**Arrays**

**1) Majority Element (Moore’s Voting Algorithm) (Leetcode - 169)**

Given an array nums of size n, return the majority element.

The majority element is the element that appears more than ⌊n / 2⌋ times. You may assume that the majority element always exists in the array.

Example 1:

Input: nums = [3,2,3]

Output: 3

Example 2:

Input: nums = [2,2,1,1,1,2,2]

Output: 2

Constraints:

n == nums.length

1 <= n <= 5 \* 104

-109 <= nums[i] <= 109

**Solution :**

class Solution {

    public int majorityElement(int[] nums) {

        int ele = 0, count = 0;

        for(int n: nums) {

            if(count == 0)

                ele = n;

            count += (n == ele)? 1: -1;

        }

        return ele;

    }

}

**2) Merge Intervals (Leetcode - 56)**

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.

Example 1:

Input: intervals = [[1,3],[2,6],[8,10],[15,18]]

Output: [[1,6],[8,10],[15,18]]

Explanation: Since intervals [1,3] and [2,6] overlap, merge them into [1,6].

Example 2:

Input: intervals = [[1,4],[4,5]]

Output: [[1,5]]

Explanation: Intervals [1,4] and [4,5] are considered overlapping.

Constraints:

1 <= intervals.length <= 104

intervals[i].length == 2

0 <= starti <= endi <= 104

**Solution :**

class Solution {

    public int[][] merge(int[][] intervals) {

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

        Arrays.sort(intervals,(a,b)->a[0]-b[0]);

        res.add(intervals[0]);

        for(int i=1;i<intervals.length;i++)

        {

            if(intervals[i][0]<=res.get(res.size()-1)[1])

            {

                res.get(res.size()-1)[1]=Math.max(res.get(res.size()-1)[1],intervals[i][1]);

            }

            else

            {

                res.add(intervals[i]);

            }

        }

        return res.toArray(new int[res.size()][2]);

    }

}

**3) Group Anagrams (Leetcode - 49)**

Given an array of strings strs, group the anagrams together. You can return the answer in any order.

An Anagram is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.

Example 1:

Input: strs = ["eat","tea","tan","ate","nat","bat"]

Output: [["bat"],["nat","tan"],["ate","eat","tea"]]

Example 2:

Input: strs = [""]

Output: [[""]]

Example 3:

Input: strs = ["a"]

Output: [["a"]]

Constraints:

1 <= strs.length <= 104

0 <= strs[i].length <= 100

strs[i] consists of lowercase English letters.

**Solution :**

class Solution {

public List<List<String>> groupAnagrams(String[] strs) {

// Create a HashMap to store the grouped anagrams

HashMap<String, List<String>> map = new HashMap<>();

// Iterate over each string in the input array

for (int i = 0; i < strs.length; i++) {

// Create a count array to store the frequency of each character

int count[] = new int[26];

for (int j = 0; j < strs[i].length(); j++) {

// Increment the count for the current character

count[strs[i].charAt(j) - 'a']++;

}

// Build a key from the character count array

StringBuilder s = new StringBuilder();

for (int j = 0; j < 26; j++) {

// Append the count of each character to the key

s.append(count[j]);

}

// Convert the StringBuilder to a String key

String str = s.toString();

// If the key is not present in the map, add it with a new list

if (!map.containsKey(str)) {

map.put(str, new ArrayList<>());

}

// Add the original string to the list corresponding to the key

map.get(str).add(strs[i]);

}

// Return all the grouped anagrams as a list of lists

return new ArrayList<>(map.values());

}

}

**4) Valid Anagram (Leetcode - 242)**

Given two strings s and t, return true if t is an anagram of s, and false otherwise.

An Anagram is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.

Example 1:

Input: s = "anagram", t = "nagaram"

Output: true

Example 2:

Input: s = "rat", t = "car"

Output: false

Constraints:

1 <= s.length, t.length <= 5 \* 104

s and t consist of lowercase English letters.

**Solution :**

class Solution {

    public boolean isAnagram(String s, String t) {

        int count[] = new int[26];

        for(int i=0;i<s.length();i++)

        {

            count[s.charAt(i)-'a']++;

        }

        for(int i=0;i<t.length();i++)

        {

            count[t.charAt(i)-'a']--;

        }

        for(int i=0;i<26;i++)

        {

            if(count[i]!=0)

            {

                return false;

            }

        }

        return true;

    }

}

**5) Longest Consecutive Sequence (Leetcode - 128)**

Given an unsorted array of integers nums, return the length of the longest consecutive elements sequence.

You must write an algorithm that runs in O(n) time.

Example 1:

Input: nums = [100,4,200,1,3,2]

Output: 4

Explanation: The longest consecutive elements sequence is [1, 2, 3, 4]. Therefore its length is 4.

Example 2:

Input: nums = [0,3,7,2,5,8,4,6,0,1]

Output: 9

Constraints:

0 <= nums.length <= 105

-10^9 <= nums[i] <= 10^9

**Solution : TC - o(n)**

class Solution {

    public int longestConsecutive(int[] nums) {

        HashSet<Integer> set = new HashSet<>();

        for(int i:nums)

        {

            set.add(i);

        }

        int max=0;

        for(int i:nums)

        {

            int prev =i-1;

            int count=1;

            if(!set.contains(prev))

            {

                int cur = i+1;

                while(set.contains(cur))

                {

                    count++;

                    cur++;

                }

            }

            max=Math.max(count,max);

        }

        return max;

    }

}

**Greedy**

**1) Lemonade Change (Leetcode - 860)**

At a lemonade stand, each lemonade costs $5. Customers are standing in a queue to buy from you and order one at a time (in the order specified by bills). Each customer will only buy one lemonade and pay with either a $5, $10, or $20 bill. You must provide the correct change to each customer so that the net transaction is that the customer pays $5.

Note that you do not have any change in hand at first.

Given an integer array bills where bills[i] is the bill the ith customer pays, return true if you can provide every customer with the correct change, or false otherwise.

Example 1:

Input: bills = [5,5,5,10,20]

Output: true

Explanation:

From the first 3 customers, we collect three $5 bills in order.

From the fourth customer, we collect a $10 bill and give back a $5.

From the fifth customer, we give a $10 bill and a $5 bill.

Since all customers got correct change, we output true.

Example 2:

Input: bills = [5,5,10,10,20]

Output: false

Explanation:

From the first two customers in order, we collect two $5 bills.

For the next two customers in order, we collect a $10 bill and give back a $5 bill.

For the last customer, we can not give the change of $15 back because we only have two $10 bills.

Since not every customer received the correct change, the answer is false.

Constraints:

1 <= bills.length <= 105

bills[i] is either 5, 10, or 20.

**Solution :**

class Solution {

    public boolean lemonadeChange(int[] bills) {

        int five = 0,ten = 0;

        for(int i : bills)

        {

            if(i==5)

            {

                five++;

            }

            else if(i==10)

            {

                if(five>0)

                {

                    five--;

                    ten++;

                }

                else

                {

                    return false;

                }

            }

            else

            {

                if(five > 0 && ten> 0)

                {

                    five--;

                    ten--;

                }

                else if(five >= 3)

                {

                    five-=3;

                }

                else

                {

                    return false;

                }

            }

        }

        return true;

    }

}

**2) Jump Game (Leetcode - 55)**

You are given an integer array nums. You are initially positioned at the array's first index, and each element in the array represents your maximum jump length at that position.

Return true if you can reach the last index, or false otherwise.

Example 1:

Input: nums = [2,3,1,1,4]

Output: true

Explanation: Jump 1 step from index 0 to 1, then 3 steps to the last index.

Example 2:

Input: nums = [3,2,1,0,4]

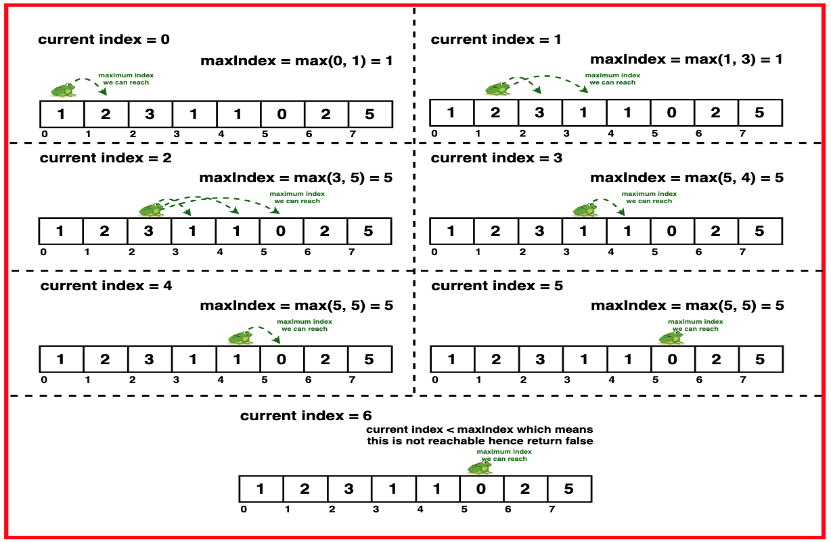
Output: false

Explanation: You will always arrive at index 3 no matter what. Its maximum jump length is 0, which makes it impossible to reach the last index.

Constraints:

1 <= nums.length <= 10^4

0 <= nums[i] <= 10^5



**Solution :**

class Solution {

    //o(n\*n)

    public boolean canJump(int i,int[] nums,int[] dp)

    {

        if(dp[i]!=0) return (dp[i]==1)?true:false;

        if(i==nums.length-1)

        {

            return true;

        }

        for(int j=1;j<=nums[i];j++)

        {

            if(canJump(i+j,nums,dp))

            {

                dp[i]=1;

                return true;

            }

        }

        dp[i]=2;

        return false;

    }

    //o(n\*n)

    public boolean tab(int[] nums)

    {

        int n = nums.length-1;

        boolean dp[] = new boolean[nums.length];

        dp[n] = true;

        for(int i=n-1;i>=0;i--)

        {

            for(int j=1;j<=nums[i];j++)

            {

                if(dp[i+j])

                {

                    dp[i]=true;

                    break;

                }

            }

        }

        return dp[0];

    }

    public boolean greedy(int[] nums)

    {

        int maxInd = 0;

        for(int i=0;i<nums.length;i++)

        {

            if(maxInd<i) return false;

            maxInd = Math.max(maxInd,i+nums[i]);

        }

        return true;

    }

    public boolean canJump(int[] nums)

    {

       return greedy(nums);

    }

}

**3) Jump Game II (Leetcode - 45)**

You are given a 0-indexed array of integers nums of length n. You are initially positioned at nums[0].

Each element nums[i] represents the maximum length of a forward jump from index i. In other words, if you are at nums[i], you can jump to any nums[i + j] where:

0 <= j <= nums[i] and

i + j < n

Return the minimum number of jumps to reach nums[n - 1]. The test cases are generated such that you can reach nums[n - 1].

Example 1:

Input: nums = [2,3,1,1,4]

Output: 2

Explanation: The minimum number of jumps to reach the last index is 2. Jump 1 step from index 0 to 1, then 3 steps to the last index.

Example 2:

Input: nums = [2,3,0,1,4]

Output: 2

Constraints:

1 <= nums.length <= 104

0 <= nums[i] <= 1000

It's guaranteed that you can reach nums[n - 1].

**Solutions :**

class Solution {

    //O(N\*N)

    public int canJump(int i,int[] nums,int[] dp)

    {

        if(i>=nums.length-1)

        {

            return 0;

        }

        if(dp[i]!=Integer.MAX\_VALUE) return dp[i];

        int steps=(int) 1e9;

        for(int j=1;j<=nums[i];j++)

        {

            int step=1+canJump(i+j,nums,dp);

            steps=Math.min(steps,step);

        }

        dp[i]=steps;

        return dp[i];

    }

    //O(n) greedy

    public int jump(int[] nums) {

        int jumps,l,r;

        jumps=l=r=0;

        while(r<nums.length-1)

        {

            int farthest=0;

            for(int i=l;i<=r;i++)

            {

                farthest = Math.max(farthest,i+nums[i]);

            }

            jumps++;

            r=farthest;

            l=l+1;

        }

        return jumps;

    }

}

**4) Job Sequencing Problem**

**Problem Statement:** You are given a set of N jobs where each job comes with a**deadline** and **profit**. The profit can only be earned upon completing the job within its deadline. Find the **number of jobs** done and the **maximum profit** that can be obtained. Each job takes a **single unit**of time and only **one job** can be performed at a time.

**Examples**

**Example 1:**

**Input:** N = 4, Jobs = {(1,4,20),(2,1,10),(3,1,40),(4,1,30)}

**Output**: 2 60

**Explanation:** The 3rd job with a deadline 1 is performed during the first unit of time .The 1st job is performed during the second unit of time as its deadline is 4.

Profit = 40 + 20 = 60

**Example 2:**

**Input:** N = 5, Jobs = {(1,2,100),(2,1,19),(3,2,27),(4,1,25),(5,1,15)}

**Output:** 2 127

**Explanation:** The first and third job both having a deadline 2 give the highest profit.

Profit = 100 + 27 = 127

**Solution**

***Disclaimer***: *Don't jump directly to the solution, try it out yourself first.*

**Approach**:  The strategy to maximize profit should be to pick up jobs that offer**higher profits.**Hence we should **sort** the jobs in descending order of profit. Now say if a job has a deadline of 4 we can perform it anytime between day 1-4, but it is preferable to perform the job on its **last day**. This leaves enough empty slots on the previous days to perform other jobs.

Basic Outline of the approach:-

* Sort the jobs in descending order of profit.
* If the maximum deadline is x, make an array of size x .Each array index is set to -1 initially as no jobs have been performed yet.
* For every job check if it can be performed on its last day.
* If possible mark that index with the job id and add the profit to our answer.
* If not possible, loop through the previous indexes until an empty slot is found.

**Solution :**

class Job {

int id, profit, deadline;

Job(int x, int y, int z) {

this.id = x;

this.deadline = y;

this.profit = z;

}

}

class solve {

*// return an array of size 2 having the 0th element equal to the count*

*// and 1st element equal to the maximum profit*

int[] JobScheduling(Job arr[], int n) {

Arrays.sort(arr, (a, b) -> (b.profit - a.profit));

int maxi = 0;

for (int i = 0; i < n; i++) {

if (arr[i].deadline > maxi) {

maxi = arr[i].deadline;

}

}

int result[] = new int[maxi + 1];

for (int i = 1; i <= maxi; i++) {

result[i] = -1;

}

int countJobs = 0, jobProfit = 0;

for (int i = 0; i < n; i++) {

for (int j = arr[i].deadline; j > 0; j--) {

*// Free slot found*

if (result[j] == -1) {

result[j] = i;

countJobs++;

jobProfit += arr[i].profit;

break;

}

}

}

int ans[] = new int[2];

ans[0] = countJobs;

ans[1] = jobProfit;

return ans;

}

}

**5) Maximum Length of Pair Chain (Leetcode - 646)**

You are given an array of n pairs pairs where pairs[i] = [lefti, righti] and lefti < righti.

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.

Example 1:

Input: pairs = [[1,2],[2,3],[3,4]]

Output: 2

Explanation: The longest chain is [1,2] -> [3,4].

Example 2:

Input: pairs = [[1,2],[7,8],[4,5]]

Output: 3

Explanation: The longest chain is [1,2] -> [4,5] -> [7,8].

Constraints:

n == pairs.length

1 <= n <= 1000

-1000 <= lefti < righti <= 1000

**Solution :( N Meeting One ROOM)**class Solution {

    public int findLongestChain(int[][] pairs) {

        //sort based on right

        Arrays.sort(pairs,(a,b)->a[1]-b[1]);

        int count=1;

        //to store the previous one

        int prevMeet = 0;

        for(int i=1;i<pairs.length;i++)

        {

            if(pairs[i][0]>pairs[prevMeet][1])

            {

                count++;

                prevMeet=i;

            }

        }

        return count;

    }

}