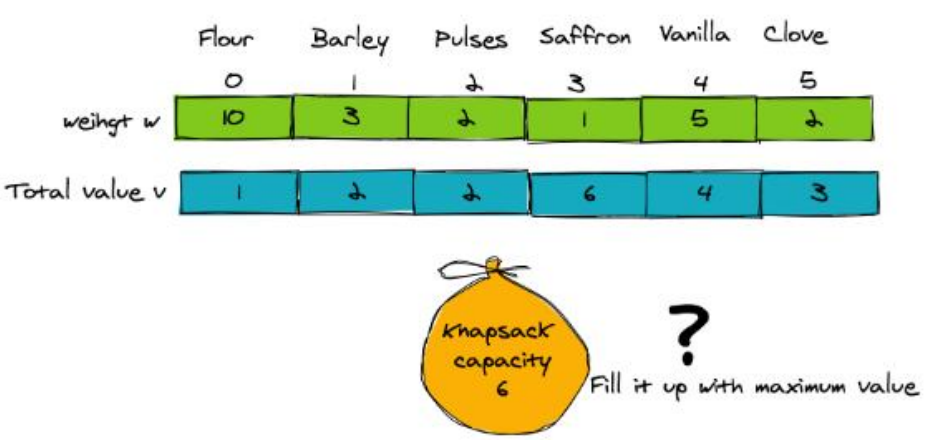
Time Complexity:

Sorting the array takes O(n log n) time.
Negating the first K negative numbers in the sorted array takes O(K) time.
The second loop for calculating the result takes O(n) time.
Overall, the time complexity is O(n log n + K).

Space Complexity:

The algorithm uses a constant amount of extra space (for variables like i, res, min). Therefore, the space complexity is O(1).

Fractional Knapsack



Paradigms

The Fractional Knapsack problem involves selecting items with weights and values to maximize total value within a fixed-capacity knapsack. The greedy algorithm sorts items by value-to-weight ratio and iteratively adds items or fractions to the knapsack until full. Though not guaranteed to find the global optimum, it is efficient with a time complexity of O(n log n), where n is the number of items.

Coding Questions

Maximum Units on a Truck [Leetcode - 1710]

Problem:

You are assigned to put some amount of boxes onto one truck. You are given a 2D array boxTypes, where boxTypes[i] = [number Of Boxes, numberOfUnitsPerBoxi]:
number Of Boxes is the number of boxes of type i.
numberOfUnitsPerBoxi is the number of units in each box of the type i.
You are also given an integer truckSize, which is the maximum number of boxes that can be put on the truck. You can choose any boxes to put on the truck as long as the number of boxes does not exceed truckSize.
Return the maximum total number of units that can be put on the truck.

Code

```
class Main{
    public int maximumUnits(int[][] boxTypes, int truckSize) {
        Arrays.sort(boxTypes, Comparator.comparingInt(o -> -o[1]));
        int ans = 0, i = 0, n = boxTypes.length;
        while (i < n && truckSize > 0) {
            int maxi = Math.min(boxTypes[i][0], truckSize);
            ans += maxi * boxTypes[i][1];
            i++;
            truckSize -= maxi;
        }
        return ans;
    }
}
```

Explanation:

The code sorts box types by units per box in descending order.It iterates through boxes, adding maximum units to total as per truck capacity.The method returns the total units that fit in the truck.Utilizes Java's sorting and greedy algorithm for efficient implementation.

Time Complexity:

The dominant factor in time complexity is the sorting operation using Arrays.sort. Sorting has a time complexity of O(n log n), where n is the number of elements in the array. In this case, the array being sorted is boxTypes, and it is sorted based on the second element of each subarray (values) in descending order.

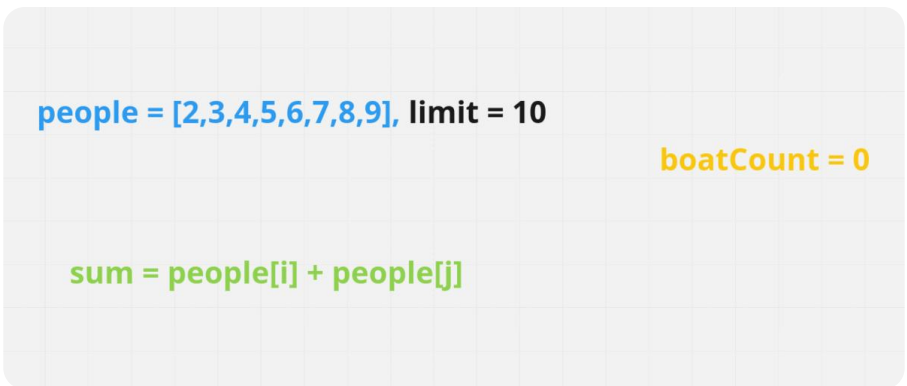
Space Complexity:

The space complexity is O(1) because the algorithm uses a constant amount of space regardless of the input size.

Boats to save People [Leetcode - 89]

Problem:

You are given an array of people where people[i] is the weight of the ith person, and an infinite number of boats where each boat can carry a maximum weight of limit. Each boat carries at most two people at the same time, provided the sum of the weight of those people is at most limited.Return the minimum number of boats to carry every given person.



Code

```
class Main{
    public int numRescueBoats(int[] people, int limit) {
        int c=0;
        Arrays.sort(people);
        int b=0;
        int start=0;
        int end=people.length-1;
        while(start<=end){
          if((people[start]+people[end])<=limit){
              start++;
              end--;
              b++;
          }else{
              end--;
              b++;
          }
        }
        return b;
    }
}
```

Explanation:

This Java code aims to find the minimum number of boats needed to rescue people, given their weights and a weight limit per boat. It sorts the array of people in ascending order, then uses a two-pointer approach (start and end) to iterate through the array, incrementing the boat count (b) based on whether the sum of weights at the pointers is within the limit or not. The method returns the total number of boats required for the rescue.

Time Complexity:

The code begins by sorting the people array using Arrays.sort. Sorting has a time complexity of O(n log n), where n is the number of people.

Two-Pointer Approach:
The algorithm uses a two-pointer approach (start and end) to iterate through the sorted array.
The while loop runs until the start pointer exceeds the end pointer.
Each iteration of the loop involves constant time operations.

Space Complexity:

The space used is primarily for a few variables (c, b, start, end), and it does not depend on the input size.
Sorting is often an in-place operation and doesn't contribute to additional space complexity.
Therefore, the space complexity is O(1) (constant).

Minimum Product Subset

Problem:

The minimum product subset of an array refers to a subset of elements from the array such that the product of the elements in the subset is minimized

Input : a[] = { -1, -1, -2, 4, 3 }
Output : -24

Explanation : Minimum product will be (-2 * -1 * -1 * 4 * 3) = -

Code

```
class Main{
    static int minProductSubset(int a[], int n)
    {
        if (n == 1)
            return a[0];
       int negmax = Integer.MIN_VALUE;
        int posmin = Integer.MAX_VALUE;
        int count_neg = 0, count_zero = 0;
        int product = 1;
        for (int i = 0; i < n; i++) {
            if (a[i] == 0) {
                count_zero++;
                continue;
            }
            if (a[i] < 0) {
                count_neg++;
                negmax = Math.max(negmax, a[i]);
            }
            if (a[i] > 0 && a[i] < posmin)
                posmin = a[i];
            product *= a[i];
        }
        if (count_zero == n
            || (count_neg == 0 && count_zero > 0))
            return 0;
        if (count_neg == 0)
            return posmin;
        if (count_neg % 2 == 0 && count_neg != 0) {
            product = product / negmax;
        }
        return product;
    }
}
```

Explanation:

This Java code calculates the minimum product of a subset from an array of integers. It considers special cases for zero, negative, and positive numbers, keeping track of the count of each. The code then adjusts the product based on the counts, ensuring the minimum product is returned, considering zero, and handling even counts of negative numbers.

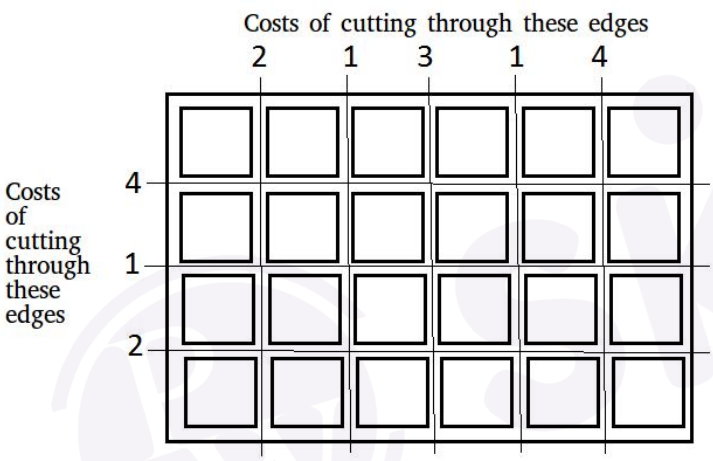
Time Complexity:

O(n) where 'n' is the size of the input array. The function iterates through the array once.

Space Complexity:

O(1) as the function uses a constant amount of extra space, independent of the input size.

Minimum Cost to cut a board into squares



Problem:

A board of length m and width n is given, we need to break this board into m*n squares such that the cost of breaking is minimum. the cutting cost for each edge will be given for the board. In short, we need to choose such a sequence of cutting such that cost is minimized.

Code

```
class Main
{
    static int minimumCostOfBreaking(Integer X[], Integer Y[],
                                                    int m, int n)
    {
        int res = 0;
        Arrays.sort(X, Collections.reverseOrder());
        Arrays.sort(Y, Collections.reverseOrder());
        int hzntl = 1, vert = 1;
        int i = 0, j = 0;
        while (i < m && j < n)
        {
            if (X[i] > Y[j])
            {
                res += X[i] * vert;
                hzntl++;
                i++;
            }
            else
            {
                res += Y[j] * hzntl;
                vert++;
                j++;
            }
        }
        int total = 0;
        while (i < m)
            total += X[i++];
        res += total * vert;
        total = 0;
        while (j < n)
            total += Y[j++];
        res += total * hzntl;

        return res;
    }
}
```

Explanation:

This Java code calculates the minimum cost of breaking a given rectangular grid into individual cells using horizontal and vertical cuts. It sorts the horizontal and vertical cut costs in descending order, iterates through them while adjusting horizontal and vertical counts, and accumulates the total cost. The method then returns the overall minimum cost for the grid decomposition.

Time Complexity:

Sorting: O(m * log(m) + n * log(n))
Traversal: O(m + n)
Therefore, the overall time complexity is O(m * log(m) + n * log(n)).

Space Complexity:

Input arrays: O(m + n)
Additional space: O(1)

Construct String With Repeat Limit [Leetcode - 2182]

Problem:

You are given a string s and an integer repeatLimit. Construct a new string repeatLimitedString using the characters of s such that no letter appears more than repeatLimit times in a row. You do not have to use all characters from s.Return the lexicographically largest repeatLimitedString possible.A string a is lexicographically larger than a string b if in the first position where a and b differ, string a has a letter that appears later in the alphabet than the corresponding letter in b. If the first min(a.length, b.length) characters do not differ, then the longer string is the lexicographically larger one.

Input: s = "cczazcc", repeatLimit = 3
Output: "zzcccac"

Explanation: We use all of the characters from s to construct the repeatLimitedString "zzcccac".
The letter 'a' appears at most 1 time in a row.
The letter 'c' appears at most 3 times in a row.
The letter 'z' appears at most 2 times in a row.
Hence, no letter appears more than repeatLimit times in a row and the string is a valid repeatLimitedString.
The string is the lexicographically largest repeatLimitedString possible so we return "zzcccac".

Note that the string "zzcccca" is lexicographically larger but the letter 'c' appears more than 3 times in a row, so it is not a valid repeatLimitedString.

Code

```
class Main{
    public String repeatLimitedString(String s, int repeatLimit) {
        Map<Character, Integer> map = new HashMap<>();
        for(char c :  s.toCharArray())
            map.put(c, map.getOrDefault(c, 0) + 1);

  Queue<Map.Entry<Character, Integer>> pq = new PriorityQueue<>(
            (a, b) -> Character.compare(b.getKey(), a.getKey())
        );
        pq.addAll(map.entrySet());
        StringBuilder ret = new StringBuilder();
        int pivot = -1;
        while(!pq.isEmpty()) {
            Map.Entry<Character, Integer> node = pq.poll();
            if(pivot == -1 || ret.charAt(pivot) != node.getKey() ||
 ret.length() - pivot < repeatLimit) {
                ret.append(node.getKey());
                node.setValue(node.getValue()-1);
                if(node.getValue()!=0)
                    pq.add(node);
                if(pivot == -1 || ret.charAt(pivot)
!= node.getKey())
                    pivot = ret.length()-1;
                continue;
            }
            if(pq.isEmpty()) return ret.toString();
            Map.Entry<Character, Integer> node1 = pq.poll();
            ret.append(node1.getKey());
            node1.setValue(node1.getValue()-1);
            if(node1.getValue()!=0)
                pq.add(node1);
            pq.add(node);
            pivot = ret.length()-1;
        }
        return ret.toString();
    }
}
```

Explanation:

This Java code defines a method repeatLimitedString that takes a string s and an integer repeatLimit as input. It creates a frequency map of characters in the input string and uses a priority queue to build a string by repeatedly appending characters with the highest frequency while ensuring no more than repeatLimit consecutive characters are added. The method returns the resulting string.

Time Complexity:

Building Frequency Map: O(N) - Loop through each character in the input string.
Building Priority Queue: O(N log N) - Inserting entries into the priority queue.
Building Result String: O(N) - Main loop iterating through characters.
Overall Time Complexity: O(N log N) dominates due to priority queue operations.

Space Complexity:

HashMap (Frequency Map): O(N) - In the worst case, where all characters are unique.
Priority Queue: O(N) - In the worst case, it can have up to N distinct characters.
StringBuilder: O(N) - Proportional to the final output string.
Other Variables: Constant space.
Overall Space Complexity: O(N) considering HashMap, PriorityQueue, StringBuilder, and other variables.

Rabbits in Forest [Leetcode - 78]

Problem:

There is a forest with an unknown number of rabbits. We asked n rabbits "How many rabbits have the same color as you?" and collected the answers in an integer array answers where answers[i] is the answer of the ith rabbit.Given the array answers, return the minimum number of rabbits that could be in the forest.

Code

```
class Main{
    public int numRabbits(int[] answers) {
        HashMap<Integer, Integer> map=new HashMap<>();
        for(int i=0; i<answers.length ; i++){
            if(map.containsKey(answers[i])){
                map.put(answers[i], map.get(answers[i])+1);
            }else{
                map.put(answers[i],1);
            }
        }
        int ans=0;
        for(Integer key : map.keySet()){
            int q = (map.get(key))/(key+1);
            int r = (map.get(key))%(key+1);
            q += r==0 ? 0 : 1;
            ans += q*(key+1);
        }
        return ans;
    }
}
```

Explanation:

This Java code calculates the minimum number of rabbits in a forest based on the answers provided by other rabbits. It uses a HashMap to count the occurrences of each answer. Then, it iterates through the unique answers, calculating the quotient and remainder when dividing the count by (answer + 1). It adds the product of quotient and (answer + 1) to the total, considering the remainder. The final result is the sum of these products, representing the minimum number of rabbits in the forest.

Time Complexity:

Building Frequency Map: O(N)
Iterating Through Map: O(N)
Overall Time Complexity: O(N)

Space Complexity:

HashMap (Frequency Map): O(N)
Other Variables: Constant
Overall Space Complexity: O(N)

Meeting Room II [Leetcode - 253]

Problem:

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

Input: [[0, 30],[5, 10],[15, 20]]
Output: 2

Code

```
class Main{
  public int minMeetingRooms(int[][] intervals) {
    Arrays.sort(intervals, (a, b) → (a[0] - b[0]));
    // Store the end times of each room.
    Queue<Integer> minHeap = new PriorityQueue<>();
    for (int[] interval : intervals) {
      // There's no overlap, so we can reuse the same room.
      if (!minHeap.isEmpty() && interval[0] ⩾ minHeap.peek())
        minHeap.poll();
      minHeap.offer(interval[1]);
    }
    return minHeap.size();
  }
}
```

Explanation:

This Java code calculates the minimum number of meeting rooms required for a given set of intervals. It sorts the intervals based on their start times and uses a min heap to track the end times of active meetings. While iterating through the sorted intervals, it checks if the current interval can reuse an existing room (if its start time is after the end time of the earliest ending meeting) or if a new room is needed. The final result is the size of the min heap, representing the minimum number of meeting rooms required. The time complexity is O(n log n), where n is the number of intervals.

Time Complexity:

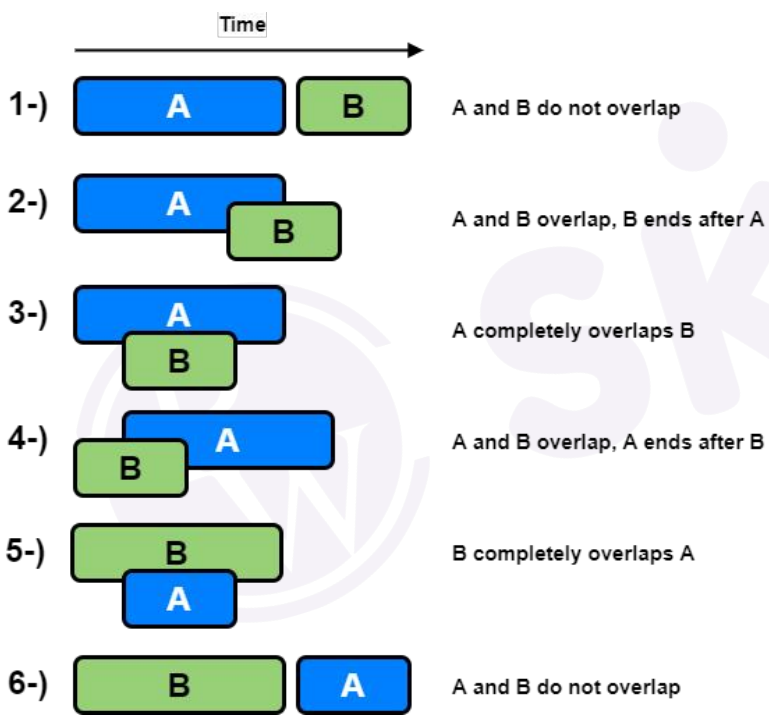
Sorting intervals takes O(n log n) time.
Iterating through intervals takes O(n log n) time due to the priority queue operations.
Overall, the time complexity is O(n log n).

Space Complexity:

The space complexity is O(n) due to the priority queue storing end times.

In summary:
Time Complexity: O(n log n)
Space Complexity: O(n)

Merge Intervals [Leetcode - 56]



Problem:

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

Code

```
class Main{
    public int[][] merge(int[][] intervals) {
        Arrays.sort(intervals,(a,b)→a[0]-b[0]);
        List<int []> l=new ArrayList<>();
        int px=intervals[0][0];
        int py=intervals[0][1];
        l.add(new int[]{px,py});
        for(int i=1;i<intervals.length;i++){
            int curx=intervals[i][0];
            int cury=intervals[i][1];
            if(curx≥px && curx≤py){
                l.remove(l.size()-1);
                l.add(new int []
{Math.min(px,curx),Math.max(py,cury)});
            }else{
               l.add(new int []{curx,cury});
            }
            px=l.get(l.size()-1)[0];
            py=l.get(l.size()-1)[1];

        }

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

    }
}
```

Explanation:

This Java code implements the merging of overlapping intervals in a 2D array. It first sorts the intervals based on their start points. Then, it iterates through the sorted intervals, merging overlapping ones and adding non-overlapping ones to a list. Finally, it converts the list of merged intervals into a 2D array and returns the result. The time complexity is O(n log n) due to the sorting step, where n is the number of intervals.

Time Complexity:

Sorting: O(n log n)
Iterating through intervals: O(n)
The overall time complexity is dominated by sorting, resulting in O(n log n).

Space Complexity:

List l: O(n)
2D array ans: O(n)
The overall space complexity is O(n).

Leetcode 452 || Minimum Number of Arrows to Burst Balloons

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] = [xstart, xend] denotes a balloon whose horizontal diameter stretches between xstart and xend. 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 xstart and xend is burst by an arrow shot at x if xstart <= x <= xend. 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.

Example 1:

Input: points = [[10,16],[2,8],[1,6],[7,12]]
Output: 2

Explanation: The balloons can be burst by 2 arrows:
- Shoot an arrow at x = 6, bursting the balloons [2,8] and [1,6].
- Shoot an arrow at x = 11, bursting the balloons [10,16] and [7,12].

Example 2:

Input: points = [[1,2],[3,4],[5,6],[7,8]]
Output: 4

Explanation: One arrow needs to be shot for each balloon for a total of 4 arrows.

Example 3:
Input: points = [[1,2],[2,3],[3,4],[4,5]]
Output: 2

Explanation: The balloons can be burst by 2 arrows:
- Shoot an arrow at x = 2, bursting the balloons [1,2] and [2,3].
- Shoot an arrow at x = 4, bursting the balloons [3,4] and [4,5].

Constraints:

- 1 <= points.length <= 105
- points[i].length == 2
- -231 <= xstart < xend <= 231 - 1

Approach

Greedy algorithm

1. An arrow shoots through multiple intervals, all of which are coincident. We try to stack the overlapping intervals together and give a jump.
2. To facilitate identifying coincidence in a single traversal, we sort in ascending order on the right

Complexity

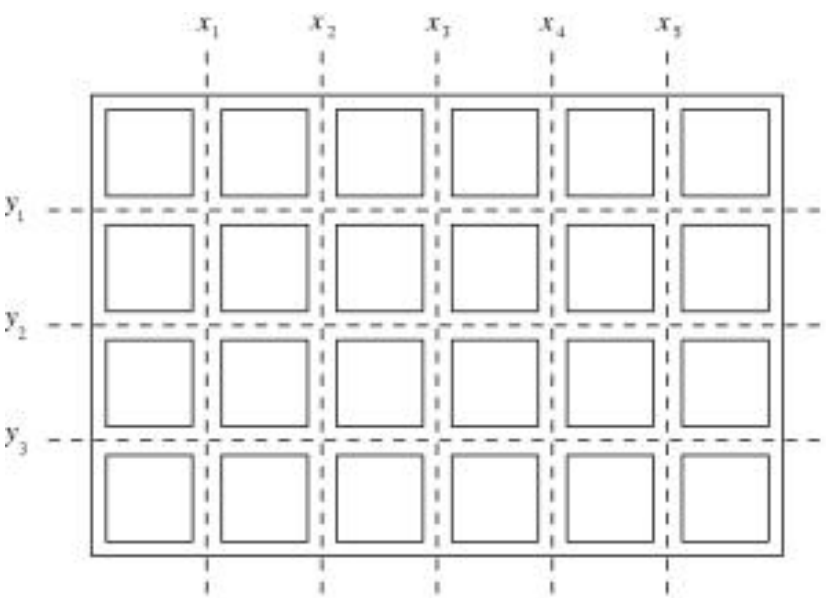
- Time complexity:O(NlogN)
- Space complexity:O(1)

Code

```
public int findMinArrowShots(int[][] points) {
        if (points.length == 0) {
            return 0;
        }
        Arrays.sort(points, (a, b) -> a[1] - b[1]);
        int arrowPos = points[0][1];
        int arrowCnt = 1;
        for (int i = 1; i < points.length; i++) {
            if (arrowPos >= points[i][0]) {
                continue;
            }
            arrowCnt++;
            arrowPos = points[i][1];
        }
        return arrowCnt;
    }
```

SPOJ: Chocola problem

We are given a bar of chocolate composed of m*n square pieces. One should break the chocolate into single squares. Parts of the chocolate may be broken along the vertical and horizontal lines as indicated by the broken lines in the picture.
A single break of a part of the chocolate along a chosen vertical or horizontal line divides that part into two smaller ones. Each break of a part of the chocolate is charged a cost expressed by a positive integer. This cost does not depend on the size of the part that is being broken but only depends on the line the break goes along.
Let us denote the costs of breaking along consecutive vertical lines with x1, x2, ..., xm-1 and along horizontal lines with y1, y2, ..., yn-1.
The cost of breaking the whole bar into single squares is the sum of the successive breaks. One should compute the minimal cost of breaking the whole chocolate into single squares.



For example, if we break the chocolate presented in the picture first along the horizontal lines, and next each obtained part along vertical lines then the cost of that breaking will be y1+y2+y3+4*(x1+x2+x3+x4+x5).
Task

Write a program that for each test case:

- Reads the numbers x1, x2, ..., xm-1 and y1, y2, ..., yn-1
- Computes the minimal cost of breaking the whole chocolate into single squares, writes the result.

Input
One integer in the first line, stating the number of test cases, followed by a blank line. There will be not more than 20 tests.
For each test case, at the first line there are two positive integers m and n separated by a single space, 2 <= m,n <= 1000. In the successive m-1 lines there are numbers x1, x2, ..., xm-1, one per line, 1 <= xi <= 1000. In the successive n-1 lines there are numbers y1, y2, ..., yn-1, one per line, 1 <= yi <= 1000.
The test cases will be separated by a single blank line.
Output
For each test case : write one integer - the minimal cost of breaking the whole chocolate into single squares.

Example

Input:
1
6 4
2
1
3
1
4
4
1
2

Output:
42

APPROACH:

1. Sort Costs:
   - Sort the horizontal and vertical costs in descending order. This allows us to consider the highest costs first during the process.
2. Initialize Variables
   - Initialize variables to keep track of horizontal and vertical parts (initially set to 1) and the total cost (initially set to 0).
3. Iterate Through Costs:
   - Use two pointers, one for the horizontal cost array and one for the vertical cost array.
   - Compare the costs at the current pointers.
   - If the horizontal cost is greater, add the cost to the total multiplied by the current vertical parts.
   - Increment the horizontal parts.
   - Move the horizontal pointer.
   - If the vertical cost is greater, add the cost to the total multiplied by the current horizontal parts.
   - Increment the vertical parts.
   - Move the vertical pointer.
   - Repeat until one of the arrays is fully processed.

4. Process Remaining Costs:
   - If there are remaining elements in the horizontal array, add their cost multiplied by the current vertical parts to the total.
   - If there are remaining elements in the vertical array, add their cost multiplied by the current horizontal parts to the total.
5. Return Result:
   - The total cost obtained represents the minimum cost to break the board into m x n squares.

Code

```
import java.util.Arrays;
import java.util.Collections;
public class MinimumCostOfBreakingBoard {
    // Function to calculate the minimum cost to break the board
into m * n squares
    static int minimumCostOfBreaking(int[] horizontalCost, int[]
verticalCost, int m, int n) {
        int totalCost = 0;
        // Sort the horizontal cost array in descending order
        Arrays.sort(horizontalCost);
        reverseArray(horizontalCost);
        // Sort the vertical cost array in descending order
        Arrays.sort(verticalCost);
        reverseArray(verticalCost);
        int horizontalParts = 1, verticalParts = 1; // Initialize
current width as 1
        int i = 0, j = 0;
        while (i < m && j < n) {
            if (horizontalCost[i] > verticalCost[j]) {
                // Add cost of the current horizontal part
                totalCost += horizontalCost[i] * verticalParts;
                // Increase the current horizontal part count by 1
                horizontalParts++;
                i++;
            } else {
                // Add cost of the current vertical part
                totalCost += verticalCost[j] * horizontalParts;
                // Increase the current vertical part count by 1
                verticalParts++;
                j++;
            }
        }
        // Loop for the remaining horizontal array
        int remainingHorizontalCost = 0;
        while (i < m)
            remainingHorizontalCost += horizontalCost[i++];
        totalCost += remainingHorizontalCost * verticalParts;
        // Loop for the remaining vertical array
        int remainingVerticalCost = 0;
        while (j < n)
            remainingVerticalCost += verticalCost[j++];
        totalCost += remainingVerticalCost * horizontalParts;
        return totalCost;
    }
    // Function to reverse an array
    static void reverseArray(int[] arr) {
        int start = 0;
        int end = arr.length - 1;
        while (start < end) {
            int temp = arr[start];
            arr[start] = arr[end];
            arr[end] = temp;
            start++;
            end--;
        }
    }
    public static void main(String[] args) {
        int[] horizontalCost = {2, 1, 3, 1, 4};
        int[] verticalCost = {4, 1, 2};
        int m = horizontalCost.length;
        int n = verticalCost.length;
        System.out.println(minimumCostOfBreaking(horizontalCost,
verticalCost, m, n));
    }
}
```

Code

```
public int findMinArrowShots(int[][] points) {
        if (points.length == 0) {
            return 0;
        }
        Arrays.sort(points, (a, b) -> a[1] - b[1]);
        int arrowPos = points[0][1];
        int arrowCnt = 1;
        for (int i = 1; i < points.length; i++) {
            if (arrowPos >= points[i][0]) {
                continue;
            }
            arrowCnt++;
            arrowPos = points[i][1];
        }
        return arrowCnt;
    }
```

Activity selection problem

The activity selection problem is a mathematical optimization problem. Our first illustration is the problem of scheduling a resource among several challenge activities. We find a greedy algorithm provides a well designed and simple method for selecting a maximum- size set of manually compatible activities.

Suppose S = {1, 2....n} is the set of n proposed activities. The activities share resources which can be used by only one activity at a time, e.g., Tennis Court, Lecture Hall, etc. Each Activity "i" has start time si and a finish time fi, where si ≤fi. If selected activity "i" take place meanwhile the half-open time interval [si,fi). Activities i and j are compatible if the intervals (si, fi) and [si, fi) do not overlap (i.e. i and j are compatible if si ≥fi or si ≥fi). The activity-selection problem chosen the maximum- size set of mutually consistent activities.

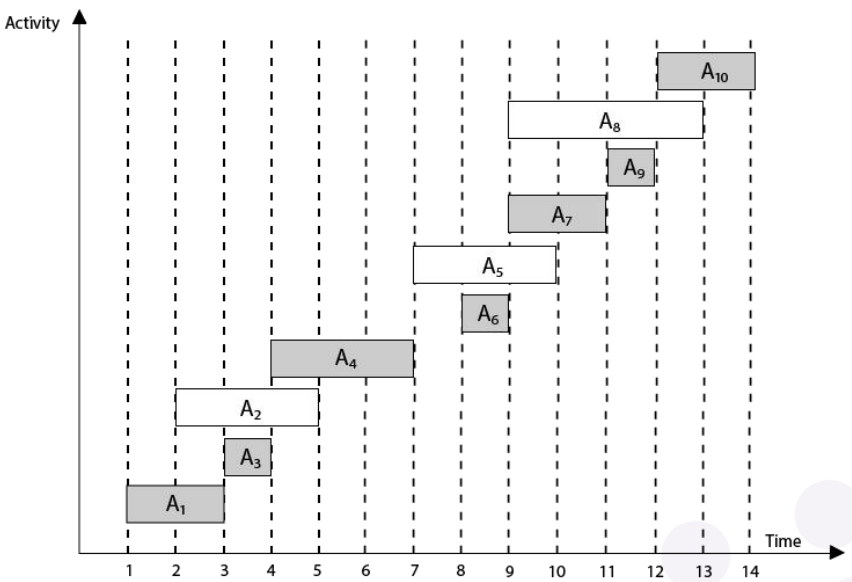
Algorithm Of Greedy- Activity Selector:

GREEDY- ACTIVITY SELECTOR (s, f)
1. n ← length [s]
2. A ← {1}
3. j ← 1.
4. for i ← 2 to n
5. do if si ≥ fi
6. then A ← A {i}
7. j ← i
8. return A

Example: Given 10 activities along with their start and end time as
S = (A1 A2 A3 A4 A5 A6 A7 A8 A9 A10)
Si = (1,2,3,4,7,8,9,9,11,12)
fi = (3,5,4,7,10,9,11,13,12,14)

Solution:

The solution to the above Activity scheduling problem using a greedy strategy is illustrated below:
Arranging the activities in increasing order of end time



Now, schedule A1
Next schedule A3 as A1 and A3 are non-interfering.
Next skip A2 as it is interfering.
Next, schedule A4 as A1 A3 and A4 are non-interfering, then next, schedule A6 as A1 A3 A4 and A6 are non-interfering.
Skip A5 as it is interfering.
Next, schedule A7 as A1 A3 A4 A6 and A7 are non-interfering.
Next, schedule A9 as A1 A3 A4 A6 A7 and A9 are non-interfering.
Skip A8 as it is interfering.
Next, schedule A10 as A1 A3 A4 A6 A7 A9 and A10 are non-interfering.
Thus the final Activity schedule is:

$$(A_1\ A_3\ A_4\ A_6\ A_7\ A_9\ A_{10})$$

Leetcode 480 || Sliding Window Median

The median is the middle value in an ordered integer list. If the size of the list is even, there is no middle value. So the median is the mean of the two middle values.

- For examples, if arr = [2,3,4], the median is 3.
- For examples, if arr = [1,2,3,4], the median is (2 + 3) / 2 = 2.5.

You are given an integer array nums and an integer k. There is a sliding window of size k which is moving from the very left of the array to the very right. You can only see the k numbers in the window. Each time the sliding window moves right by one position.
Return the median array for each window in the original array. Answers within 10-5 of the actual value will be accepted.

Example 1:

Input: nums = [1,3,-1,-3,5,3,6,7], k = 3
Output: [1.00000,-1.00000,-1.00000,3.00000,5.00000,6.00000]

Explanation:

Window position	Median
[1 3 -1] -3 5 3 6 7	1
1 [3 -1 -3] 5 3 6 7	-1
1 3 [-1 -3 5] 3 6 7	-1
1 3 -1 [-3 5 3] 6 7	3
1 3 -1 -3 [5 3 6] 7	5
1 3 -1 -3 5 [3 6 7]	6

Example 2:

Input: nums = [1,2,3,4,2,3,1,4,2], k = 3
Output: [2.00000,3.00000,3.00000,3.00000,2.00000,3.00000,2.00000]

Constraints:

- 1 <= k <= nums.length <= 105
- -231 <= nums[i] <= 231 - 1

Intuition

The medianSlidingWindow function maintains a sorted window of elements using vector operations and calculates medians as the window slides through the input array. It utilizes efficient insertions and deletions with lower_bound and updates the result vector with calculated medians. The final result vector contains medians for each sliding window.

Approach

1. Initialization: The function starts by initializing the result vector res, indices i and j to manage the sliding window, and a temporary vector temp with the first k-1 elements of the input vector nums. This temporary vector is sorted to represent the initial sliding window.
2. Sliding Window: The main loop runs while the end index j is within the range of the input vector nums. In each iteration:
    - The current element (nums[j]) is inserted into the sorted window temp at the correct position using lower_bound.
    - The median of the current window is calculated based on the size of the window k.
    - If k is odd, the median is the middle element; otherwise, it is the average of the middle two elements.
    - The median is then pushed into the result vector res.
3. Updating Window: After calculating the median, the oldest element in the window (nums[i]) is removed using erase and lower_bound.
    - The indices i and j are incremented to slide the window to the next position.
4. Final Result: The function returns the result vector res containing the medians for each sliding window.

Complexity

Time Complexity:

The time complexity of the medianSlidingWindow function is primarily dominated by the operations inside the main loop. Let (n) be the length of the input array nums.

1. Sorting the initial window (temp): (O(k \log k)) - Sorting a window of size (k-1).
2. Main Loop: (O(n \log k)) - Iterating through the array and performing insertion and deletion operations on the sorted window.

Therefore, the overall time complexity is (O(n \log k)).

Space Complexity:

The space complexity is determined by the space used to store the temporary window (temp) and the result vector (res).

1. Temporary Window (temp): (O(k)) - The size of the sorted window.
2. Result Vector (res): (O(n)) - In the worst case, the result vector contains one median for each window.

Therefore, the overall space complexity is (O(k + n)).

Code

```
public class Solution {
    PriorityQueue<Integer> minHeap = new PriorityQueue<Integer>();
    PriorityQueue<Integer> maxHeap = new PriorityQueue<Integer>(
        new Comparator<Integer>() {
            public int compare(Integer i1, Integer i2) {
                return i2.compareTo(i1);
            }
        }
    );

    public double[] medianSlidingWindow(int[] nums, int k) {
        int n = nums.length - k + 1;
        if (n <= 0) return new double[0];
        double[] result = new double[n];

        for (int i = 0; i <= nums.length; i++) {
            if (i >= k) {
                result[i - k] = getMedian();
                remove(nums[i - k]);
            }
            if (i < nums.length) {
                add(nums[i]);
            }
        }

        return result;
    }

    private void add(int num) {
        if (num < getMedian()) {
            maxHeap.add(num);
        }
        else {
            minHeap.add(num);
        }
        if (maxHeap.size() > minHeap.size()) {
            minHeap.add(maxHeap.poll());
        }
        if (minHeap.size() - maxHeap.size() > 1) {
            maxHeap.add(minHeap.poll());
        }
    }

    private void remove(int num) {
        if (num < getMedian()) {
            maxHeap.remove(num);
        }
        else {
            minHeap.remove(num);
        }
        if (maxHeap.size() > minHeap.size()) {
            minHeap.add(maxHeap.poll());
        }
        if (minHeap.size() - maxHeap.size() > 1) {
            maxHeap.add(minHeap.poll());
        }
    }

    private double getMedian() {
        if (maxHeap.isEmpty() && minHeap.isEmpty()) return 0;

        if (maxHeap.size() == minHeap.size()) {
            return ((double)maxHeap.peek() +
(double)minHeap.peek()) / 2.0;
        }
        else {
            return (double)minHeap.peek();
        }
    }
}
```

PW SKILLS

THANK YOU !