**Binary Search**

**1) Count Pairs Whose Sum is Less than Target (Leetcode - 2824) (Easy)**

Given a 0-indexed integer array nums of length n and an integer target, return the number of pairs (i, j) where 0 <= i < j < n and nums[i] + nums[j] < target.

Example 1:

Input: nums = [-1,1,2,3,1], target = 2

Output: 3

Explanation: There are 3 pairs of indices that satisfy the conditions in the statement:

- (0, 1) since 0 < 1 and nums[0] + nums[1] = 0 < target

- (0, 2) since 0 < 2 and nums[0] + nums[2] = 1 < target

- (0, 4) since 0 < 4 and nums[0] + nums[4] = 0 < target

Note that (0, 3) is not counted since nums[0] + nums[3] is not strictly less than the target.

Example 2:

Input: nums = [-6,2,5,-2,-7,-1,3], target = -2

Output: 10

Explanation: There are 10 pairs of indices that satisfy the conditions in the statement:

- (0, 1) since 0 < 1 and nums[0] + nums[1] = -4 < target

- (0, 3) since 0 < 3 and nums[0] + nums[3] = -8 < target

- (0, 4) since 0 < 4 and nums[0] + nums[4] = -13 < target

- (0, 5) since 0 < 5 and nums[0] + nums[5] = -7 < target

- (0, 6) since 0 < 6 and nums[0] + nums[6] = -3 < target

- (1, 4) since 1 < 4 and nums[1] + nums[4] = -5 < target

- (3, 4) since 3 < 4 and nums[3] + nums[4] = -9 < target

- (3, 5) since 3 < 5 and nums[3] + nums[5] = -3 < target

- (4, 5) since 4 < 5 and nums[4] + nums[5] = -8 < target

- (4, 6) since 4 < 6 and nums[4] + nums[6] = -4 < target

Constraints:

1 <= nums.length == n <= 50

-50 <= nums[i], target <= 50

**Solution :**

class Solution {

    public int countPairs(List<Integer> nums, int target) {

        Collections.sort(nums);

        int count=0;

        int l=0;

        int h=nums.size()-1;

        while(l<h)

        {

            int sum=nums.get(l)+nums.get(h);

            System.out.println(l);

            if(sum<target)

            {

                count+=h-l;

                l++;

            }

            else

            {

                h--;

            }

        }

        return count;

    }

}

**2) Find Target Indices After Sorting Array (Leetcode - 2089)**

You are given a 0-indexed integer array nums and a target element target.A target index is an index i such that nums[i] == target.Return a list of the target indices of nums after sorting nums in non-decreasing order. If there are no target indices, return an empty list. The returned list must be sorted in increasing order.

Example 1:

Input: nums = [1,2,5,2,3], target = 2

Output: [1,2]

Explanation: After sorting, nums is [1,2,2,3,5].

The indices where nums[i] == 2 are 1 and 2.

Example 2:

Input: nums = [1,2,5,2,3], target = 3

Output: [3]

Explanation: After sorting, nums is [1,2,2,3,5].

The index where nums[i] == 3 is 3.

Example 3:

Input: nums = [1,2,5,2,3], target = 5

Output: [4]

Explanation: After sorting, nums is [1,2,2,3,5].

The index where nums[i] == 5 is 4.

Constraints:

1 <= nums.length <= 100

1 <= nums[i], target <= 100

**Solution :**

class Solution {

     public static int binaryLow(int[] a,int l,int h,int ele)

    {

        int temp=-1;

        while(l<=h)

        {

            int mid=(l+h)/2;

            if(a[mid]<ele)

            {

                l=mid+1;

            }

            else if(a[mid]>=ele)

            {

                temp=mid;

                h=mid-1;

            }

        }

        return temp;

    }

    public static int binaryHigh(int[] a,int l,int h,int ele)

    {

        int temp=-1;

        while(l<=h)

        {

            int mid=(l+h)/2;

            if(a[mid]<=ele)

            {

                temp=mid;

                l=mid+1;

            }

            else if(a[mid]>ele)

            {

                h=mid-1;

            }

        }

        return temp;

    }

    public List<Integer> targetIndices(int[] nums, int target) {

        Arrays.sort(nums);

        int n1=binaryLow(nums,0,nums.length-1,target);

        int n2=binaryHigh(nums,0,nums.length-1,target);

        List<Integer> l=new ArrayList<>();

        if (n1 == -1 || n2 == -1) {

            return l;

        }

        for(int i=n1;i<=n2;i++)

        {

            l.add(i);

        }

        return l;

    }

}

**3) Longest Subsequence With Limited Sum (Leetcode - 2389)**

You are given an integer array nums of length n, and an integer array queries of length m.

Return an array answer of length m where answer[i] is the maximum size of a subsequence that you can take from nums such that the sum of its elements is less than or equal to queries[i].

A subsequence is an array that can be derived from another array by deleting some or no elements without changing the order of the remaining elements.

Example 1:

Input: nums = [4,5,2,1], queries = [3,10,21]

Output: [2,3,4]

Explanation: We answer the queries as follows:

- The subsequence [2,1] has a sum less than or equal to 3. It can be proven that 2 is the maximum size of such a subsequence, so answer[0] = 2.

- The subsequence [4,5,1] has a sum less than or equal to 10. It can be proven that 3 is the maximum size of such a subsequence, so answer[1] = 3.

- The subsequence [4,5,2,1] has a sum less than or equal to 21. It can be proven that 4 is the maximum size of such a subsequence, so answer[2] = 4.

Example 2:

Input: nums = [2,3,4,5], queries = [1]

Output: [0]

Explanation: The empty subsequence is the only subsequence that has a sum less than or equal to 1, so answer[0] = 0.

Constraints:

n == nums.length

m == queries.length

1 <= n, m <= 1000

1 <= nums[i], queries[i] <= 106

**Solution :**

class Solution {

    private static int floorBS(int[] a,int k)

    {

        int start=0;

        int end=a.length-1;

        int ans=0;

        while(start<=end)

        {

            int mid=start+(end-start)/2;

            if(a[mid]<=k)

            {

                ans=mid+1;

                start=mid+1;

            }

            else

            {

                end=mid-1;

            }

        }

        return ans;

    }

    public int[] answerQueries(int[] nums, int[] queries) {

        Arrays.sort(nums);

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

        {

            nums[i]+=nums[i-1];

        }

        System.out.println(Arrays.toString(nums));

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

        {

            queries[i]=floorBS(nums,queries[i]);

        }

        return queries;

    }

}

**4) Missing Number (Leetcode - 268)**

Given an array nums containing n distinct numbers in the range [0, n], return the only number in the range that is missing from the array.

Example 1:

Input: nums = [3,0,1]

Output: 2

Explanation: n = 3 since there are 3 numbers, so all numbers are in the range [0,3]. 2 is the missing number in the range since it does not appear in nums.

Example 2:

Input: nums = [0,1]

Output: 2

Explanation: n = 2 since there are 2 numbers, so all numbers are in the range [0,2]. 2 is the missing number in the range since it does not appear in nums.

Example 3:

Input: nums = [9,6,4,2,3,5,7,0,1]

Output: 8

Explanation: n = 9 since there are 9 numbers, so all numbers are in the range [0,9]. 8 is the missing number in the range since it does not appear in nums.

Constraints:

n == nums.length

1 <= n <= 104

0 <= nums[i] <= n

All the numbers of nums are unique.

**Solution : (Cyclic sort)**

class Solution {

    public int missingNumber(int[] a) {

        int i=0;

        while(i<a.length)

        {

            int correct=a[i];

            if(a[i]<=a.length-1 && a[i]!=i)

            {

                int temp=a[i];

                a[i]=a[correct];

                a[correct]=temp;

            }

            else

            {

                i++;

            }

        }

        for(i=0;i<a.length;i++)

        {

            if(i!=a[i])

            {

                return i;

            }

        }

    return a.length;

    }

}

**5) Kth Missing Positive Number (Leetcode - 1539)**

Given an array arr of positive integers sorted in a strictly increasing order, and an integer k.

Return the kth positive integer that is missing from this array.

Example 1:

Input: arr = [2,3,4,7,11], k = 5

Output: 9

Explanation: The missing positive integers are [1,5,6,8,9,10,12,13,...]. The 5th missing positive integer is 9.

Example 2:

Input: arr = [1,2,3,4], k = 2

Output: 6

Explanation: The missing positive integers are [5,6,7,...]. The 2nd missing positive integer is 6.

Constraints:

1 <= arr.length <= 1000

1 <= arr[i] <= 1000

1 <= k <= 1000

arr[i] < arr[j] for 1 <= i < j <= arr.length

**Solution :**

//missing elements at mid = arr[mid]-mid-1

class Solution {

    public int findKthPositive(int[] arr, int k) {

        int start=0,end=arr.length-1;

        while(start<=end)

        {

            int mid=start+(end-start)/2;

            if(arr[mid]-mid-1 < k)

            {

                start=mid+1;

            }

            else

            {

                end=mid-1;

            }

        }

        System.out.println(start+" "+end);

        return start+k;

    }

}

//take u forward

**6) Guess Number Higher or Lower (Leetcode - 374)**

We are playing the Guess Game. The game is as follows:

I pick a number from 1 to n. You have to guess which number I picked.

Every time you guess wrong, I will tell you whether the number I picked is higher or lower than your guess.

You call a pre-defined API int guess(int num), which returns three possible results:

-1: Your guess is higher than the number I picked (i.e. num > pick).

1: Your guess is lower than the number I picked (i.e. num < pick).

0: your guess is equal to the number I picked (i.e. num == pick).

Return the number that I picked.

Example 1:

Input: n = 10, pick = 6

Output: 6

Example 2:

Input: n = 1, pick = 1

Output: 1

Example 3:

Input: n = 2, pick = 1

Output: 1

Constraints:

1 <= n <= 231 - 1

1 <= pick <= n

**Solution :**

public class Solution extends GuessGame {

    public int guessNumber(int n) {

        int low=0;

        int high=n;

        while(low<=high)

        {

            int mid=low+(high-low)/2;

            if(guess(mid)==0)

            {

                return mid;

            }

            else if(guess(mid)==1)

            {

                low=mid+1;

            }

            else

            {

                high=mid-1;

            }

        }

        return 0;

    }

}

**7) Find Smallest Letter Greater Than Target (Leetcode – 744)**

You are given an array of characters letters that is sorted in non-decreasing order, and a character target. There are at least two different characters in letters.

Return the smallest character in letters that is lexicographically greater than target. If such a character does not exist, return the first character in letters.

Example 1:

Input: letters = ["c","f","j"], target = "a"

Output: "c"

Explanation: The smallest character that is lexicographically greater than 'a' in letters is 'c'.

Example 2:

Input: letters = ["c","f","j"], target = "c"

Output: "f"

Explanation: The smallest character that is lexicographically greater than 'c' in letters is 'f'.

Example 3:

Input: letters = ["x","x","y","y"], target = "z"

Output: "x"

Explanation: There are no characters in letters that is lexicographically greater than 'z' so we return letters[0].

Constraints:

2 <= letters.length <= 104

letters[i] is a lowercase English letter.

letters is sorted in non-decreasing order.

letters contains at least two different characters.

target is a lowercase English letter.

**Solution :**

class Solution {

    public char nextGreatestLetter(char[] a, char target) {

        int low=0;

       int high=a.length-1;

       char ans=a[0];

       while(low<=high)

       {

           int mid=(low+high)/2;

           if(a[mid]>target)

           {

               ans=a[mid];

               high=mid-1;

           }

           else

           {

               low=mid+1;

           }

       }

       return ans;

    }

}

**8) Search Insert Position (Leetcode - 35)**

Given a sorted array of distinct integers and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.

You must write an algorithm with O(log n) runtime complexity.

Example 1:

Input: nums = [1,3,5,6], target = 5

Output: 2

Example 2:

Input: nums = [1,3,5,6], target = 2

Output: 1

Example 3:

Input: nums = [1,3,5,6], target = 7

Output: 4

Constraints:

1 <= nums.length <= 104

-104 <= nums[i] <= 104

nums contains distinct values sorted in ascending order.

-104 <= target <= 104

**Solution :**

class Solution {

    public int searchInsert(int[] a, int target) {

       int low=0;

       int high=a.length-1;

       int ans=-1;

       while(low<=high)

       {

           int mid=(low+high)/2;

           if(a[mid]<=target)

           {

               ans=mid;

               low=mid+1;

           }

           else

           {

               high=mid-1;

           }

       }

       if(ans>=0 && a[ans]==target)

        return ans;

        else

        return ans+1;

    }

}

**9) First Bad Version (Leetcode - 278)**

You are a product manager and currently leading a team to develop a new product. Unfortunately, the latest version of your product fails the quality check. Since each version is developed based on the previous version, all the versions after a bad version are also bad.

Suppose you have n versions [1, 2, ..., n] and you want to find out the first bad one, which causes all the following ones to be bad.

You are given an API bool isBadVersion(version) which returns whether version is bad. Implement a function to find the first bad version. You should minimize the number of calls to the API.

Example 1:

Input: n = 5, bad = 4

Output: 4

Explanation:

call isBadVersion(3) -> false

call isBadVersion(5) -> true

call isBadVersion(4) -> true

Then 4 is the first bad version.

Example 2:

Input: n = 1, bad = 1

Output: 1

Constraints:

1 <= bad <= n <= 231 - 1

**Solution :**

public class Solution extends VersionControl {

    public int firstBadVersion(int n) {

        int start=1;

        int end=n;

        int ans=0;

        while(start<=end){

            int mid = start+(end-start)/2;

            boolean x = isBadVersion(mid);

            if(x==true){

                ans = mid;

                end = mid-1;

            }

            else{

                start = mid+1;

            }

        }

        return ans;

    }

}

**10) Sqrt(x) (Leetcode - 69)**

Given a non-negative integer x, return the square root of x rounded down to the nearest integer. The returned integer should be non-negative as well.

You must not use any built-in exponent function or operator.

For example, do not use pow(x, 0.5) in c++ or x \*\* 0.5 in python.

Example 1:

Input: x = 4

Output: 2

Explanation: The square root of 4 is 2, so we return 2.

Example 2:

Input: x = 8

Output: 2

Explanation: The square root of 8 is 2.82842..., and since we round it down to the nearest integer, 2 is returned.

Constraints:

0 <= x <= 231 – 1

**Solution :**

class Solution {

    public int mySqrt(int x) {

        long low=1;

        long high=x;

        long res=0;

        while(low<=high)

        {

            long mid=(low+high)/2;

            System.out.println(mid);

             if(mid\*mid<=x)

            {

               low=mid+1;

               res=mid;

            }

            else

            {

                 high=mid-1;

            }

        }

        return (int) res;

    }

}

**1) Capacity To Ship Packages Within D Days (Leetcode - 1011) (Medium)**

A conveyor belt has packages that must be shipped from one port to another within days days.

The ith package on the conveyor belt has a weight of weights[i]. Each day, we load the ship with packages on the conveyor belt (in the order given by weights). We may not load more weight than the maximum weight capacity of the ship.

Return the least weight capacity of the ship that will result in all the packages on the conveyor belt being shipped within days days.

Example 1:

Input: weights = [1,2,3,4,5,6,7,8,9,10], days = 5

Output: 15

Explanation: A ship capacity of 15 is the minimum to ship all the packages in 5 days like this:

1st day: 1, 2, 3, 4, 5

2nd day: 6, 7

3rd day: 8

4th day: 9

5th day: 10

Note that the cargo must be shipped in the order given, so using a ship of capacity 14 and splitting the packages into parts like (2, 3, 4, 5), (1, 6, 7), (8), (9), (10) is not allowed.

Example 2:

Input: weights = [3,2,2,4,1,4], days = 3

Output: 6

Explanation: A ship capacity of 6 is the minimum to ship all the packages in 3 days like this:

1st day: 3, 2

2nd day: 2, 4

3rd day: 1, 4

Example 3:

Input: weights = [1,2,3,1,1], days = 4

Output: 3

Explanation:

1st day: 1

2nd day: 2

3rd day: 3

4th day: 1, 1

Constraints:

1 <= days <= weights.length <= 5 \* 104

1 <= weights[i] <= 500

**Solution :**

class Solution {

    public int getDays(int[] weights,int k)

    {

        int days=1;

        int sum=0;

        for(int i:weights)

        {

            if(sum+i<=k)

            {

                sum+=i;

            }

            else

            {

                sum=i;

                days+=1;

            }

        }

        return days;

    }

    public int shipWithinDays(int[] weights, int days) {

        int start=0;

        int end=0;

        for(int i:weights)

        {

            if(i>start)

            {

                start=i;

            }

            end+=i;

        }

        int res=0;

        while(start<=end)

        {

            int mid=start+(end-start)/2;

            int d=getDays(weights,mid);

            if(d<=days)

            {

                end=mid-1;

                res=mid;

            }

            else

            {

                start=mid+1;

            }

        }

        return res;

    }

}

**2) Peak Index in a Mountain Array (Leetcode - 852)**

You are given an integer mountain array arr of length n where the values increase to a peak element and then decrease.

Return the index of the peak element.

Your task is to solve it in O(log(n)) time complexity.

Example 1:

Input: arr = [0,1,0]

Output: 1

Example 2:

Input: arr = [0,2,1,0]

Output: 1

Example 3:

Input: arr = [0,10,5,2]

Output: 1

Constraints:

3 <= arr.length <= 105

0 <= arr[i] <= 106

arr is guaranteed to be a mountain array.

**Solution :**

class Solution {

    public int peakIndexInMountainArray(int[] arr) {

        int start=0;

        int end=arr.length-1;

        int e=0;

        while (start < end) {

            int mid = (start + end) / 2;

            if (arr[mid] > arr[mid+1])

            {

                end=mid;

            } else{

                start=mid+1;

            }

        }

        return start;

    }

}

**3) Maximize the Confusion of an Exam (Leetcode - 2024)**

A teacher is writing a test with n true/false questions, with 'T' denoting true and 'F' denoting false. He wants to confuse the students by maximizing the number of consecutive questions with the same answer (multiple trues or multiple falses in a row).

You are given a string answerKey, where answerKey[i] is the original answer to the ith question. In addition, you are given an integer k, the maximum number of times you may perform the following operation:

Change the answer key for any question to 'T' or 'F' (i.e., set answerKey[i] to 'T' or 'F').

Return the maximum number of consecutive 'T's or 'F's in the answer key after performing the operation at most k times.

Example 1:

Input: answerKey = "TTFF", k = 2

Output: 4

Explanation: We can replace both the 'F's with 'T's to make answerKey = "TTTT".

There are four consecutive 'T's.

Example 2:

Input: answerKey = "TFFT", k = 1

Output: 3

Explanation: We can replace the first 'T' with an 'F' to make answerKey = "FFFT".

Alternatively, we can replace the second 'T' with an 'F' to make answerKey = "TFFF".

In both cases, there are three consecutive 'F's.

Example 3:

Input: answerKey = "TTFTTFTT", k = 1

Output: 5

Explanation: We can replace the first 'F' to make answerKey = "TTTTTFTT"

Alternatively, we can replace the second 'F' to make answerKey = "TTFTTTTT".

In both cases, there are five consecutive 'T's.

Constraints:

n == answerKey.length

1 <= n <= 5 \* 104

answerKey[i] is either 'T' or 'F'

1 <= k <= n

**Solution :**

class Solution {

    private int maxAnswer(String s,int k,char c)

    {

        int count=0,l=0,ans=0;

        for(int i=0;i<s.length();i++)

        {

            if(s.charAt(i)==c)

            {

                count++;

            }

            while(count>k)

            {

                if(s.charAt(l)==c) count--;

                l++;

            }

            System.out.println(l+" "+c);

            ans=Math.max(ans,i-l+1);

        }

        System.out.println(ans);

        return ans;

    }

    public int maxConsecutiveAnswers(String answerKey, int k) {

        return Math.max(maxAnswer(answerKey,k,'T'),maxAnswer(answerKey,k,'F'));

    }

}

**4) Two Sum II - Input Array Is Sorted (Leetcode - 167)**

Given a 1-indexed array of integers numbers that is already sorted in non-decreasing order, find two numbers such that they add up to a specific target number. Let these two numbers be numbers[index1] and numbers[index2] where 1 <= index1 < index2 <= numbers.length.

Return the indices of the two numbers, index1 and index2, added by one as an integer array [index1, index2] of length 2.

The tests are generated such that there is exactly one solution. You may not use the same element twice.

Your solution must use only constant extra space.

Example 1:

Input: numbers = [2,7,11,15], target = 9

Output: [1,2]

Explanation: The sum of 2 and 7 is 9. Therefore, index1 = 1, index2 = 2. We return [1, 2].

Example 2:

Input: numbers = [2,3,4], target = 6

Output: [1,3]

Explanation: The sum of 2 and 4 is 6. Therefore index1 = 1, index2 = 3. We return [1, 3].

Example 3:

Input: numbers = [-1,0], target = -1

Output: [1,2]

Explanation: The sum of -1 and 0 is -1. Therefore index1 = 1, index2 = 2. We return [1, 2].

Constraints:

2 <= numbers.length <= 3 \* 104

-1000 <= numbers[i] <= 1000

numbers is sorted in non-decreasing order.

-1000 <= target <= 1000

The tests are generated such that there is exactly one solution.

**Solution :**

class Solution {

    public int[] twoSum(int[] a, int target) {

        int low=0;

        int high=a.length-1;

        int [] res={-1,-1};

        while(low<high)

        {

            if(a[low]+a[high]==target)

            {

                res[0]=low+1;

                res[1]=high+1;

                return res;

            }

            else if(a[low]+a[high]>target)

            {

                high--;

            }

            else

            {

                low++;

            }

        }

        return res;

    }

}

**5) Find the Duplicate Number (Leetcode - 287)**

Given an array of integers nums containing n + 1 integers where each integer is in the range [1, n] inclusive.

There is only one repeated number in nums, return this repeated number.

You must solve the problem without modifying the array nums and uses only constant extra space.

Example 1:

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

Output: 2

Example 2:

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

Output: 3

Example 3:

Input: nums = [3,3,3,3,3]

Output: 3

Constraints:

1 <= n <= 105

nums.length == n + 1

1 <= nums[i] <= n

All the integers in nums appear only once except for precisely one integer which appears two or more times.

**Solution 1:(using BinarySearch )**

class Solution {

    public int findDuplicate(int[] nums) {

        int left = 1;

        int right = nums.length - 1;

        int ans=0;

        while (left <= right) {

            int mid = left + (right - left) / 2;

            int count = 0;

            // Count the numbers less than or equal to mid

            for (int num : nums) {

                if (num <= mid) {

                    count++;

                }

            }

            System.out.println(mid+" "+count);

            // If count is greater than mid, the duplicate lies in the left half

            if (count > mid) {

                ans=mid;

                right = mid-1;

            } else { // Otherwise, it lies in the right half

                left = mid + 1;

            }

        }

        return ans;

    }

}

**Solution 2:(Using cyclic sort)**

class Solution {

    public int findDuplicate(int[] a) {

        int i=0;

        while(i<a.length)

        {

                 int correct=a[i]-1;

                 if(a[i]!=a[correct])

                 {

                    int temp=a[i];

                    a[i]=a[correct];

                    a[correct]=temp;

                 }

                 else

                 {

                    i++;

                 }

        }

        for(i=0;i<a.length;i++)

        {

            if(i+1!=a[i])

            {

                return a[i];

            }

        }

        return -1;

    }

}

**6) Find the Smallest Divisor Given a Threshold (Leetcode - 1283)**

Given an array of integers nums and an integer threshold, we will choose a positive integer divisor, divide all the array by it, and sum the division's result. Find the smallest divisor such that the result mentioned above is less than or equal to threshold.

Each result of the division is rounded to the nearest integer greater than or equal to that element. (For example: 7/3 = 3 and 10/2 = 5).

The test cases are generated so that there will be an answer.

Example 1:

Input: nums = [1,2,5,9], threshold = 6

Output: 5

Explanation: We can get a sum to 17 (1+2+5+9) if the divisor is 1.

If the divisor is 4 we can get a sum of 7 (1+1+2+3) and if the divisor is 5 the sum will be 5 (1+1+1+2).

Example 2:

Input: nums = [44,22,33,11,1], threshold = 5

Output: 44

Constraints:

1 <= nums.length <= 5 \* 104

1 <= nums[i] <= 106

nums.length <= threshold <= 106

**Solution :**

class Solution {

    public int getThresh(int[] nums,int k)

    {

        int thresh=0;

        for(int i:nums)

        {

            thresh+=Math.ceil((double)i/k);

        }

        return thresh;

    }

    public int smallestDivisor(int[] nums, int threshold) {

        int start=1;

        int end=0;

        for(int i:nums)

        {

            if(i>end)

            {

                end=i;

            }

        }

        int res=0;

        while(start<=end)

        {

            int mid=start+(end-start)/2;

            int thresh=getThresh(nums,mid);

            if(thresh<=threshold)

            {

                res=mid;

                end=mid-1;

            }

            else

            {

                start=mid+1;

            }

        }

        return res;

    }

}

**7) Single Element in a Sorted Array (Leetcode - 540)**

You are given a sorted array consisting of only integers where every element appears exactly twice, except for one element which appears exactly once.

Return the single element that appears only once.

Your solution must run in O(log n) time and O(1) space.

Example 1:

Input: nums = [1,1,2,3,3,4,4,8,8]

Output: 2

Example 2:

Input: nums = [3,3,7,7,10,11,11]

Output: 10

Constraints:

1 <= nums.length <= 105

0 <= nums[i] <= 105

**Solution 1:**

**Solution 2:**

//2nd Approach using XOR, T=O(n)

class Solution {

    public int singleNonDuplicate(int[] nums) {

        int ans = 0;

        for (int num : nums) {

            ans ^= num;

        }

        return ans;

    }

}

**8) Minimum Number of Days to Make m Bouquets (Leetcode - 1482)**

You are given an integer array bloomDay, an integer m and an integer k.

You want to make m bouquets. To make a bouquet, you need to use k adjacent flowers from the garden.

The garden consists of n flowers, the ith flower will bloom in the bloomDay[i] and then can be used in exactly one bouquet.

Return the minimum number of days you need to wait to be able to make m bouquets from the garden. If it is impossible to make m bouquets return -1.

Example 1:

Input: bloomDay = [1,10,3,10,2], m = 3, k = 1

Output: 3

Explanation: Let us see what happened in the first three days. x means flower bloomed and \_ means flower did not bloom in the garden.

We need 3 bouquets each should contain 1 flower.

After day 1: [x, \_, \_, \_, \_] // we can only make one bouquet.

After day 2: [x, \_, \_, \_, x] // we can only make two bouquets.

After day 3: [x, \_, x, \_, x] // we can make 3 bouquets. The answer is 3.

Example 2:

Input: bloomDay = [1,10,3,10,2], m = 3, k = 2

Output: -1

Explanation: We need 3 bouquets each has 2 flowers, that means we need 6 flowers. We only have 5 flowers so it is impossible to get the needed bouquets and we return -1.

Example 3:

Input: bloomDay = [7,7,7,7,12,7,7], m = 2, k = 3

Output: 12

Explanation: We need 2 bouquets each should have 3 flowers.

Here is the garden after the 7 and 12 days:

After day 7: [x, x, x, x, \_, x, x]

We can make one bouquet of the first three flowers that bloomed. We cannot make another bouquet from the last three flowers that bloomed because they are not adjacent.

After day 12: [x, x, x, x, x, x, x]

It is obvious that we can make two bouquets in different ways.

Constraints:

bloomDay.length == n

1 <= n <= 105

1 <= bloomDay[i] <= 109

1 <= m <= 106

1 <= k <= n

**Solution :**

class Solution {

    public int findBou(int[] arr,int days,int k)

    {

        int count=0;

        int bou=0;

        for(int i=0;i<arr.length;i++)

        {

            if(arr[i]<=days)

            {

                count+=1;

            }

            else

            {

                count=0;

            }

            if(count==k)

            {

                bou+=1;

                count=0;

            }

        }

        return bou;

    }

    public int minDays(int[] bloomDay, int m, int k) {

        if ((long) m \* k > bloomDay.length) {

            return -1;

        }

        int start=Integer.MAX\_VALUE;

        int end=0;

        for(int i:bloomDay)

        {

            if(i<start)

            {

                start=i;

            }

            if(i>end)

            {

                end=i;

            }

        }

        int res=0;

        while(start<=end)

        {

            int mid=start+(end-start)/2;

            int bo = findBou(bloomDay,mid,k);

            if(bo>=m)

            {

               res=mid;

               end=mid-1;

            }

            else

            {

                start=mid+1;

            }

        }

        return res;

    }

}

**9) Search a 2D Matrix II (Leetcode - 240)**

Write an efficient algorithm that searches for a value target in an m x n integer matrix matrix. This matrix has the following properties:

Integers in each row are sorted in ascending from left to right.

Integers in each column are sorted in ascending from top to bottom.

Example 1:



Input: matrix = [[1,4,7,11,15],[2,5,8,12,19],[3,6,9,16,22],[10,13,14,17,24],[18,21,23,26,30]], target = 5

Output: true

Example 2:



Input: matrix = [[1,4,7,11,15],[2,5,8,12,19],[3,6,9,16,22],[10,13,14,17,24],[18,21,23,26,30]], target = 20

Output: false

Constraints:

m == matrix.length

n == matrix[i].length

1 <= n, m <= 300

-109 <= matrix[i][j] <= 109

All the integers in each row are sorted in ascending order.

All the integers in each column are sorted in ascending order.

-109 <= target <= 109

**Solution :**

class Solution {

    public boolean searchMatrix(int[][] matrix, int target) {

        int n = matrix.length;

        int m = matrix[0].length;

        int row = 0,col=m-1;

        while(row<n && col>=0)

        {

            if(matrix[row][col]>target)

            {

                col-=1;

            }

            else if(matrix[row][col]<target)

            {

                row+=1;

            }

            else

            {

                return true;

            }

        }

        return false;

    }

}

**10) Find a Peak Element II (Leetcode - 1901)**

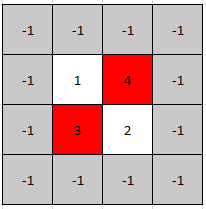
A peak element in a 2D grid is an element that is strictly greater than all of its adjacent neighbors to the left, right, top, and bottom.

Given a 0-indexed m x n matrix mat where no two adjacent cells are equal, find any peak element mat[i][j] and return the length 2 array [i,j].

You may assume that the entire matrix is surrounded by an outer perimeter with the value -1 in each cell.

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

Example 1:

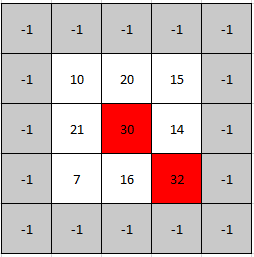


Input: mat = [[1,4],[3,2]]

Output: [0,1]

Explanation: Both 3 and 4 are peak elements so [1,0] and [0,1] are both acceptable answers.

Example 2:



Input: mat = [[10,20,15],[21,30,14],[7,16,32]]

Output: [1,1]

Explanation: Both 30 and 32 are peak elements so [1,1] and [2,2] are both acceptable answers.

Constraints:

m == mat.length

n == mat[i].length

1 <= m, n <= 500

1 <= mat[i][j] <= 105

No two adjacent cells are equal.

**Solution :**

//take u forward

class Solution {

    public int[] findPeakGrid(int[][] mat) {

        int n = mat.length;

        int m = mat[0].length;

        int low = 0;

        int high = m-1;

        while (low<=high) {

            int mid = (low+high)/2;

            int row = find\_max\_in\_col(mat,n,mid);

            int left = (mid-1 >= 0)?mat[row][mid-1]:-1;

            int right = (mid+1 < m)?mat[row][mid+1]:-1;

            if (left<mat[row][mid] && mat[row][mid]>right) {

                return new int[] {row,mid};

            }

            else if(left>mat[row][mid]) {

                high = mid-1;

            }

            else {

                low = mid+1;

            }

        }

        return new int[] {-1,-1};

    }

    public int find\_max\_in\_col(int mat[][], int n, int col) {

        int idx = 0;

        int value = 0;

        for (int i=0; i<n; i++) {

            if (mat[i][col]>value) {

                value = mat[i][col];

                idx = i;

            }

        }

        return idx;

    }

}

**11) Find Minimum in Rotated Sorted Array (Leetcode - 153)**

Suppose an array of length n sorted in ascending order is rotated between 1 and n times.

For example, the array nums = [0,1,2,4,5,6,7] might become:

[4,5,6,7,0,1,2] if it was rotated 4 times.

[0,1,2,4,5,6,7] if it was rotated 7 times.

Notice that rotating an array [a[0], a[1], a[2], ..., a[n-1]] 1 time results in the array [a[n-1], a[0], a[1], a[2], ..., a[n-2]].

Given the sorted rotated array nums of unique elements, return the minimum element of this array.

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

Example 1:

Input: nums = [3,4,5,1,2]

Output: 1

Explanation: The original array was [1,2,3,4,5] rotated 3 times.

Example 2:

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

Output: 0

Explanation: The original array was [0,1,2,4,5,6,7] and it was rotated 4 times.

Example 3:

Input: nums = [11,13,15,17]

Output: 11

Explanation: The original array was [11,13,15,17] and it was rotated 4 times.

Constraints:

n == nums.length

1 <= n <= 5000

-5000 <= nums[i] <= 5000

All the integers of nums are unique.

nums is sorted and rotated between 1 and n times.

**Solution :**

class Solution {

    public static int pivot(int[] a)

    {

        int start=0;

        int end=a.length-1;

        while(start<end)

        {

            int mid=start+(end-start)/2;

            if(mid<end && a[mid]>a[mid+1])

            {

                return mid;

            }

            if(mid>start && a[mid-1]>a[mid])

            {

                return mid-1;

            }

            if(a[start]>a[mid])

            {

                end=mid-1;

            }

            else

            {

                start=mid+1;

            }

        }

        return -1;

    }

    public int findMin(int[] nums) {

        return nums[pivot(nums)+1];

    }

}

**12) Search a 2D Matrix (Leetcode - 74)**

You are given an m x n integer matrix matrix with the following two properties:

Each row is sorted in non-decreasing order.

The first integer of each row is greater than the last integer of the previous row.

Given an integer target, return true if target is in matrix or false otherwise.

You must write a solution in O(log(m \* n)) time complexity.

Example 1:



Input: matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 3

Output: true

Example 2:



Input: matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 13

Output: false

Constraints:

m == matrix.length

n == matrix[i].length

1 <= m, n <= 100

-104 <= matrix[i][j], target <= 104

**Solution :**

class Solution {

    public boolean searchMatrix(int[][] matrix, int target) {

        int n = matrix.length;

        int m = matrix[0].length;

        int low = 0,high=m\*n-1;

        while(low<=high)

        {

            int mid = (low+high)/2;

            if(matrix[mid/m][mid%m]==target) return true;

            else if(matrix[mid/m][mid%m]<target) low=mid+1;

            else high=mid-1;

        }

        return false;

    }

}

**13) Koko Eating Bananas (Leetcode - 875)**

Koko loves to eat bananas. There are n piles of bananas, the ith pile has piles[i] bananas. The guards have gone and will come back in h hours.

Koko can decide her bananas-per-hour eating speed of k. Each hour, she chooses some pile of bananas and eats k bananas from that pile. If the pile has less than k bananas, she eats all of them instead and will not eat any more bananas during this hour.

Koko likes to eat slowly but still wants to finish eating all the bananas before the guards return.

Return the minimum integer k such that she can eat all the bananas within h hours.

Example 1:

Input: piles = [3,6,7,11], h = 8

Output: 4

Example 2:

Input: piles = [30,11,23,4,20], h = 5

Output: 30

Example 3:

Input: piles = [30,11,23,4,20], h = 6

Output: 23

Constraints:

1 <= piles.length <= 104

piles.length <= h <= 109

1 <= piles[i] <= 109

**Solution :**

class Solution {

    public long findHours(int[] piles,long k)

    {

        long hours=0;

        for(int i:piles)

        {

            hours+=(int) Math.ceil((double)i/k);

        }

        return hours;

    }

    public int minEatingSpeed(int[] piles, int h) {

        long start=1;

        long end=0;

        for(int i:piles)

        {

            if(i>end)

            {

                end=i;

            }

        }

         long res=0;

        while(start<=end)

        {

            long k=start+(end-start)/2;

            long hours = findHours(piles,k);

            System.out.println(k+" "+hours);

            if(hours<=h)

            {

                end=k-1;

                res=k;

            }

            else

            {

                start=k+1;

            }

        }

    return (int)res;

    }

}

**14) Find Peak Element (Leetcode - 162)**

A peak element is an element that is strictly greater than its neighbors.

Given a 0-indexed integer array nums, find a peak element, and return its index. If the array contains multiple peaks, return the index to any of the peaks.

You may imagine that nums[-1] = nums[n] = -∞. In other words, an element is always considered to be strictly greater than a neighbor that is outside the array.

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

Example 1:

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

Output: 2

Explanation: 3 is a peak element and your function should return the index number 2.

Example 2:

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

Output: 5

Explanation: Your function can return either index number 1 where the peak element is 2, or index number 5 where the peak element is 6.

Constraints:

1 <= nums.length <= 1000

-231 <= nums[i] <= 231 - 1

nums[i] != nums[i + 1] for all valid i.

**Solution :**

**//[1,2]**

class Solution {

    public int findPeakElement(int[] arr) {

        int start=0;

        int end=arr.length-1;

        while (start < end) {

            int mid = (start + end) / 2;

            System.out.println(mid);

            if (arr[mid] > arr[mid+1])

            {

                end=mid;

            } else{

                start=mid+1;

            }

        }

        return start;

    }

  }

**15) Find First and Last Position of Element in Sorted Array (Leetcode - 34)**

Given an array of integers nums sorted in non-decreasing order, find the starting and ending position of a given target value.

If target is not found in the array, return [-1, -1].

You must write an algorithm with O(log n) runtime complexity.

Example 1:

Input: nums = [5,7,7,8,8,10], target = 8

Output: [3,4]

Example 2:

Input: nums = [5,7,7,8,8,10], target = 6

Output: [-1,-1]

Example 3:

Input: nums = [], target = 0

Output: [-1,-1]

Constraints:

0 <= nums.length <= 105

-109 <= nums[i] <= 109

nums is a non-decreasing array.

-109 <= target <= 109

**Solution :**

class Solution {

    public int search(int[] a,int target,boolean start)

    {

       int low=0;

       int high=a.length-1;

       int ans=-1;

       while(low<=high)

       {

           int mid=(low+high)/2;

           if(a[mid]==target)

           {

               ans=mid;

               if(start)

                high=mid-1;

               else

                low=mid+1;

           }

           else if(a[mid]<target)

           {

               low=mid+1;

           }

           else

           {

               high=mid-1;

           }

       }

        return ans;

    }

    public int[] searchRange(int[] nums, int target) {

        int[] a = {-1,-1};

        a[0]=search(nums,target,true);

        a[1]=search(nums,target,false);

        return a;

    }

}

**16) Search in Rotated Sorted Array (Leetcode -33)**

There is an integer array nums sorted in ascending order (with distinct values).

Prior to being passed to your function, nums is possibly rotated at an unknown pivot index k (1 <= k < nums.length) such that the resulting array is [nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]] (0-indexed). For example, [0,1,2,4,5,6,7] might be rotated at pivot index 3 and become [4,5,6,7,0,1,2].

Given the array nums after the possible rotation and an integer target, return the index of target if it is in nums, or -1 if it is not in nums.

You must write an algorithm with O(log n) runtime complexity.

Example 1:

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

Output: 4

Example 2:

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

Output: -1

Example 3:

Input: nums = [1], target = 0

Output: -1

Constraints:

1 <= nums.length <= 5000

-104 <= nums[i] <= 104

All values of nums are unique.

nums is an ascending array that is possibly rotated.

-104 <= target <= 104

**Solution :**

class Solution {

    public int findPivotElement(int[] arr) {

        int start=0;

        int end=arr.length-1;

        while (start <= end) {

            int mid = (start + end) / 2;

            if (mid+1<=end && arr[mid] > arr[mid+1])

            {

                return mid;

            }

            else if(mid-1>=0 && arr[mid]<arr[mid-1])

            {

                return mid-1;

            }

            else if(arr[start]>=arr[mid])

            {

                end=mid-1;

            }

            else{

                start=mid+1;

            }

        }

        return -1;

    }

    int binarySearch(int[] arr, int l, int r, int x)

    {

        while (l <= r) {

            int mid = (l + r) / 2;

            if (arr[mid] == x) {

                return mid;

            } else if (arr[mid] > x) {

                r = mid - 1;

            } else {

              l = mid + 1;

            }

        }

        return -1;

    }

    public int search(int[] nums, int target) {

        int pivot=findPivotElement(nums);

        int f=binarySearch(nums,0,pivot,target);

        if(f==-1)

        {

            return binarySearch(nums,pivot+1,nums.length-1,target);

        }

        return f;

    }

}

**17) Aggressive Cows (GFG)**

You are given an array consisting of n integers which denote the position of a stall. You are also given an integer k which denotes the number of aggressive cows. You are given the task of assigning stalls to k cows such that the minimum distance between any two of them is the maximum possible.

The first line of input contains two space-separated integers n and k.

The second line contains n space-separated integers denoting the position of the stalls.

Example 1:

Input:

n=5

k=3

stalls = [1 2 4 8 9]

Output:

3

Explanation:

The first cow can be placed at stalls[0],

the second cow can be placed at stalls[2] and

the third cow can be placed at stalls[3].

The minimum distance between cows, in this case, is 3,

which also is the largest among all possible ways.

Example 2:

Input:

n=5

k=3

stalls = [10 1 2 7 5]

Output:

4

Explanation:

The first cow can be placed at stalls[0],

the second cow can be placed at stalls[1] and

the third cow can be placed at stalls[4].

The minimum distance between cows, in this case, is 4,

which also is the largest among all possible ways.

Your Task:

Complete the function int solve(), which takes integer n, k, and a vector stalls with n integers as input and returns the largest possible minimum distance between cows.

Expected Time Complexity: O(n\*log(10^9)).

Expected Auxiliary Space: O(1).

Constraints:

2 <= n <= 10^5

2 <= k <= n

0 <= stalls[i] <= 10^9

**Solution :**

class Solution {

public static int getCows(int[] stalls,int k)

{

int c=1;

int last=stalls[0];

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

{

if(stalls[i]-last>=k)

{

c+=1;

last=stalls[i];

}

}

return c;

}

public static int solve(int n, int k, int[] stalls) {

Arrays.sort(stalls);

int start=1;

int max=Integer.MIN\_VALUE;

int min=Integer.MAX\_VALUE;

for(int i:stalls)

{

min=Math.min(min,i);

max=Math.max(max,i);

}

int end=max-min;

while(start<=end)

{

int mid = start+(end-start)/2;

int c=getCows(stalls,mid);

if(c>=k)

{

start=mid+1;

}

else

{

end=mid-1;

}

}

return end;

}

}

**18) Allocate Books (coding ninjas)**

Given an array ‘arr’ of integer numbers, ‘arr[i]’ represents the number of pages in the ‘i-th’ book.

There are ‘m’ number of students, and the task is to allocate all the books to the students.

Allocate books in such a way that:

1. Each student gets at least one book.

2. Each book should be allocated to only one student.

3. Book allocation should be in a contiguous manner.

You have to allocate the book to ‘m’ students such that the maximum number of pages assigned to a student is minimum.

If the allocation of books is not possible, return -1.

Example:

Input: ‘n’ = 4 ‘m’ = 2

‘arr’ = [12, 34, 67, 90]

Output: 113

Explanation: All possible ways to allocate the ‘4’ books to '2' students are:

12 | 34, 67, 90 - the sum of all the pages of books allocated to student 1 is ‘12’, and student two is ‘34+ 67+ 90 = 191’, so the maximum is ‘max(12, 191)= 191’.

12, 34 | 67, 90 - the sum of all the pages of books allocated to student 1 is ‘12+ 34 = 46’, and student two is ‘67+ 90 = 157’, so the maximum is ‘max(46, 157)= 157’.

12, 34, 67 | 90 - the sum of all the pages of books allocated to student 1 is ‘12+ 34 +67 = 113’, and student two is ‘90’, so the maximum is ‘max(113, 90)= 113’.

We are getting the minimum in the last case.

Hence answer is ‘113’.

Detailed explanation ( Input/output format, Notes, Images )

Sample Input 1:

4 2

12 34 67 90

Sample Output 1:

113

Explanation of sample input 1:

All possible ways to allocate the ‘4’ books to '2' students are:

12 | 34, 67, 90 - the sum of all the pages of books allocated to student 1 is ‘12’, and student two is ‘34+ 67+ 90 = 191’, so the maximum is ‘max(12, 191)= 191’.

12, 34 | 67, 90 - the sum of all the pages of books allocated to student 1 is ‘12+ 34 = 46’, and student two is ‘67+ 90 = 157’, so the maximum is ‘max(46, 157)= 157’.

12, 34, 67 | 90 - the sum of all the pages of books allocated to student 1 is ‘12+ 34 +67 = 113’, and student two is ‘90’, so the maximum is ‘max(113, 90)= 113’.

We are getting the minimum in the last case.

Hence answer is ‘113’.

Sample Input 2:

5 4

25 46 28 49 24

Sample Output 2:

71

Explanation of sample input 2:

All possible ways to allocate the ‘5’ books to '4' students are:

25 | 46 | 28 | 49 24 - the sum of all the pages of books allocated to students 1, 2, 3, and 4 are '25', '46', '28', and '73'. So the maximum is '73'.

25 | 46 | 28 49 | 24 - the sum of all the pages of books allocated to students 1, 2, 3, and 4 are '25', '46', '77', and '24'. So the maximum is '77'.

25 | 46 28 | 49 | 24 - the sum of all the pages of books allocated to students 1, 2, 3, and 4 are '25', '74', '49', and '24'. So the maximum is '74'.

25 46 | 28 | 49 | 24 - the sum of all the pages of books allocated to students 1, 2, 3, and 4 are '71', '28', '49', and '24'. So the maximum is '71'.

We are getting the minimum in the last case.

Hence answer is ‘71’.

Expected time complexity:

The expected time complexity is O(n \* log(s)), where ‘n’ is the number of integers in the array ‘arr’ and ‘s’ is the sum of all the elements of ‘arr’.

Constraints:

2 <= 'n' <= 10 ^ 3

1 <= 'm' <= 10 ^ 3

1 <= 'arr[i]' <= 10 ^ 9

The sum of all arr[i] does not exceed 10 ^ 9.

Where ‘n’ denotes the number of books and ‘m’ denotes the number of students. ‘arr[i]’ denotes an element at position ‘i’ in the sequence.

Time limit: 1 second

**Solution :**

import java.util.ArrayList;

public class Solution {

public static int findStudent(ArrayList<Integer> arr,int k)

{

int sum=0;

int students=1;

for(int i:arr)

{

if(sum+i<=k)

{

sum+=i;

}

else{

sum=i;

students+=1;

}

}

return students;

}

public static int findPages(ArrayList<Integer> arr, int n, int m) {

if(m>n) return -1;

int start = Integer.MIN\_VALUE;

int end=0;

for(int i:arr)

{

start=Math.max(start,i);

end+=i;

}

int res=0;

while(start<=end)

{

int mid=start+(end-start)/2;

int s = findStudent(arr,mid);

if(s<=m)

{

end=mid-1;

}

else

{

start=mid+1;

}

}

return start;

}

}

**19) Median of row wise sorted matrix**

Example 1:

Input Format:M = 3, N = 3, matrix[][] =

1 4 9

2 5 6

3 8 7

Result: 5

Explanation: If we find the linear sorted array, the array becomes 1 2 3 4 5 6 7 8 9. So, median = 5

Example 2:

Input Format:M = 3, N = 3, matrix[][] =

1 3 8

2 3 4

1 2 5

Result: 3

Explanation: If we find the linear sorted array, the array becomes 1 1 2 2 3 3 4 5 7 8. So, median = 3

**Solution :**

import java.util.\*;

class tUf {

static int upperBound(int[] arr, int x, int n) {

int low = 0, high = n - 1;

int ans = n;

while (low <= high) {

int mid = (low + high) / 2;

// maybe an answer

if (arr[mid] > x) {

ans = mid;

// look for a smaller index on the left

high = mid - 1;

} else {

low = mid + 1; // look on the right

}

}

return ans;

}

static int countSmallEqual(int[][] matrix, int m, int n, int x) {

int cnt = 0;

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

cnt += upperBound(matrix[i], x, n);

}

return cnt;

}

static int median(int[][] matrix, int m, int n) {

int low = Integer.MAX\_VALUE, high = Integer.MIN\_VALUE;

// point low and high to right elements

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

low = Math.min(low, matrix[i][0]);

high = Math.max(high, matrix[i][n - 1]);

}

int req = (n \* m) / 2;

while (low <= high) {

int mid = (low + high) / 2;

int smallEqual = countSmallEqual(matrix, m, n, mid);

if (smallEqual <= req) low = mid + 1;

else high = mid - 1;

}

return low;

}

public static void main(String[] args) {

int[][] matrix = {

{1, 2, 3, 4, 5},

{8, 9, 11, 12, 13},

{21, 23, 25, 27, 29}

};

int m = matrix.length;

int n = matrix[0].length;

int ans = median(matrix, m, n);

System.out.println("The median element is: " + ans);

}

}

**1) Split Array Largest Sum (Leetcode - 410) (Hard)**

Given an integer array nums and an integer k, split nums into k non-empty subarrays such that the largest sum of any subarray is minimized.

Return the minimized largest sum of the split.

A subarray is a contiguous part of the array.

Example 1:

Input: nums = [7,2,5,10,8], k = 2

Output: 18

Explanation: There are four ways to split nums into two subarrays.

The best way is to split it into [7,2,5] and [10,8], where the largest sum among the two subarrays is only 18.

Example 2:

Input: nums = [1,2,3,4,5], k = 2

Output: 9

Explanation: There are four ways to split nums into two subarrays.

The best way is to split it into [1,2,3] and [4,5], where the largest sum among the two subarrays is only 9.

Constraints:

1 <= nums.length <= 1000

0 <= nums[i] <= 106

1 <= k <= min(50, nums.length)

**Solution :**

class Solution {

    public int splitArray(int[] nums, int k) {

        int start=0;

        int end=0;

        for(int i:nums)

        {

            start=Math.max(start,i);

            end+=i;

        }

        while(start<end)

        {

            int mid=(start+end)/2;

            int sum=0;

            int pieces=1;

            for(int i:nums)

            {

                if(sum+i>mid)

                {

                    sum=i;

                    pieces++;

                }

                else

                {

                    sum+=i;

                }

            }

            if(pieces>k)

            {

                start=mid+1;

            }

            else

            {

                end=mid;

            }

        }

        return end;

    }

}

**2) Median of Two Sorted Arrays (Leetcode - 4)**

Given two sorted arrays nums1 and nums2 of size m and n respectively, return the median of the two sorted arrays.

The overall run time complexity should be O(log (m+n)).

Example 1:

Input: nums1 = [1,3], nums2 = [2]

Output: 2.00000

Explanation: merged array = [1,2,3] and median is 2.

Example 2:

Input: nums1 = [1,2], nums2 = [3,4]

Output: 2.50000

Explanation: merged array = [1,2,3,4] and median is (2 + 3) / 2 = 2.5.

Constraints:

nums1.length == m

nums2.length == n

0 <= m <= 1000

0 <= n <= 1000

1 <= m + n <= 2000

-106 <= nums1[i], nums2[i] <= 106

**Solution :**

class Solution {

    public double findMedianSortedArrays(int[] nums1, int[] nums2) {

        int n1 = nums1.length;

        int n2 = nums2.length;

        if(n1>n2) return findMedianSortedArrays(nums2,nums1);

        int low = 0,high = n1;

        int left = (n1+n2+1)/2;

        int n=n1+n2;

        while(low<=high)

        {

            int mid1 = (low+high)/2;

            int mid2 = left-mid1;

            int l1 = Integer.MIN\_VALUE,l2=Integer.MIN\_VALUE;

            int r1 = Integer.MAX\_VALUE,r2=Integer.MAX\_VALUE;

            System.out.println(mid1+" "+mid2);

            if(mid1<n1) r1 = nums1[mid1];

            if(mid2<n2) r2 = nums2[mid2];

            if(mid1-1 >= 0) l1=nums1[mid1-1];

            if(mid2-1 >= 0) l2 = nums2[mid2-1];

            if(l1<=r2 && l2 <= r1)

            {

                if(n%2==1) return Math.max(l1,l2);

                return (double) (Math.max(l1,l2)+Math.min(r1,r2))/2.0;

            }

            else if(l1>r2) high=mid1-1;

            else low=mid1+1;

        }

        return 0;

    }

}

**3) Kth element of two sorted arrays**

Given two sorted arrays of size m and n respectively, you are tasked with finding the element that would be at the kth position of the final sorted array.

**Solution :**

public static int kthElement(ArrayList<Integer> a, ArrayList<Integer> b, int m, int n, int k) {

if (m > n) return kthElement(b, a, n, m, k);

int left = k; // length of left half

// apply binary search:

int low = Math.max(0, k - n), high = Math.min(k, m);

while (low <= high) {

int mid1 = (low + high) >> 1;

int mid2 = left - mid1;

// calculate l1, l2, r1, and r2

int l1 = Integer.MIN\_VALUE, l2 = Integer.MIN\_VALUE;

int r1 = Integer.MAX\_VALUE, r2 = Integer.MAX\_VALUE;

if (mid1 < m) r1 = a.get(mid1);

if (mid2 < n) r2 = b.get(mid2);

if (mid1 - 1 >= 0) l1 = a.get(mid1 - 1);

if (mid2 - 1 >= 0) l2 = b.get(mid2 - 1);

if (l1 <= r2 && l2 <= r1) {

return Math.max(l1, l2);

}

// eliminate the halves:

else if (l1 > r2) high = mid1 - 1;

else low = mid1 + 1;

}

return 0; // dummy statement

}

**4) Find in Mountain Array (Leetcode - 1095)**

(This problem is an interactive problem.)

You may recall that an array arr is a mountain array if and only if:

arr.length >= 3

There exists some i with 0 < i < arr.length - 1 such that:

arr[0] < arr[1] < ... < arr[i - 1] < arr[i]

arr[i] > arr[i + 1] > ... > arr[arr.length - 1]

Given a mountain array mountainArr, return the minimum index such that mountainArr.get(index) == target. If such an index does not exist, return -1.

You cannot access the mountain array directly. You may only access the array using a MountainArray interface:

MountainArray.get(k) returns the element of the array at index k (0-indexed).

MountainArray.length() returns the length of the array.

Submissions making more than 100 calls to MountainArray.get will be judged Wrong Answer. Also, any solutions that attempt to circumvent the judge will result in disqualification.

Example 1:

Input: array = [1,2,3,4,5,3,1], target = 3

Output: 2

Explanation: 3 exists in the array, at index=2 and index=5. Return the minimum index, which is 2.

Example 2:

Input: array = [0,1,2,4,2,1], target = 3

Output: -1

Explanation: 3 does not exist in the array, so we return -1.

Constraints:

3 <= mountain\_arr.length() <= 104

0 <= target <= 109

0 <= mountain\_arr.get(index) <= 109

**Solution :**

class Solution {

    public int findPeakElement(MountainArray arr) {

        int start=0;

        int end=arr.length()-1;

        int e=0;

        while (start < end) {

            int mid = (start + end) / 2;

            if (arr.get(mid) > arr.get(mid+1))

            {

                end=mid;

            } else{

                start=mid+1;

            }

        }

        return end;

    }

    int binarySearch(MountainArray arr, int l, int r, int x)

    {

        while (l <= r) {

            int mid = (l + r) / 2;

            if (arr.get(mid) == x) {

                return mid;

            } else if (arr.get(mid) > x) {

                r = mid - 1;

            } else {

              l = mid + 1;

            }

        }

        return -1;

    }

    static int binarySearchD(MountainArray arr, int l, int r, int x)

    {

        while (l <= r) {

            int mid = (l + r) / 2;

            if (arr.get(mid) == x) {

                return mid;

            } else if (arr.get(mid) > x) {

                l = mid + 1;

            } else {

              r = mid - 1;

            }

        }

        return -1;

    }

    public int findInMountainArray(int target, MountainArray mountainArr) {

        int peak=findPeakElement(mountainArr);

        int f=binarySearch(mountainArr,0,peak,target);

        if(f==-1)

        {

            return binarySearchD(mountainArr,peak+1,mountainArr.length()-1,target);

        }

        return f;

    }

}

**Back Tracking**

**1) Permutations (Leetcode - 46) (Medium)**

Given an array nums of distinct integers, return all the possible permutations. You can return the answer in any order.

Example 1:

Input: nums = [1,2,3]

Output: [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]

Example 2:

Input: nums = [0,1]

Output: [[0,1],[1,0]]

Example 3:

Input: nums = [1]

Output: [[1]]

Constraints:

1 <= nums.length <= 6

-10 <= nums[i] <= 10

All the integers of nums are unique.

**Solution 1:(using visited array more efficient (space – o(n))**

class Solution {

    public void permutations(int[] nums,boolean[] v,List<Integer> l,List<List<Integer>> r)

    {

        if(l.size()==nums.length)

        {

            r.add(new ArrayList<>(l));

            return;

        }

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

        {

            if(!v[i])

            {

                v[i]=true;

                l.add(nums[i]);

                permutations(nums,v,l,r);

                l.remove(l.size()-1);

                v[i]=false;

            }

        }

    }

    public List<List<Integer>> permute(int[] nums) {

        List<List<Integer>> r = new ArrayList<>();

        List<Integer> l = new ArrayList<>();

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

        permutations(nums,v,l,r);

        return r;

    }

}

**Solution 2:(using the same list search takes o(n))**

class Solution {

    public void permutations(int[] nums,List<Integer> l,List<List<Integer>> r)

    {

        if(l.size()==nums.length)

        {

            r.add(new ArrayList<>(l));

            return;

        }

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

        {

            if(!l.contains(nums[i]))

            {

                l.add(nums[i]);

                permutations(nums,l,r);

                l.remove(l.size()-1);

            }

        }

    }

    public List<List<Integer>> permute(int[] nums) {

        List<List<Integer>> r = new ArrayList<>();

        List<Integer> l = new ArrayList<>();

        permutations(nums,l,r);

        return r;

    }

}

**2) Permutations II (Leetcode - 47)**

Given a collection of numbers, nums, that might contain duplicates, return all possible unique permutations in any order.

Example 1:

Input: nums = [1,1,2]

Output:

[[1,1,2],

[1,2,1],

[2,1,1]]

Example 2:

Input: nums = [1,2,3]

Output: [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]

Constraints:

1 <= nums.length <= 8

-10 <= nums[i] <= 10

**Solution :**

class Solution {

   public void permutations(int[] nums,boolean[] v,ArrayList<Integer> l,List<List<Integer>> r)

    {

        if(l.size()==nums.length && !r.contains(l))

        {

            r.add(new ArrayList<>(l));

            return;

        }

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

        {

            if(!v[i])

            {

                v[i]=true;

                l.add(nums[i]);

                permutations(nums,v,l,r);

                l.remove(l.size()-1);

                v[i]=false;

            }

        }

    }

    public List<List<Integer>> permuteUnique(int[] nums) {

        List<List<Integer>> r = new ArrayList<>();

        ArrayList<Integer> l = new ArrayList<>();

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

        permutations(nums,v,l,r);

        return r;

    }

}

**3) Subsets (Leetcode - 78)**

Given an integer array nums of unique elements, return all possible

subsets

(the power set).

The solution set must not contain duplicate subsets. Return the solution in any order.

Example 1:

Input: nums = [1,2,3]

Output: [[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]

Example 2:

Input: nums = [0]

Output: [[],[0]]

Constraints:

1 <= nums.length <= 10

-10 <= nums[i] <= 10

All the numbers of nums are unique.

**Solution :**

class Solution {

    List<List<Integer>> res = new ArrayList<>();

    public void combinations(int[] nums,int idx,ArrayList<Integer> l)

    {

        if(idx==nums.length)

        {

            res.add(new ArrayList<>(l));

            return ;

        }

        l.add(nums[idx]);

        combinations(nums,idx+1,l);

        l.remove(l.size()-1);

        combinations(nums,idx+1,l);

    }

    public List<List<Integer>> subsets(int[] nums) {

        combinations(nums,0,new ArrayList<>());

        return res;

    }

}

**4) Combination Sum (Leetcode - 39)**

Given an array of distinct integers candidates and a target integer target, return a list of all unique combinations of candidates where the chosen numbers sum to target. You may return the combinations in any order.

The same number may be chosen from candidates an unlimited number of times. Two combinations are unique if the frequency of at least one of the chosen numbers is different.

The test cases are generated such that the number of unique combinations that sum up to target is less than 150 combinations for the given input.

Example 1:

Input: candidates = [2,3,6,7], target = 7

Output: [[2,2,3],[7]]

Explanation:

2 and 3 are candidates, and 2 + 2 + 3 = 7. Note that 2 can be used multiple times.

7 is a candidate, and 7 = 7.

These are the only two combinations.

Example 2:

Input: candidates = [2,3,5], target = 8

Output: [[2,2,2,2],[2,3,3],[3,5]]

Example 3:

Input: candidates = [2], target = 1

Output: []

Constraints:

1 <= candidates.length <= 30

2 <= candidates[i] <= 40

All elements of candidates are distinct.

1 <= target <= 40

**Solution :**

class Solution {

    List<List<Integer>> res = new ArrayList<>();

    public void combSum(int[] candidates,int idx,int target,ArrayList<Integer> l)

    {

        if(target==0)

        {

            res.add(new ArrayList<>(l));

            return ;

        }

        if(idx==candidates.length || target < 0) return ;

        l.add(candidates[idx]);

        combSum(candidates,idx,target-candidates[idx],l);

        l.remove(l.size()-1);

        combSum(candidates,idx+1,target,l);

    }

    public List<List<Integer>> combinationSum(int[] candidates, int target) {

        combSum(candidates,0,target,new ArrayList<>());

        return res;

    }

}

**5) Combinations (Leetcode - 77)**

Given two integers n and k, return all possible combinations of k numbers chosen from the range [1, n].

You may return the answer in any order.

Example 1:

Input: n = 4, k = 2

Output: [[1,2],[1,3],[1,4],[2,3],[2,4],[3,4]]

Explanation: There are 4 choose 2 = 6 total combinations.

Note that combinations are unordered, i.e., [1,2] and [2,1] are considered to be the same combination.

Example 2:

Input: n = 1, k = 1

Output: [[1]]

Explanation: There is 1 choose 1 = 1 total combination.

Constraints:

1 <= n <= 20

1 <= k <= n

**Solution :**

import java.util.\*;

public class Solution {

    List<List<Integer>> res = new ArrayList<>();

//1st solution

    void solve1(int num, int tot, int k, List<Integer> ans) {

        if (num == tot + 1) {

            if (ans.size() == k) {

                res.add(new ArrayList<>(ans));

            }

            return;

        }

        ans.add(num);

        solve1(num + 1, tot, k, ans);

        ans.remove(ans.size() - 1);

        solve1(num + 1, tot, k, ans);

    }

//2nd solution

    void solve2(int num, int tot, int k, List<Integer> ans) {

        if (ans.size() == k) {

            res.add(new ArrayList<>(ans));

            return;

        }

        for (int i = num; i <= tot; i++) {

            ans.add(i);

            solve2(i + 1, tot, k, ans);

            ans.remove(ans.size() - 1);

        }

    }

    public List<List<Integer>> combine(int n, int k) {

        List<Integer> ans = new ArrayList<>();

        solve2(1, n, k, ans);

        return res;

    }

}

**6) Combination Sum III (Leetcode – 216)**

Find all valid combinations of k numbers that sum up to n such that the following conditions are true:

Only numbers 1 through 9 are used.

Each number is used at most once.

Return a list of all possible valid combinations. The list must not contain the same combination twice, and the combinations may be returned in any order.

Example 1:

Input: k = 3, n = 7

Output: [[1,2,4]]

Explanation:

1 + 2 + 4 = 7

There are no other valid combinations.

Example 2:

Input: k = 3, n = 9

Output: [[1,2,6],[1,3,5],[2,3,4]]

Explanation:

1 + 2 + 6 = 9

1 + 3 + 5 = 9

2 + 3 + 4 = 9

There are no other valid combinations.

Example 3:

Input: k = 4, n = 1

Output: []

Explanation: There are no valid combinations.

Using 4 different numbers in the range [1,9], the smallest sum we can get is 1+2+3+4 = 10 and since 10 > 1, there are no valid combination.

Constraints:

2 <= k <= 9

1 <= n <= 60

**Solution :**

class Solution {

    public static void comb(int k,int sum,int i,ArrayList<Integer> l,List<List<Integer>> r,int n)

    {

        if(k==0 && sum==0)

        {

            r.add(new ArrayList<>(l));

            return ;

        }

        if(i>n || k<0 || sum<0)

        {

            return ;

        }

        l.add(i);

        comb(k-1,sum-i,i+1,l,r,n);

        l.remove(l.size()-1);

        comb(k,sum,i+1,l,r,n);

    }

    public List<List<Integer>> combinationSum3(int k, int n) {

        List<List<Integer>> res = new ArrayList<>();

        ArrayList<Integer> l = new ArrayList<>();

        comb(k,n,1,l,res,9);

        return res;

    }

}

**7) Generate Parentheses (Leetcode - 22)**

Given n pairs of parentheses, write a function to generate all combinations of well-formed parentheses.

Example 1:

Input: n = 3

Output: ["((()))","(()())","(())()","()(())","()()()"]

Example 2:

Input: n = 1

Output: ["()"]

Constraints:

1 <= n <= 8

**Solution :**

class Solution {

     public static List<String> bal(int n,int oc,int cc,int i,char[] l,List<String> s)

    {

        if(i==n)

        {

            s.add(String.copyValueOf(l));

        }

        if(oc<n/2)

        {

            l[i]='(';

            bal(n,oc+1,cc,i+1,l,s);

        }

        if(oc>cc)

        {

            l[i]=')';

            bal(n,oc,cc+1,i+1,l,s);

        }

        return s;

    }

    public List<String> generateParenthesis(int n) {

        char[] l=new char[n+n];

        List<String> s = new ArrayList<>();

        return bal(n+n,0,0,0,l,s);

    }

}

**8) Palindrome Partitioning (Leetcode - 131)**

Given a string s, partition s such that every substring of the partition is a palindrome. Return all possible palindrome partitioning of s.

Example 1:

Input: s = "aab"

Output: [["a","a","b"],["aa","b"]]

Example 2:

Input: s = "a"

Output: [["a"]]

Constraints:

1 <= s.length <= 16

s contains only lowercase English letters.

**Solution :**

class Solution {

    public void part(String s,ArrayList l,List<List<String>> res,int in)

    {

        if(in>=s.length())

        {

            res.add(new ArrayList<>(l));

            return;

        }

        for(int i=in;i<s.length();i++)

        {

            if (!isPalindrome(s.substring(in,i+1))) continue;

            l.add(s.substring(in,i+1));

            part(s,l,res,i+1);

            l.remove(l.size()-1);

        }

    }

     boolean isPalindrome(String s) {

        int lower = 0;

        int higher = s.length() - 1;

        while (lower < higher) {

            if (s.charAt(lower) != s.charAt(higher)) return false;

            lower++;

            higher--;

        }

        return true;

    }

    public List<List<String>> partition(String s) {

        ArrayList<String> l = new ArrayList<>();

        List<List<String>> res = new ArrayList<>();

        part(s,l,res,0);

        return res;

    }

}

**9) Letter Combinations of a Phone Number (Leetcode - 17)**

Given a string containing digits from 2-9 inclusive, return all possible letter combinations that the number could represent. Return the answer in any order.

A mapping of digits to letters (just like on the telephone buttons) is given below. Note that 1 does not map to any letters.

Example 1:



Input: digits = "23"

Output: ["ad","ae","af","bd","be","bf","cd","ce","cf"]

Example 2:

Input: digits = ""

Output: []

Example 3:

Input: digits = "2"

Output: ["a","b","c"]

Constraints:

0 <= digits.length <= 4

digits[i] is a digit in the range ['2', '9'].

**Solution :**

class Solution {

    private static final String[] KEYPAD = {

        "", "", "abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz"

    };

    public List<String> letterCombinations(String digits) {

        List<String> combinations = new ArrayList<>();

        if (digits == null || digits.isEmpty()) {

            return combinations;

        }

        generateCombinations(combinations, new StringBuilder(), digits, 0);

        return combinations;

    }

    private void generateCombinations(List<String> combinations, StringBuilder current, String digits, int index) {

        if (index == digits.length()) {

            combinations.add(current.toString());

            return;

        }

        String letters = KEYPAD[digits.charAt(index) - '0'];

        for (char letter : letters.toCharArray()) {

            current.append(letter);

            generateCombinations(combinations, current, digits, index + 1);

            current.deleteCharAt(current.length() - 1);

        }

    }

}

**10) Subsets II (Leetcode - 90)**

Given an integer array nums that may contain duplicates, return all possible

subsets (the power set).

The solution set must not contain duplicate subsets. Return the solution in any order.

Example 1:

Input: nums = [1,2,2]

Output: [[],[1],[1,2],[1,2,2],[2],[2,2]]

Example 2:

Input: nums = [0]

Output: [[],[0]]

Constraints:

1 <= nums.length <= 10

-10 <= nums[i] <= 10

**Solution :**

**//another way to avoid dublicates is 11th (combination sum 2) using while**

class Solution {

    List<List<Integer>> res = new ArrayList<>();

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

    public void combinations(int[] nums,int idx,ArrayList<Integer> l)

    {

        if(idx==nums.length)

        {

            if(!set.contains(l))

            {

                res.add(new ArrayList<>(l));

                set.add(l);

            }

            return ;

        }

        l.add(nums[idx]);

        combinations(nums,idx+1,l);

        l.remove(l.size()-1);

        combinations(nums,idx+1,l);

    }

    public List<List<Integer>> subsetsWithDup(int[] nums) {

        Arrays.sort(nums);

        combinations(nums,0,new ArrayList<>());

        return res;

    }

}

**11) Combination Sum II (Leetcode - 40)**

Given a collection of candidate numbers (candidates) and a target number (target), find all unique combinations in candidates where the candidate numbers sum to target.

Each number in candidates may only be used once in the combination.

Note: The solution set must not contain duplicate combinations.

Example 1:

Input: candidates = [10,1,2,7,6,1,5], target = 8

Output:

[

[1,1,6],

[1,2,5],

[1,7],

[2,6]

]

Example 2:

Input: candidates = [2,5,2,1,2], target = 5

Output:

[

[1,2,2],

[5]

]

Constraints:

1 <= candidates.length <= 100

1 <= candidates[i] <= 50

1 <= target <= 30

**Solution :**

class Solution {

    List<List<Integer>> res = new ArrayList<>();

    public void combSum(int[] candidates,int idx,int target,ArrayList<Integer> l)

    {

        if(target==0)

        {

                res.add(new ArrayList<>(l));

            return ;

        }

        if(idx==candidates.length || target < 0) return ;

        l.add(candidates[idx]);

        combSum(candidates,idx+1,target-candidates[idx],l);

        l.remove(l.size()-1);

        while (idx + 1 < candidates.length && candidates[idx] == candidates[idx + 1]) {

            idx++;

        }

        combSum(candidates,idx+1,target,l);

    }

    public List<List<Integer>> combinationSum2(int[] candidates, int target) {

        Arrays.sort(candidates);

        combSum(candidates,0,target,new ArrayList<>());

        return res;

    }

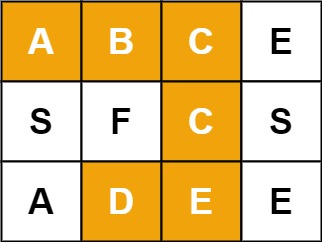
}

**12) Word Search (Leetcode - 79)**

Given an m x n grid of characters board and a string word, return true if word exists in the grid.

The word can be constructed from letters of sequentially adjacent cells, where adjacent cells are horizontally or vertically neighboring. The same letter cell may not be used more than once.

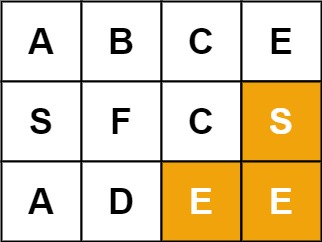
Example 1:



Input: board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "ABCCED"

Output: true

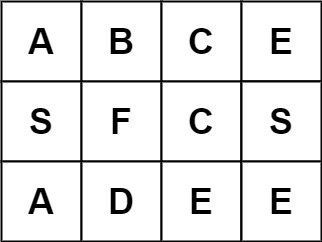
Example 2:



Input: board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "SEE"

Output: true

Example 3:



Input: board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "ABCB"

Output: false

Constraints:

m == board.length

n = board[i].length

1 <= m, n <= 6

1 <= word.length <= 15

board and word consists of only lowercase and uppercase English letters.

**Solution 1:**

class Solution {

    private boolean r =false;

    public void search(char[][] board,String word,int i,int j,boolean[][] vis,int in)

    {

        if(in==word.length())

        {

            r=true;

            return ;

        }

        int n=board.length;

        int m=board[0].length;

        if(i+1 < n && !vis[i+1][j] && word.charAt(in)==board[i+1][j])

        {

            vis[i][j]=true;

            search(board,word,i+1,j,vis,in+1);

            vis[i][j]=false;

        }

        if(j-1>=0 && !vis[i][j-1] && word.charAt(in)==board[i][j-1])

        {

            vis[i][j]=true;

            search(board,word,i,j-1,vis,in+1);

            vis[i][j]=false;

        }

        if(i-1>=0 && !vis[i-1][j] && word.charAt(in)==board[i-1][j])

        {

            vis[i][j]=true;

            search(board,word,i-1,j,vis,in+1);

            vis[i][j]=false;

        }

        if(j+1 < m && !vis[i][j+1] && word.charAt(in)==board[i][j+1])

        {

            vis[i][j]=true;

            search(board,word,i,j+1,vis,in+1);

            vis[i][j]=false;

        }

    }

    public boolean exist(char[][] board, String word) {

        int n=board.length;

        int m=board[0].length;

        boolean[][] vis=new boolean[n][m];

        char c=word.charAt(0);

        for(int i=0;i<n;i++)

        {

            for(int j=0;j<m;j++)

            {

                if(board[i][j]==c)

                {

                    search(board,word,i,j,vis,1);

                }

            }

        }

        return r;

    }

}

**Solution 2:**

public class Solution {

    public boolean exist(char[][] board, String word) {

        int m = board.length;

        int n = board[0].length;

        boolean[][] visited = new boolean[m][n];

        boolean result = false;

        for (int i = 0; i < m; i++) {

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

                if (board[i][j] == word.charAt(0)) {

                    result = backtrack(board, word, visited, i, j, 0);

                    if (result) return true;

                }

            }

        }

        return false;

    }

    private boolean backtrack(char[][] board, String word, boolean[][] visited, int i, int j, int index) {

        if (index == word.length()) {

            return true;

        }

        if (i < 0 || i >= board.length || j < 0 || j >= board[0].length || visited[i][j] || board[i][j] != word.charAt(index)) {

            return false;

        }

        visited[i][j] = true;

        if (backtrack(board, word, visited, i + 1, j, index + 1) ||

            backtrack(board, word, visited, i - 1, j, index + 1) ||

            backtrack(board, word, visited, i, j + 1, index + 1) ||

            backtrack(board, word, visited, i, j - 1, index + 1)) {

            return true;

        }

        visited[i][j] = false;

        return false;

    }

}

**13) Valid Sudoku (Leetcode - 36)**

Determine if a 9 x 9 Sudoku board is valid. Only the filled cells need to be validated according to the following rules:

Each row must contain the digits 1-9 without repetition.

Each column must contain the digits 1-9 without repetition.

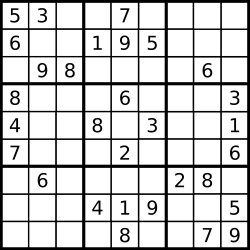
Each of the nine 3 x 3 sub-boxes of the grid must contain the digits 1-9 without repetition.

Note:

A Sudoku board (partially filled) could be valid but is not necessarily solvable.

Only the filled cells need to be validated according to the mentioned rules.

Example 1:



Input: board =

[["5","3",".",".","7",".",".",".","."]

,["6",".",".","1","9","5",".",".","."]

,[".","9","8",".",".",".",".","6","."]

,["8",".",".",".","6",".",".",".","3"]

,["4",".",".","8",".","3",".",".","1"]

,["7",".",".",".","2",".",".",".","6"]

,[".","6",".",".",".",".","2","8","."]

,[".",".",".","4","1","9",".",".","5"]

,[".",".",".",".","8",".",".","7","9"]]

Output: true

Example 2:

Input: board =

[["8","3",".",".","7",".",".",".","."]

,["6",".",".","1","9","5",".",".","."]

,[".","9","8",".",".",".",".","6","."]

,["8",".",".",".","6",".",".",".","3"]

,["4",".",".","8",".","3",".",".","1"]

,["7",".",".",".","2",".",".",".","6"]

,[".","6",".",".",".",".","2","8","."]

,[".",".",".","4","1","9",".",".","5"]

,[".",".",".",".","8",".",".","7","9"]]

Output: false

Explanation: Same as Example 1, except with the 5 in the top left corner being modified to 8. Since there are two 8's in the top left 3x3 sub-box, it is invalid.

Constraints:

board.length == 9

board[i].length == 9

board[i][j] is a digit 1-9 or '.'.

**Solution :**

class Solution {

    public boolean isValidSudoku(char[][] board) {

     for(int i=0;i<9;i++)

     {

         for(int j=0;j<9;j++)//traverse in bfs manner

         {

             char ch=board[i][j];

             if(ch!='.')//if ch is apart from '.'we need to check whether the given number is verifiable

             {

                 board[i][j]='.';//change it to . so that it cannot consider itself

                 if(!isValid(board,ch,i,j))//if not verifiable

                     return false;//invalid sudoku

                 board[i][j]=ch;//after checking change it to its initial value

             }

         }

     }

        return true;

    }

    static boolean isValid(char[][]ch,char ch1,int i,int j)

    {

      for(int k=0;k<9;k++)

      {

          if(ch[i][k]==ch1)return false;//for checking row associated with it

          if(ch[k][j]==ch1)return false;//for checking column associated with it

          if(ch[3\*(i/3)+k/3][3\*(j/3)+k%3]==ch1)return false;//for checking block associated with it

      }

        return true;

    }

}

**14)Sudoku Solver (Leetcode - 37)**

Write a program to solve a Sudoku puzzle by filling the empty cells.

A sudoku solution must satisfy all of the following rules:

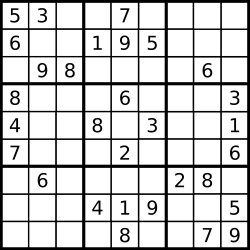
Each of the digits 1-9 must occur exactly once in each row.

Each of the digits 1-9 must occur exactly once in each column.

Each of the digits 1-9 must occur exactly once in each of the 9 3x3 sub-boxes of the grid.

The '.' character indicates empty cells.

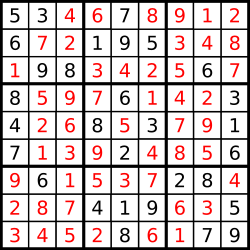
Example 1:



Input: board = [["5","3",".",".","7",".",".",".","."],["6",".",".","1","9","5",".",".","."],[".","9","8",".",".",".",".","6","."],["8",".",".",".","6",".",".",".","3"],["4",".",".","8",".","3",".",".","1"],["7",".",".",".","2",".",".",".","6"],[".","6",".",".",".",".","2","8","."],[".",".",".","4","1","9",".",".","5"],[".",".",".",".","8",".",".","7","9"]]

Output: [["5","3","4","6","7","8","9","1","2"],["6","7","2","1","9","5","3","4","8"],["1","9","8","3","4","2","5","6","7"],["8","5","9","7","6","1","4","2","3"],["4","2","6","8","5","3","7","9","1"],["7","1","3","9","2","4","8","5","6"],["9","6","1","5","3","7","2","8","4"],["2","8","7","4","1","9","6","3","5"],["3","4","5","2","8","6","1","7","9"]]

Explanation: The input board is shown above and the only valid solution is shown below:



Constraints:

board.length == 9

board[i].length == 9

board[i][j] is a digit or '.'.

It is guaranteed that the input board has only one solution.

**Solution :**

class Solution {

    public boolean solve(int row,int col,char[][] board)

    {

        if(row==board.length) return true; // row is greater than row length

        if(col==board[0].length) return solve(row+1,0,board);

        if(board[row][col]!='.') return solve(row,col+1,board);

       for(char i='1';i<='9';i++)

       {

            if(isValid(row,col,i,board))

            {

                board[row][col]=i;

                if(solve(row,col+1,board)) return true;

            }

            board[row][col]='.';

       }

        return false;

    }

    public boolean isValid(int row,int col,char val,char[][] board)

    {

        for(int i=0;i<9;i++)

        {

            if(board[row][i]==val || board[i][col]==val || board[3\*(row/3)+i/3][3\*(col/3)+i%3]==val)

            {

                return false;

            }

        }

        return true;

    }

    public void solveSudoku(char[][] board) {

        solve(0,0,board);

    }

}

**15) N-Queens (Leetcode - 51)**

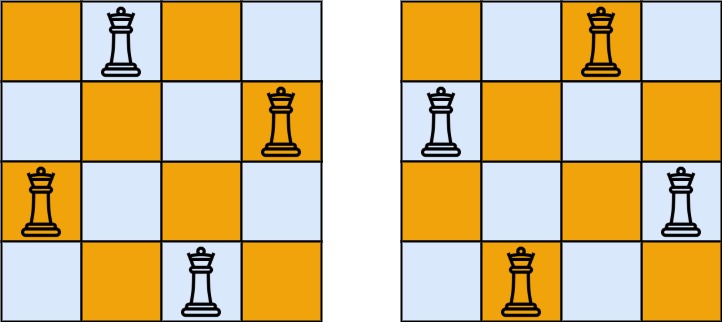
The n-queens puzzle is the problem of placing n queens on an n x n chessboard such that no two queens attack each other.

Given an integer n, return all distinct solutions to the n-queens puzzle. You may return the answer in any order.

Each solution contains a distinct board configuration of the n-queens' placement, where 'Q' and '.' both indicate a queen and an empty space, respectively.

Example 1:

Input: n = 4



Output: [[".Q..","...Q","Q...","..Q."],["..Q.","Q...","...Q",".Q.."]]

Explanation: There exist two distinct solutions to the 4-queens puzzle as shown above

Example 2:

Input: n = 1

Output: [["Q"]]

Constraints:

1 <= n <= 9

**Solution :**

class Solution {

    public boolean check(int row,int col,char[][] queens)

    {

**//column check**

        for(int i=0;i<row;i++)

        {

            if(queens[i][col]=='Q')

            {

                return false;

            }

        }

**//45 degree check**

        for(int i=row-1,j=col-1;i>=0&&j>=0;i--,j--)

        {

            if(queens[i][j]=='Q')

            {

                return false;

            }

        }

**//135 degree check**

        for(int i=row-1,j=col+1;i>=0&&j<queens[0].length;i--,j++)

        {

            if(queens[i][j]=='Q')

            {

                return false;

            }

        }

        return true;

    }

    public void backTrack(int row,char[][] queens,List<List<String>> res)

    {

        if(row==queens.length)

        {

            res.add(convert(queens));

            return ;

        }

        for(int col=0;col<queens[0].length;col++)

        {

            if(check(row,col,queens))

            {

                queens[row][col]='Q';

                backTrack(row+1,queens,res);

                queens[row][col]='.';

            }

        }

    }

**//convert queen array to like this ..Q.**

    public List<String> convert(char[][] queens)

    {

        List<String> l = new ArrayList<>();

        for(char[] i:queens)

        {

            l.add(new String(i));

        }

        return l;

    }

    public List<List<String>> solveNQueens(int n) {

        List<List<String>> res = new ArrayList<>();

        char[][] queens = new char[n][n];

        for(int i=0;i<n;i++)

        {

            for(int j=0;j<n;j++)

            {

                queens[i][j]='.';

            }

        }

        backTrack(0,queens,res);

        return res;

    }

}

**16) N-Queens II (Leetcode - 52)**

The n-queens puzzle is the problem of placing n queens on an n x n chessboard such that no two queens attack each other.

Given an integer n, return the number of distinct solutions to the n-queens puzzle.

Example 1:

Input: n = 4

Output: 2

Explanation: There are two distinct solutions to the 4-queens puzzle as shown.

Example 2:

Input: n = 1

Output: 1

Constraints:

1 <= n <= 9

**Solution : (Same as Nqueen above)**

class Solution {

    public boolean check(int row,int col,char[][] queens)

    {

**//column check**

        for(int i=0;i<row;i++)

        {

            if(queens[i][col]=='Q')

            {

                return false;

            }

        }

**//45 degree check**

        for(int i=row-1,j=col-1;i>=0&&j>=0;i--,j--)

        {

            if(queens[i][j]=='Q')

            {

                return false;

            }

        }

**//135 degree check**

        for(int i=row-1,j=col+1;i>=0&&j<queens[0].length;i--,j++)

        {

            if(queens[i][j]=='Q')

            {

                return false;

            }

        }

        return true;

    }

    public int backTrack(int row,char[][] queens)

    {

        if(row==queens.length)

        {

            return 1;

        }

        int count=0;

        for(int col=0;col<queens[0].length;col++)

        {

            if(check(row,col,queens))

            {

                queens[row][col]='Q';

                count+=backTrack(row+1,queens);

                queens[row][col]=' ';

            }

        }

        return count;

    }

    public int totalNQueens(int n) {

        char[][] queens = new char[n][n];

        return backTrack(0,queens);

    }

}