15. 3Sum

Given an array `nums` of *n* integers, are there elements *a*, *b*, *c* in `nums` such that *a* + *b* + *c* = 0? Find all unique triplets in the array which gives the sum of zero.

Notice that the solution set must not contain duplicate triplets.

Example 1:

```
Input: nums = [-1,0,1,2,-1,-4]
Output: [[-1,-1,2],[-1,0,1]]
```

Example 2:

```
Input: nums = []
Output: []
```

Example 3:

```
Input: nums = [0]
Output: []
```

Constraints:

- `0 <= nums.length <= 3000`
- `-10^5 <= nums[i] <= 10^5`

```
class Solution {
    public List<List<Integer>> threeSum(int[] nums) {
        Arrays.sort(nums);
        List<List<Integer>> ans=new ArrayList<>();
        int n=nums.length;
        for(int i=0;i<n-1;i++)
        {
            int left=i+1;
            int right=n-1;
            int sum=-nums[i];
            while(left<right)
            {
                int twosum=nums[left]+nums[right];
                if(twosum==sum)
                {
                    ans.add(new ArrayList<Integer>(Arrays.asList(nums[i],nums[left],n
ums[right])));
                    while(left<right && nums[left]==nums[left+1])
                        left++;
                    while(left<right && nums[right]==nums[right-1])
                        right--;
                    left++;
                    right--;
                }
                else if(twosum<sum)
                {
                    left++;
                }
                else
                {
                    right--;
                }
            }
            while(i<n-1 && nums[i]==nums[i+1])
                i++;
        }
        return ans;
    }
}
```

18. 4Sum

Given an array `nums` of *n* integers and an integer `target`, are there elements *a*, *b*, *c*, and *d* in `nums` such that *a* + *b* + *c* + *d* = `target`? Find all unique quadruplets in the array which gives the sum of `target`.

Notice that the solution set must not contain duplicate quadruplets.

Example 1:

```
Input: nums = [1,0,-1,0,-2,2], target = 0
Output: [[-2,-1,1,2],[-2,0,0,2],[-1,0,0,1]]
```

Example 2:

```
Input: nums = [], target = 0
Output: []
```

Constraints:

- 0 <= nums.length <= 200
- -10^9 <= nums[i] <= 10^9
- -10^9 <= target <= 10^9

```
class Solution {
    public List<List<Integer>> fourSum(int[] A, int B) {
        int n=A.length;
        List<List<Integer>> ans=new ArrayList<>();
        if(n<4)
          return ans;
        Arrays.sort(A);
        for(int i=0;i<n-3;i++)
        {
            for(int j=i+1;j<n-2;j++)
            {
                int sum=B-(A[i]+A[j]);
                int left=j+1;
                int right=n-1;
                while(left<right)
                {
                    int tofind=A[left]+A[right];
                    if(tofind==sum)
                    {
                        ans.add(new ArrayList<>(Arrays.asList(A[i],A[j],A[left],A[rig
ht])));
                        while(left<right && A[left]==A[left+1])
                          left++;
                        while(left<right && A[right]==A[right-1])
                          right--;
                        left++;
                        right--;
                    }
                    else if(tofind<sum)
                    {
                        left++;
                    }
                    else
                    {
                        right--;
                    }
                }
                while(j<n-1 && A[j]==A[j+1])
                  j++;
            }
            while(i<n-1 && A[i]==A[i+1])
                  i++;
        }
        return ans;
    }
}
```

31. Next Permutation

Implement **next permutation**, which rearranges numbers into the lexicographically next greater permutation of numbers.

If such an arrangement is not possible, it must rearrange it as the lowest possible order (i.e., sorted in ascending order).

The replacement must be **in place (http://en.wikipedia.org/wiki/In-place_algorithm)** and use only constant extra memory.

Example 1:

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

Example 2:

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

Example 3:

```
Input: nums = [1,1,5]
Output: [1,5,1]
```

Example 4:

```
Input: nums = [1]
Output: [1]
```

Constraints:

- `1 <= nums.length <= 100`
- `0 <= nums[i] <= 100`

/*

```
Approach:
As we know unit place has lower value than ten place and so on
so we have to start from right(from unit place)
and find first number which is smaller than previous right digit(we will change this
number)
We will swap this number with just greater number and remaining elemnts sould be sort
ed
But as they are already in decresing order we just havr to reverse them
watch take u foeward video fro intution

Basically 3 steps:
1) find first smallest digit(number) in array which is smaller than its right digit
2) Swap this FIRST smaller number with number just greater than it from right aide of
array
3) from next index of first smaller number sort ascending order

*/
```

```java
//gfg solution
class Solution{
    static List<Integer> nextPermutation(int N, int arr[]){
        int back=-1;
        for(int i=N-2;i>=0;i--)
        {
            if(arr[i]<arr[i+1])
            {
                back=i;
                break;
            }
        }
        int swap=N-1;
        for(int i=N-1;i>back && back!=-1;i--)
        {
            if(arr[i]>arr[back])
            {
                swap=i;
                int temp=arr[swap];
                arr[swap]=arr[back];
                arr[back]=temp;
                break;
            }
        }
        for(int i=0;i<(N-back-1)/2;i++)
        {
            int t=arr[i+back+1];
            arr[i+back+1]=arr[N-1-i];
            arr[N-1-i]=t;
        }
        ArrayList<Integer> ans=new ArrayList<>();
        for(int num:arr)
          ans.add(num);
        return ans;
    }
}
```

```
//Leetcode solution
class Solution {
    public void nextPermutation(int[] nums) {
        int i=0;
        int index=-1;
        int n=nums.length;
        while(i<n-1)
        {
            if(nums[i]<nums[i+1])
                index=i;
            i++;
        }
        int last=n-1;
        while(last>=index && index!=-1)
        {
            if(nums[last]>nums[index])
            {
                int temp=nums[last];
                nums[last]=nums[index];
                nums[index]=temp;
                break;
            }
            last--;
        }
        int j=n-1;
        i=index;
        i++;
        while(i<j)
        {
            int temp=nums[i];
            nums[i]=nums[j];
            nums[j]=temp;
            j--;
            i++;
        }
    }
}
```

---

41. First Missing Positive

Given an unsorted integer array `nums`, find the smallest missing positive integer.

Example 1:

```
Input: nums = [1,2,0]
Output: 3
```

Example 2:

```
Input: nums = [3,4,-1,1]
Output: 2
```

Example 3:

```
Input: nums = [7,8,9,11,12]
Output: 1
```

Constraints:

- `0 <= nums.length <= 300`
- `-2^31 <= nums[i] <= 2^31 - 1`

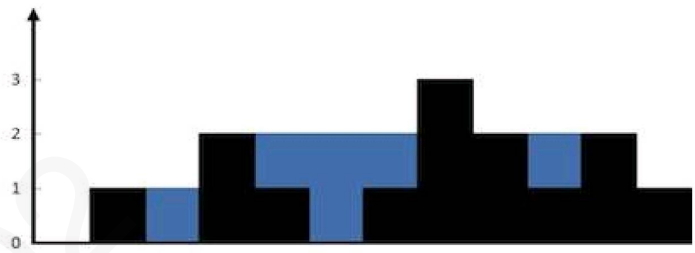
Follow up: Could you implement an algorithm that runs in `O(n)` time and uses constant extra space?

```
class Solution {
    public int firstMissingPositive(int[] A) {
        int n=A.length;
        for(int i=0;i<n;i++)
        {
            if(A[i]<0 || A[i]>n)
            {
                A[i]=0;
            }
        }
        for(int i=0;i<n;i++)
        {
            if((A[i]%(n+1))>=1 && (A[i]%(n+1))<=n)
              A[(A[i]%(n+1))-1]+=(n+1);
        }
        for(int i=0;i<n;i++)
        {
            if(A[i]<(n+1))
              return i+1;
        }
        return n+1;
    }
}
```

42. Trapping Rain Water

Given `n` non-negative integers representing an elevation map where the width of each bar is `1`, compute how much water it can trap after raining.

Example 1:




```
Input: height = [0,1,0,2,1,0,1,3,2,1,2,1]
Output: 6
Explanation: The above elevation map (black section) is represented by array [0,1,0,2,1
```

Example 2:

```
Input: height = [4,2,0,3,2,5]
Output: 9
```

Constraints:

- `n == height.length`
- `0 <= n <= 3 * 10^4`
- `0 <= height[i] <= 10^5`

/*Explanation: Maintain 2 arrays left and right Left array : It will store maximum value available at left side of i in array for every element at ith position in array Right array : It will store maximum value available at right side of i in array for every element at ith position in array so for every i (height in array): we will subtract height at i from (min of (left at i ,right at i)) add above to answer

basically we are creating walls of height at left and right at every position so water can store in that wall for every i and selecting min wall

we can solve above problem using two point approach as well. with constant space. */

```
class Solution {
    public int trap(int[] height) {
        int n=height.length;
        if(n==0)
            return 0;
        int left[]=new int[n];
        int right[]=new int[n];
        left[0]=height[0];
        right[n-1]=height[n-1];
        for(int i=1;i<n;i++)
        {
            left[i]=Math.max(left[i-1],height[i]);
        }
        for(int j=n-2;j>=0;j--)
        {
            right[j]=Math.max(right[j+1],height[j]);
        }
        int ans=0;
        for(int i=0;i<n;i++)
        {
            ans+=Math.min(left[i],right[i])-height[i];
        }
        return ans;
    }
}
```

```
//Two pointer approach

public class Solution {
    public int trap(final int[] A) {
        int n=A.length;
        if(n==0)
            return 0;
        int ans=0;
        int l=0;
        int lmax=A[0];
        int rmax=A[n-1];
        int r=n-1;
        while(l<r)
        {
            if(A[l]<A[r])
            {
                lmax=Math.max(A[l],lmax);
                ans+=(lmax-A[l]);
                l++;
            }
            else
            {
                rmax=Math.max(A[r],rmax);
                ans+=(rmax-A[r]);
                r--;
            }
        }
        return ans;
    }
}
```

---

49. Group Anagrams

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:

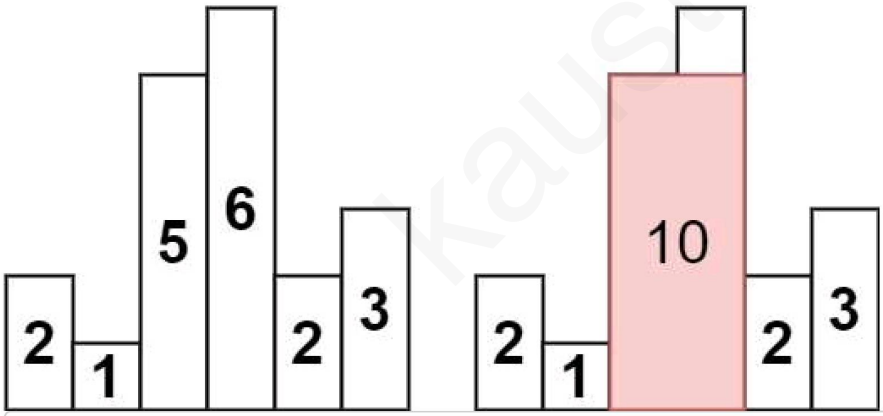
- `1 <= strs.length <= 10^4`
- `0 <= strs[i].length <= 100`
- `strs[i]` consists of lower-case English letters.

```
class Solution {
    public List<List<String>> groupAnagrams(String[] strs) {
        if(strs.length==0)
            return new ArrayList();
        HashMap<String,List> h=new HashMap<>();
        for(String s:strs)
        {
            char c[]=s.toCharArray();
            Arrays.sort(c);
            String modify=String.valueOf(c);
            if(h.containsKey(modify))
                h.get(modify).add(s);
            else
                h.put(modify,new ArrayList<>(Arrays.asList(s)));
        }
        return new ArrayList(h.values());
    }
}
```

84. Largest Rectangle in Histogram

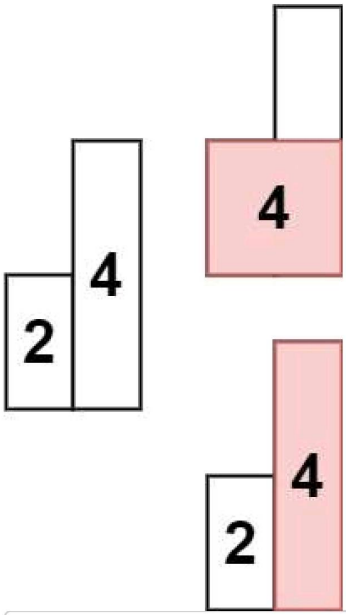
Given an array of integers `heights` representing the histogram's bar height where the width of each bar is `1`, return *the area of the largest rectangle in the histogram*.

Example 1:




```
Input: heights = [2,1,5,6,2,3]
Output: 10
Explanation: The above is a histogram where width of each bar is 1.
The largest rectangle is shown in the red area, which has an area = 10 units.
```

Example 2:



Input: heights = [2,4]
Output: 4

Constraints:

- 1 <= heights.length <= 10^5
- 0 <= heights[i] <= 10^4

```
Explanation:
1) Brute force will be O(n^2)
It is hard version of nearest smallest elemnt in left side of number in array
1. Basic idea is we will consider every bar in array
2. While traversing array we will conisder current bar as max height for rectangle
3. Now we have to found just smaller elemnt than above current elemnt from left and r
ight of array
4. Now our height become : curr elemnt
5. and width become : right-left+1
6. To find left and right element which is just smaller than current in O(1) we need
to precompute above in left[] and right array
7. left[i] : considering ith elemnt as max height in rectangle left[i] denotes just s
maller elemnt than currnet max height position in array
8. right[i] : same as left[i] but denotes from right part of array
9. We will use stack for precomputation
    //Watch solution of 503 from github nearest smaller elemnt
10. Take care of corner cases
```

```
class Solution {
    public int largestRectangleArea(int[] heights) {
        int ans=0;
        int n=heights.length;
        Stack<Integer> s=new Stack<Integer>();
        int left[]=new int[n];
        int right[]=new int[n];
        for(int i=0;i<n;i++)
        {
            while(!s.isEmpty() && heights[s.peek()]>=heights[i])
                s.pop();
            left[i]=s.isEmpty()?0:s.peek()+1;
            s.push(i);
        }
        s=new Stack<Integer>();
        for(int i=n-1;i>=0;i--)
        {
            while(!s.isEmpty() && heights[s.peek()]>=heights[i])
                s.pop();
            right[i]=s.isEmpty()?n-1:s.peek()-1;
            s.push(i);
        }
        for(int i=0;i<n;i++)
            ans=Math.max(ans,(right[i]-left[i]+1)*heights[i]);
        return ans;
    }
}
```

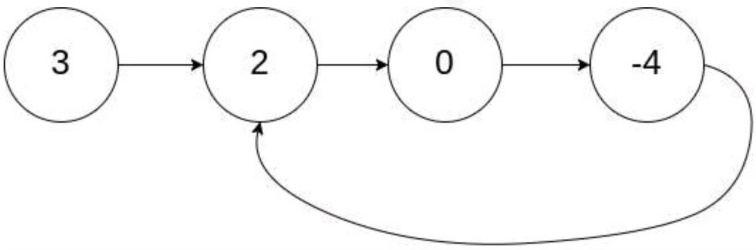
142. Linked List Cycle II

Given a linked list, return the node where the cycle begins. If there is no cycle, return `null`.

There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the `next` pointer. Internally, `pos` is used to denote the index of the node that tail's `next` pointer is connected to. **Note that `pos` is not passed as a parameter**.

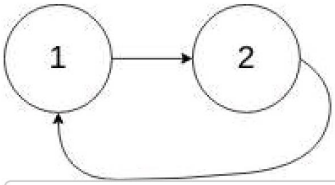
Notice that you **should not modify** the linked list.

Example 1:




```
Input: head = [3,2,0,-4], pos = 1
Output: tail connects to node index 1
Explanation: There is a cycle in the linked list, where tail connects to the second nod
```

Example 2:




```
Input: head = [1,2], pos = 0
Output: tail connects to node index 0
Explanation: There is a cycle in the linked list, where tail connects to the first node
```

Example 3:


```
Input: head = [1], pos = -1
Output: no cycle
Explanation: There is no cycle in the linked list.
```

Constraints:

- The number of the nodes in the list is in the range `[0, 10^4]`.
- `-10^5 <= Node.val <= 10^5`
- `pos` is `-1` or a **valid index** in the linked-list.

Follow up: Can you solve it using `O(1)` (i.e. constant) memory?

```
/*
Explanation:
First we will find whether cycle present or not
For that we will use slow and fast pointer
slow pointer==moves by one position
fast pointer==moves by two position

Suppose if cycle is present then any pointer will not reach null
After some time slow and fast pointer meet if cycle is there

Then we have to find length of cycle ...why??
Suppose some how yo find length of cycle then your question turn into
find elemnt at distance n(length of cycle) from end of linkedlist
differnce here is end of list is not null but common pointer

Now how to find length??
You can initiate variable l for length and
now only travel slow pointer till again meet fast pointer and increment l

IMPORTANT:
but no need to find length again as we alreday know length
just move fast pointer to start and increment both slow and fast by 1 while they meet
each other
for intution watch Take u forward find missing number solution
*/
```

```java
public class Solution {
    public ListNode detectCycle(ListNode head) {
        int flag=1;
        ListNode slow=head;
        ListNode fast=head;
        while(fast!=null && fast.next!=null)
        {
            slow=slow.next;
            fast=fast.next.next;
            if(slow==fast)
            {
                flag=0;
                break;
            }
        }
        if(flag==1)
            return null;
        slow=head;
        while(fast!=slow)
        {
            slow=slow.next;
            fast=fast.next;
        }
        return slow;
    }
}
```

---

155. Min Stack

Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

Implement the `MinStack` class:

- `MinStack()` initializes the stack object.
- `void push(val)` pushes the element `val` onto the stack.
- `void pop()` removes the element on the top of the stack.
- `int top()` gets the top element of the stack.
- `int getMin()` retrieves the minimum element in the stack.

Example 1:

```
Input
["MinStack","push","push","push","getMin","pop","top","getMin"]
[[],[-2],[0],[-3],[],[],[],[]]

Output
[null,null,null,null,-3,null,0,-2]

Explanation
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin(); // return -3
minStack.pop();
minStack.top();    // return 0
minStack.getMin(); // return -2
```

Constraints:

- $-2^{31} <= val <= 2^{31} - 1$
- Methods `pop`, `top` and `getMin` operations will always be called on **non-empty** stacks.
- At most $3 * 10^4$ calls will be made to `push`, `pop`, `top`, and `getMin`.

```
/*
Explanation:
Normal stack can be perform pop(),push() and peek() in O(1)
But we have to implement min element in O(1)
So we will maintain one another stack for min element till that point
Note:We can also use pair object with only one stack
          pair<Integer,Integer> =pair(element,min_element till point)
*/
```

```
class MinStack {

    Stack<Integer> ans;
    Stack<Integer> min;
    public MinStack() {
        ans=new Stack<>();
        min=new Stack<>();
    }

    public void push(int x) {
        ans.push(x);
        if(min.isEmpty())
            min.push(x);
        else
            min.push(Math.min(min.peek(),x));
    }

    public void pop() {
        ans.pop();
        min.pop();
    }

    public int top() {
        return ans.peek();
    }

    public int getMin() {
        return min.peek();
    }
}
```

```
class MinStack {

    ArrayList<Integer> ans;
    ArrayList<Integer> min;
    public MinStack() {
        ans=new ArrayList<>();
        min=new ArrayList<>();
    }

    public void push(int x) {
        ans.add(x);
        if(min.size()==0)
            min.add(x);
        else
            min.add(Math.min(min.get(min.size()-1),x));
    }

    public void pop() {
        ans.remove(ans.size()-1);
        min.remove(min.size()-1);
    }

    public int top() {
        return ans.get(ans.size()-1);
    }

    public int getMin() {
        return min.get(ans.size()-1);
    }
}
```

```
Optimize: O(1) extra space (except one stack )
O(1) time

idea:
We required one stack
and one min varaible to store current elemnt

There are two cases:
1) push elemnt x is greater than current min
2) push elemnt x is smaller than current min

1) for case 1 min elemnt will not change so we will push elemnt x as it is
2) for case 2 min elemnt will change and min will be x
   but we have to preserve currnet min i.e stored in min varibale
     so we will store previous min in form of something in stack
     Now we will push x+x-min in stack and update min=x
     in stack elemnt=2*x-min and new min=x

Note : Observe carefully elements present in stack are not actual elemnts
It is same elemnt as input in stack if x (current elemnt) is grater than min
In case of x smaller than min modified value is pushed in stack

now how to retrive or pop()??

Again there are two cases:
One thin we have notice that if currnt top of stack is greater than min
then we need not to worry about min as above elemnt is not contributing to min and of
case 1 (read note)

In other case: now our min elemnt is top of stack (why?? beacuse in this condition we
are changing actual elemnt while pushing in stack)
we have push let top=2*x-min;  i.e min=2*x-top
in stack we have top (top elemnt of stack)
we have store min=x while pushing
so while poping we will change min=2*min (as min=x) -top
```

```
class MinStack {

    Stack<Long> ans;
    long min=Integer.MAX_VALUE;
    public MinStack() {
        ans=new Stack<>();
    }

    public void push(int x) {
        if(ans.isEmpty())
        {
            ans.push((long)x);
            min=x;
        }
        else if(x>=min)
            ans.push((long)x);
        else
        {
            ans.push((long)x+x-min);
            min=x;
        }
    }

    public void pop() {
        if(ans.peek()>=min)
            ans.pop();
        else
        {
            min=min+min-ans.pop();
        }
    }

    public int top() {
        if(ans.peek()>min)
            return (int)(long)ans.peek();
        return (int)min;
    }

    public int getMin() {
        return (int)min;
    }
}
```

---

205. Isomorphic Strings

Given two strings `s` and `t`, *determine if they are isomorphic*.

Two strings `s` and `t` are isomorphic if the characters in `s` can be replaced to get `t`.

All occurrences of a character must be replaced with another character while preserving the order of characters. No two characters may map to the same character, but a character may map to itself.

Example 1:

```
Input: s = "egg", t = "add"
Output: true
```

Example 2:

```
Input: s = "foo", t = "bar"
Output: false
```

Example 3:

```
Input: s = "paper", t = "title"
Output: true
```

Constraints:

- `1 <= s.length <= 5 * 10^4`
- `t.length == s.length`
- `s` and `t` consist of any valid ascii character.

```
Lets start with given:
1)characters in s can b replaced to get c
2)we have to map character from t to charater from s
3)No two characters may map to the same characte

From above we have conclude that we have to do 1:1 mapping
i.e exactly one char from t should be mappped with only one char from s

1) HashMap in java used for many :1 mapping
conisder hashmap(key,value)
here key is unique we have to make value also unique
we can check containsvalue function but it will take O(n) time

So we will use hashset for value to do above task in O(1)
hashmap(key,value) ==> key is unique
hashset(value) ==> value is unique
Now both are unique hence 1:1 mapping can be achieved

//Not sure about following
we can use graph
and check indegree and outdegree of every elemnt
if indegree or outdegree of graph is greater than 1 then 1:1 mapping no there
```

```
class Solution {
    public boolean isIsomorphic(String s, String t) {
        if(s.length()!=t.length())
            return false;
        int n=s.length();
        HashMap<Character,Character> h=new HashMap<>();
        HashSet<Character> hs=new HashSet<>();
        for(int i=0;i<n;i++)
        {
            char one=s.charAt(i);
            char two=t.charAt(i);
            if(h.containsKey(two))
            {
                if(h.get(two)!=one)
                    return false;
            }
            else if(hs.contains(one))
                return false;
            h.put(two,one);
            hs.add(one);
        }
        return true;
    }
}
```

//Nice solution without using hahsmap

```
public class Solution {
    public boolean isIsomorphic(String s1, String s2) {
        int[] m = new int[512];
        for (int i = 0; i < s1.length(); i++) {
            if (m[s1.charAt(i)] != m[s2.charAt(i)+256]) return false;
            m[s1.charAt(i)] = m[s2.charAt(i)+256] = i+1;
        }
        return true;
    }
}
```

---

215. Kth Largest Element in an Array

Given an integer array `nums` and an integer `k`, return *the* k^{th} *largest element in the array*.

Note that it is the k^{th} largest element in the sorted order, not the k^{th} distinct element.

Example 1:

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

Example 2:

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

Constraints:

- 1 <= k <= nums.length <= 10^4
- -10^4 <= nums[i] <= 10^4

```
//For understanding see implementation of heap
//We can also use Priority queue but time comlexity will be nlogn or nlogk

class Solution {
    public int findKthLargest(int[] nums, int k) {
        for(int i=(nums.length-2)/2;i>=0;i--)
          heapify(i,nums,nums.length);
        for(int i=0;i<k-1;i++)
            extract_min(nums,nums.length-i);
        return extract_min(nums,nums.length-k+1);
    }

   void heapify(int index,int a[],int n)
   {
       int idx=index,lc=2*index+1,rc=lc+1;
       if(lc<n && a[idx]<=a[lc])
          idx=lc;
       if(rc<n && a[idx]<=a[rc])
          idx=rc;
       if(idx!=index)
       {
          int temp=a[idx];
          a[idx]=a[index];
          a[index]=temp;
          heapify(idx,a,n);
       }
       return;
   }


   int extract_min(int a[],int n)
   {
       int minm=a[0];
       a[0]=a[n-1];
       n--;
       heapify(0,a,n);
       return minm;
   }

}
```

239. Sliding Window Maximum

You are given an array of integers `nums`, there is a sliding window of size `k` which is moving from the very left of the array to the very right. You can only see the `k` numbers in the window. Each time the sliding window moves right by one position.

Return *the max sliding window*.

Example 1:

```
Input: nums = [1,3,-1,-3,5,3,6,7], k = 3
Output: [3,3,5,5,6,7]
Explanation:
Window position                Max
---------------               -----
[1  3  -1] -3  5  3  6  7       3
 1 [3  -1  -3] 5  3  6  7       3
 1  3 [-1  -3  5] 3  6  7       5
 1  3  -1 [-3  5  3] 6  7       5
 1  3  -1  -3 [5  3  6] 7       6
 1  3  -1  -3  5 [3  6  7]      7
```

Example 2:

```
Input: nums = [1], k = 1
Output: [1]
```

Example 3:

```
Input: nums = [1,-1], k = 1
Output: [1,-1]
```

Example 4:

```
Input: nums = [9,11], k = 2
Output: [11]
```

Example 5:

```
Input: nums = [4,-2], k = 2
Output: [4]
```

Constraints:

- $1 <= nums.length <= 10^5$
- $-10^4 <= nums[i] <= 10^4$
- $1 <= k <= nums.length$

```
explanation from basic:
https://www.youtube.com/watch?v=xFJXtB5vSmM

My idea:
Brute force: O(n*k)
 we will find repeated maximum elemnts using two nexted loop
 outer loop from 1 to n and inner loop of 1 to i+k

Optimize:
Following two things we have to achieve:
1) Remove from front of array
2) Insert in back of array or window

From that we get we have to use queue
we will use double ended queue that also monotonous
i.e queue will be index of elemnts in decreasing order (as we want maximum elemnt onl
y)
thus maximum elemnt will be at front of queue (q.peek())
We will remove from front till index greater than i-k (to maintain window size)
and add elemnt while maintaining decrasing order

Note: maxium elemnt in iteration will be of previous window
so find max elemnt of last window explicitely outside loop
```

```
class Solution {
    public int[] maxSlidingWindow(int[] nums, int k) {
        Deque<Integer> q=new LinkedList<>();
        int n=nums.length;
        int ans[]=new int[n-k+1];
        for(int i=0;i<k;i++)
        {
            while(!q.isEmpty() && nums[i]>=nums[q.peekLast()])
              q.removeLast();
            q.addLast(i);
        }
        for(int i=k;i<n;i++)
        {
            ans[i-k]=nums[q.peek()];
            while(!q.isEmpty() && q.peek()<=i-k)
              q.removeFirst();
            while(!q.isEmpty() && nums[i]>=nums[q.peekLast()])
              q.removeLast();
            q.addLast(i);
        }
        ans[n-k]=(nums[q.peek()]);
        return ans;
    }
}
```

---

274. H-Index

Given an array of integers `citations` where `citations[i]` is the number of citations a researcher received for their `i`th paper, return compute the researcher's `h` **-index**.

According to the definition of h-index on Wikipedia (https://en.wikipedia.org/wiki/H-index): A scientist has an index `h` if `h` of their `n` papers have at least `h` citations each, and the other `n − h` papers have no more than `h` citations each.

If there are several possible values for `h`, the maximum one is taken as the `h` **-index**.

Example 1:

```
Input: citations = [3,0,6,1,5]
Output: 3
Explanation: [3,0,6,1,5] means the researcher has 5 papers in total and each of them ha
Since the researcher has 3 papers with at least 3 citations each and the remaining two
```

Example 2:

```
Input: citations = [1,3,1]
Output: 1
```

Constraints:

- `n == citations.length`
- `1 <= n <= 5000`
- `0 <= citations[i] <= 1000`

```
class Solution {
    public int hIndex(int[] citations) {
        int n=citations.length;
        Arrays.sort(citations);
        for(int i=0;i<n;i++)
        {
            if(citations[i]>=n-i)
                return n-i;
        }
        return 0;
    }
}
```

```
//Bucket sort

class Solution {
    public int hIndex(int[] citations) {
        int n=citations.length;
        int freq[]=new int[n+1];
        for(int val:citations)
        {
            if(val>=n)
                freq[n]++;
            else
                freq[val]++;
        }
        int count=0;
        for(int i=n;i>=0;i--)
        {
            count+=freq[i];
            if(count>=i)
                return i;
        }
        return 0;
    }
}
```

287. Find the Duplicate Number

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*.

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 = [1,1]
Output: 1
```

Example 4:

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

Constraints:

- `2 <= n <= 3 * 10^4`

- `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.

Follow up:

- How can we prove that at least one duplicate number must exist in `nums` ?
- Can you solve the problem **without** modifying the array `nums` ?
- Can you solve the problem using only constant, `O(1)` extra space?
- Can you solve the problem with runtime complexity less than $O(n^2)$?

Slow fast pointer technique

```
class Solution {
    public int findDuplicate(int[] nums) {
        if(nums.length<=2)
            return nums[0];
        int slow=nums[nums[0]];
        int fast=nums[nums[nums[0]]];
        while(slow!=fast)
        {
            slow=nums[slow];
            fast=nums[nums[fast]];
        }
        slow=nums[0];
        while(fast!=slow)
        {
            slow=nums[slow];
            fast=nums[fast];
        }
        return slow;
    }
}
```

---

295. Find Median from Data Stream

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

- For example, for `arr = [2,3,4]` , the median is `3` .
- For example, for `arr = [2,3]` , the median is `(2 + 3) / 2 = 2.5` .

Implement the MedianFinder class:

- `MedianFinder()` initializes the `MedianFinder` object.
- `void addNum(int num)` adds the integer `num` from the data stream to the data structure.
- `double findMedian()` returns the median of all elements so far. Answers within 10^{-5} of the actual answer will be accepted.

Example 1:

```
Input
["MedianFinder", "addNum", "addNum", "findMedian", "addNum", "findMedian"]
[[], [1], [2], [], [3], []]
Output
[null, null, null, 1.5, null, 2.0]

Explanation
MedianFinder medianFinder = new MedianFinder();
medianFinder.addNum(1);    // arr = [1]
medianFinder.addNum(2);    // arr = [1, 2]
medianFinder.findMedian(); // return 1.5 (i.e., (1 + 2) / 2)
medianFinder.addNum(3);    // arr[1, 2, 3]
medianFinder.findMedian(); // return 2.0
```

Constraints:

- -10^5 <= num <= 10^5
- There will be at least one element in the data structure before calling `findMedian`.
- At most $5 * 10^4$ calls will be made to `addNum` and `findMedian`.

Follow up:

- If all integer numbers from the stream are in the range `[0, 100]`, how would you optimize your solution?
- If `99%` of all integer numbers from the stream are in the range `[0, 100]`, how would you optimize your solution?

```
We will maintain two heap
1. max heap of minimum of n/2 elements
2. min heap of maximum of n/2 elements

max heap of n/2 min elemnts                    min heap of n/2 max elemnts
    1  4  6  7                                        9  10  8  11

Now our task is to check if n is even or odd
suppose size of min heap==size of max heap
then n will be even
then we will ouput median as average of top of both heaps

else
we will output elemnt from max heap

Why we are printing elemnt from max heap??
Beacuse There is possiblity that n will not be even
so while implementing we have to maintain max heap size always grater than min heap
```

```
class MedianFinder {
    PriorityQueue<Integer> max;
    PriorityQueue<Integer> min;
    public MedianFinder() {
        max=new PriorityQueue<>((x,y)->y-x);
        min=new PriorityQueue<>();
    }

    public void addNum(int num) {
        //Add input elemnt to max heap
        max.add(num);
        //Now add greatest of  max heap in min heap to balance heap
        min.add(max.remove());

        //Now we have to maintainn size of max heap grater than min heap
        if(max.size()<min.size())
            max.add(min.remove());
    }

    public double findMedian() {
        if(max.size()==min.size())
            return (max.peek()+min.peek())/2.0;
        else
            return max.peek();
    }
}
```

---

299. Bulls and Cows

You are playing the **Bulls and Cows (https://en.wikipedia.org/wiki/Bulls_and_Cows)** game with your friend.

You write down a secret number and ask your friend to guess what the number is. When your friend makes a guess, you provide a hint with the following info:

- The number of "bulls", which are digits in the guess that are in the correct position.
- The number of "cows", which are digits in the guess that are in your secret number but are located in the wrong position. Specifically, the non-bull digits in the guess that could be rearranged such that they become bulls.

Given the secret number `secret` and your friend's guess `guess`, return *the hint for your friend's guess*.

The hint should be formatted as `"xAyB"`, where `x` is the number of bulls and `y` is the number of cows. Note that both `secret` and `guess` may contain duplicate digits.

Example 1:

```
Input: secret = "1807", guess = "7810"
Output: "1A3B"
Explanation: Bulls are connected with a '|' and cows are underlined:
"1807"
  |
"7810"
```

Example 2:

```
Input: secret = "1123", guess = "0111"
Output: "1A1B"
Explanation: Bulls are connected with a '|' and cows are underlined:
"1123"        "1123"
  |      or     |
"0111"        "0111"
Note that only one of the two unmatched 1s is counted as a cow since the non-bull digit
```

Example 3:

```
Input: secret = "1", guess = "0"
Output: "0A0B"
```

Example 4:

```
Input: secret = "1", guess = "1"
Output: "1A0B"
```

Constraints:

- `1 <= secret.length, guess.length <= 1000`
- `secret.length == guess.length`
- `secret` and `guess` consist of digits only.

```
We can solve using two loop simple implementation
Below is One pass implementation
class Solution {
    public String getHint(String secret, String guess) {
        int freq[]=new int[10];
        int n=guess.length();
        int bulls=0;
        int cows=0;
        for(int i=0;i<n;i++)
        {
            if(secret.charAt(i)==guess.charAt(i))
                bulls++;
            freq[secret.charAt(i)-48]++;
            freq[guess.charAt(i)-48]--;
        }
        int uncommon=0;
        for(int i=0;i<10;i++)
            uncommon+=(Math.abs(freq[i]));
        cows=(2*n-uncommon-2*bulls)/2;
        return new StringBuilder(bulls+"A"+cows+"B").toString();
    }
}
```

---

347. Top K Frequent Elements

Given an integer array `nums` and an integer `k`, return *the* `k` *most frequent elements*. You may return the answer in **any order**.

Example 1:

```
Input: nums = [1,1,1,2,2,3], k = 2
Output: [1,2]
```

Example 2:

```
Input: nums = [1], k = 1
Output: [1]
```

Constraints:

- `1 <= nums.legth <= 10^5`
- `k` is in the range `[1, the number of unique elements in the array]`.
- It is **guaranteed** that the answer is **unique**.

Follow up: Your algorithm's time complexity must be better than `O(n log n)`, where n is the array's size.

```
class Solution {
    public int[] topKFrequent(int[] nums, int k) {
        HashMap<Integer,Integer> h=new HashMap<>();
        for(int val:nums)
            h.put(val,h.getOrDefault(val,0)+1);
        List<Integer>[] bucket=new List[nums.length+1];
        for(int i=0;i<bucket.length;i++)
            bucket[i]=new ArrayList<>();
        for(Integer key:h.keySet())
            bucket[h.get(key)].add(key);
        int ans[]=new int[k];
        int cnt=0;
        for(int i=bucket.length-1;k!=0;i--)
        {
            for(int j=0;j<bucket[i].size();j++)
            {
                ans[cnt]=bucket[i].get(j);
                cnt++;
                k--;
            }
        }
        return ans;
    }
}
```

```
class Solution {
    class pair
    {

        int val;
        int freq;
        pair(int f,int v)
        {
            val=v;
            freq=f;
        }
    }
    public int[] topKFrequent(int[] nums, int k) {
        HashMap<Integer,Integer> h=new HashMap<>();
        for(int val:nums)
            h.put(val,h.getOrDefault(val,0)+1);
        int ans[]=new int[k];
        PriorityQueue<pair> q=new PriorityQueue<>((a,b)->b.freq-a.freq);
        for(Integer key:h.keySet())
            q.offer(new pair(h.get(key),key));
        int i=0;
        while(i!=k)
        {
            ans[i]=q.poll().val;
            i++;
        }
        return ans;
    }
}
```

---

373. Find K Pairs with Smallest Sums

You are given two integer arrays **nums1** and **nums2** sorted in ascending order and an integer **k**.

Define a pair **(u,v)** which consists of one element from the first array and one element from the second array.

Find the k pairs $(u_1,v_1),(u_2,v_2) \ldots(u_k,v_k)$ with the smallest sums.

Example 1:

```
Input: nums1 = [1,7,11], nums2 = [2,4,6], k = 3
Output: [[1,2],[1,4],[1,6]]
Explanation: The first 3 pairs are returned from the sequence:
             [1,2],[1,4],[1,6],[7,2],[7,4],[11,2],[7,6],[11,4],[11,6]
```

Example 2:

```
Input: nums1 = [1,1,2], nums2 = [1,2,3], k = 2
Output: [1,1],[1,1]
Explanation: The first 2 pairs are returned from the sequence:
             [1,1],[1,1],[1,2],[2,1],[1,2],[2,2],[1,3],[1,3],[2,3]
```

Example 3:

```
Input: nums1 = [1,2], nums2 = [3], k = 3
Output: [1,3],[2,3]
Explanation: All possible pairs are returned from the sequence: [1,3],[2,3]
```

Similar problem like merge k sorted array
Basic idea: Use min_heap to keep track on next minimum pair sum, and we only need to maintain K possible candidates in the data structure.
We will create min heap of object or array
1. elemnt from num1
2. elemnt form num2
3. index of elemnt of num2

Now why above structure??
first we will create priority queue of object of num[0] and all elemnt from num2 upto k objects
Now after removing smallest sum from heap form object we will get index of elemnt fro m array num2 and current index
Now we will push next pair of elemnt from num1 and elemnt from nums2 of index current index+1

The run time complexity is O(kLogk) since que.size <= k and we do at most k loop.

```
class Solution {
    public List<List<Integer>> kSmallestPairs(int[] nums1, int[] nums2, int k) {
        List<List<Integer>> ans=new ArrayList<>();
        if(k==0 || nums1.length==0 || nums2.length==0)
            return ans;
        PriorityQueue<int[]> q=new PriorityQueue<>((x,y)->(x[0]+x[1])-(y[0]+y[1]));
        for(int i=0;i<nums1.length && i<k ;i++)
        {
            q.offer(new int[]{nums1[i],nums2[0],1});
        }

        while(!q.isEmpty() && k!=0)
        {
            int curr[]=q.poll();
            ArrayList<Integer> temp=new ArrayList<>();
            temp.add(curr[0]);
            temp.add(curr[1]);
            ans.add(temp);
            if(curr[2]!=nums2.length)
                q.offer(new int[]{curr[0],nums2[curr[2]],curr[2]+1});
            k--;
        }
        return ans;
    }
}
```

```
https://www.interviewbit.com/problems/merge-k-sorted-arrays/
https://practice.geeksforgeeks.org/problems/merge-k-sorted-arrays/1#
public class Solution {
    class bundle
    {
        int index;
        int val;
        int arr_index;
        bundle(int arr_index,int index,int val)
        {
            this.arr_index=arr_index;
            this.index=index;
            this.val=val;
        }
    }

    public int[] solve(int[][] A) {
        int k=A.length;
        int n=A[0].length;
        int ans[]=new int[n*k];
        PriorityQueue<bundle> q=new PriorityQueue<>((x,y)->x.val-y.val);
        for(int i=0;i<k;i++)
            q.offer(new bundle(i,0,A[i][0]));
        int i=0;
        while(!q.isEmpty())
        {
            bundle curr=q.poll();
            ans[i++]=curr.val;
            int pointer=curr.index;
            int arr_pointer=curr.arr_index;
            if(pointer!=n-1)
              q.offer(new bundle(arr_pointer,pointer+1,A[arr_pointer][pointer+1]));
        }
        return ans;
    }
}
```

387. First Unique Character in a String

Given a string, find the first non-repeating character in it and return its index. If it doesn't exist, return -1.

Examples:

```
s = "leetcode"
return 0.

s = "loveleetcode"
return 2.
```

Note: You may assume the string contains only lowercase English letters.

```
class Solution {
    public int firstUniqChar(String s) {
        int store[]=new int[26];
        for(int i=0;i<s.length();i++)
        {
            store[s.charAt(i)-'a']++;
        }
        for(int i=0;i<s.length();i++)
        {
            if(store[s.charAt(i)-'a']==1)
                return i;
        }
        return -1;
    }
}
```

```
Follow up : hard version

https://www.interviewbit.com/problems/first-non-repeating-character-in-a-stream-of-ch
aracters/

public class Solution {
    public String solve(String A) {
        int freq[]=new int[26];
        StringBuilder ans=new StringBuilder();
        int n=A.length();
        int index=0;
        for(int i=0;i<n;i++)
        {
            freq[A.charAt(i)-'a']++;
            while(index<i && freq[A.charAt(index)-'a']>1)
            {
                index++;
            }
            if(freq[A.charAt(index)-'a']==1)
              ans.append(A.charAt(index));
            else
              ans.append('#');
        }
        return ans.toString();
    }
}
```

---

409. Longest Palindrome

Given a string `s` which consists of lowercase or uppercase letters, return *the length of the **longest palindrome*** that can be built with those letters.

Letters are **case sensitive**, for example, `"Aa"` is not considered a palindrome here.

Example 1:

```
Input: s = "abccccdd"
Output: 7
Explanation:
One longest palindrome that can be built is "dccaccd", whose length is 7.
```

Example 2:

```
Input: s = "a"
Output: 1
```

Example 3:

```
Input: s = "bb"
Output: 2
```

Constraints:

- `1 <= s.length <= 2000`
- `s` consists of lowercase **and/or** uppercase English letters only.

```
class Solution {
    public int longestPalindrome(String s) {
        int sum=0;
        int freq[]=new int[59];
        for(int i=0;i<s.length();i++)
        {
            freq[s.charAt(i)-'A']++;
        }
        for(int i=0;i<59;i++)
        {
            if(freq[i]%2!=0)
                sum+=freq[i]-1;
            else
                sum+=freq[i];
        }
        if(sum!=s.length())
            sum++;
        return sum;
    }
}
```

```
//Using Hashset only

class Solution {
    public int longestPalindrome(String s) {
        int sum=0;
        HashSet <Character> freq=new HashSet<Character>();
        for(int i=0;i<s.length();i++)
        {
            char current=s.charAt(i);
            if(freq.contains(current))
            {
                sum+=2;
                freq.remove(current);
            }
            else
               freq.add(current);
        }

        if(sum!=s.length())
            sum++;
        return sum;
    }
}
```

503. Next Greater Element II

Given a circular array (the next element of the last element is the first element of the array), print the Next Greater Number for every element. The Next Greater Number of a number x is the first greater number to its traversing-order next in the array, which means you could search circularly to find its next greater number. If it doesn't exist, output -1 for this number.

Example 1:

```
Input: [1,2,1]
Output: [2,-1,2]
Explanation: The first 1's next greater number is 2;
The number 2 can't find next greater number;
The second 1's next greater number needs to search circularly, which is also 2.
```

Note: The length of given array won't exceed 10000.

```
Explnation:
We need to follow similar approach like to find next greater element in array(Using s
tack)
In above approach we mark -1 for elements which are present in stack after one pass

But in given question Array is cicrcular so we need once again check for elemnts in s
tack
as there are elemnts in front of last elemnts of array
Only difference is we will  not push current elemnt as we have already find gretaer e
lemnt for current in first pass
We will check only for elemnts in stack after first pass

Example:
[1,3,5,2,1]
Now normal gretaer elemnt qyestion output will be:
[3,5,-1,-1,-1]
elements in stack will be [5,2,1]
 But as in given question array is circular
 i.e after last elemnt 1 we will start from first elemnt
 i.e 1,3,5,2,1 1,3,5,2,1 ....
 Now there are elemnts next to last elements
 so answer after first pass same as normal greater elemnt
 [3,5,-1,-1,-1]
elements in stack will be [5,2,1]

Now check for [5,2,1] if gretaer number in array from left to right(as array is circu
lar)
We will not push current elemnt as we have already found greter number for [1,3]
final answer
[3,5,-1,3,3]
```

```
class Solution {
    public int[] nextGreaterElements(int[] nums) {
        int n=nums.length;
        int ans[]=new int[n];
        if(n==0)
            return ans;
        Arrays.fill(ans,-1);
        Stack<Integer> st=new Stack<>();
        st.push(0);
        for(int i=0;i<n;i++)
        {
            while(!st.isEmpty() && nums[st.peek()]<nums[i])
                ans[st.pop()]=nums[i];
            st.push(i);
        }
        for(int i=0;i<n;i++)
        {
            while(!st.isEmpty() && nums[st.peek()]<nums[i])
                ans[st.pop()]=nums[i];
        }
        return ans;
    }
}
```

https://www.interviewbit.com/problems/nearest-smaller-element/
(https://www.interviewbit.com/problems/nearest-smaller-element/)

```
public class Solution {
    public int[] prevSmaller(int[] A) {
        int n=A.length;
        int ans[]=new int[n];
        if(n==0)
          return ans;
        Stack<Integer> st=new Stack<>();
        ans[0]=-1;
        st.push(0);
        for(int i=1;i<n;i++)
        {
            while(!st.isEmpty() && A[st.peek()]>=A[i])
              st.pop();
            ans[i]=(st.isEmpty())?-1:A[st.peek()];
            st.push(i);
        }
        return ans;
    }
}
```

---

556. Next Greater Element III

Given a positive integer `n`, find *the smallest integer which has exactly the same digits existing in the integer* `n` *and is greater in value than* `n`. If no such positive integer exists, return `-1`.

Note that the returned integer should fit in **32-bit integer**, if there is a valid answer but it does not fit in **32-bit integer**, return `-1`.

Example 1:

```
Input: n = 12
Output: 21
```

Example 2:

```
Input: n = 21
Output: -1
```

Constraints:

- `1 <= n <= 2^31 - 1`

```
Similar to find next permutation
Only difeeence that we should return -1 if no greater elemnt present unlike permutati
o
problem no : 31 (next permutation)
```

```
class Solution {
    public int nextGreaterElement(int n) {
        ArrayList<Integer> input=new ArrayList<>();
        while(n!=0)
        {
            input.add(n%10);
            n=n/10;
        }
        Collections.reverse(input);
        int ans=nextPermutation(input.size(),input);
        if(ans<0)
            return -1;
        return ans;
    }

    int nextPermutation(int N, ArrayList<Integer> arr){
        int back=-1;
        for(int i=N-2;i>=0;i--)
        {
            if(arr.get(i)<arr.get(i+1))
            {
                back=i;
                break;
            }
        }
        if(back==-1)
            return -1;
        int swap=N-1;
        for(int i=N-1;i>back;i--)
        {
            if(arr.get(i)>arr.get(back))
            {
                swap=i;
                break;
            }
        }
        int temp=arr.set(swap,arr.get(back));
        arr.set(back,temp);

        for(int i=0;i<(N-back-1)/2;i++)
        {
            int t=arr.set(i+back+1,arr.get(N-1-i));
            arr.set(N-1-i,t);
        }
        int ans=0;
        int mul=1;
        for(int i=arr.size()-1;i>=0;i--)
        {
            if((long)ans+(long)mul*arr.get(i)>Integer.MAX_VALUE)
                return -1;
            ans+=(mul*arr.get(i));
            mul=mul*10;
        }
        return ans;
    }
}
```

581. Shortest Unsorted Continuous Subarray

Given an integer array `nums`, you need to find one **continuous subarray** that if you only sort this subarray in ascending order, then the whole array will be sorted in ascending order.

Return *the shortest such subarray and output its length*.

Example 1:

```
Input: nums = [2,6,4,8,10,9,15]
Output: 5
Explanation: You need to sort [6, 4, 8, 10, 9] in ascending order to make the whole arr
```

Example 2:

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

Example 3:

```
Input: nums = [1]
Output: 0
```

Constraints:

- `1 <= nums.length <= 10^4`
- `-10^5 <= nums[i] <= 10^5`

Follow up: Can you solve it in `O(n)` time complexity?

```
Explanation:
As we required continuos unorted array
We have to validate following condition:
While moving from left to right:
1.  if we find that current number is greater than max elemnt found till curr number
then our property is true till curr element.
2.   else curr index should be in unsorted array
Last element we found violating property will be last element of unsorted array

Similarly from right to left
1.  if we find that current number is smaller than min elemnt found till curr number
then our property is true till curr element.
2.   else curr index should be in unsorted array
Last element we found violating property will be first element of unsorted array

------> should be incresing (current element should be gretaer than max element found
in subarray [0,curr index])
2,6,4,8,10,9,15
<------should be decreasing(current element should be smaller than min element found
in subarray [curr index,r])
```

```
class Solution {
    public int findUnsortedSubarray(int[] nums) {
        int r=nums.length-1;
        int l=0;
        int start=0;
        int end=-1;
        int max=Integer.MIN_VALUE;
        int min=Integer.MAX_VALUE;
        while(r>=0)
        {
            if(nums[l]>=max)
                max=nums[l];
            else
                end=l;
            if(nums[r]<=min)
                min=nums[r];
            else
                start=r;
            r--;
            l++;
        }
        return end-start+1;
    }
}
```

```
Simple to understand not clean code
The idea behind this method is that the correct position of the minimum element in th
e unsorted subarray helps to determine the required left boundary. Similarly, the cor
rect position of the maximum element in the unsorted subarray helps to determine the
required right boundary.
```

public class Solution { public int findUnsortedSubarray(int[] nums) { int min = Integer.MAX_VALUE, max = Integer.MIN_VALUE; for (int i = 1; i < nums.length; i++) { if (nums[i] < nums[i - 1]) min = Math.min(min, nums[i]); } for (int i = nums.length - 2; i >= 0; i--) { if (nums[i] > nums[i + 1]) max = Math.max(max, nums[i]); } int l, r; for (l = 0; l < nums.length; l++) { if (min < nums[l]) break; } for (r = nums.length - 1; r >= 0; r--) { if (max > nums[r]) break; } return r - l < 0 ? 0 : r - l + 1; } }

645. Set Mismatch

You have a set of integers `s`, which originally contains all the numbers from `1` to `n`. Unfortunately, due to some error, one of the numbers in `s` got duplicated to another number in the set, which results in **repetition of one** number and **loss of another** number.

You are given an integer array `nums` representing the data status of this set after the error.

Find the number that occurs twice and the number that is missing and return *them in the form of an array*.

Example 1:

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

Example 2:

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

Constraints:

- 2 <= nums.length <= 10^4
- 1 <= nums[i] <= 10^4

```
/*
Explanation:
Other than following methods:
1.Brute Force (comapare each index and check duplicate) O(n^2)
2.Sorting ( Sort array and check which consecutive index are same) O(nlogn)
3.Using HashMap O(n) / Or frequency array
4.Using XOR ( Same as sum where we want to find 2 odd times repeating number)
   (By XORING all array and natural number, checking set bit from right) O(n)
*/
```

```
/*
Explanation:
Using Following Equations:
x^2 -y^2= (x+y)(x-y)
Consider x as repeatiting number and y as missing number
we know , sum of natural number till n let say val=(n*(n+1)/2)
and sum of square of natural number till n let say sval=(n*(n+1)*(2*n+1)/6)
consider sum as sum of all elements in array
Here sum will be val(sum of natural) +x (Extra repeating number) -y(missing number)
i.e. sum=val+(x-y)
and ssum as square of all elements in array
similarly ssum=sval + x^2 -y^2;
so, ssum=sval+(x+y)(x-y);
     ssum=sval +(x+y)(sum-val)
         in above we already find ssum,sval,sum,val
now we can calculate x+y from above equations
no we have x+y and x-y so we can calculate x and y i.e our answer
converting above logic in program:
*/
```

```
class Solution {
    public int[] findErrorNums(int[] nums) {
        long sum=0,ssum=0;
        long n=nums.length;
        for(int i=0;i<(int)n;i++)
        {
            sum+=nums[i];
            ssum+=(nums[i]*nums[i]);
        }
        long val=sum-(n*(n+1)/2);
        long sval=ssum-(n*(n+1)*(2*n+1)/6);
        long sadd=sval/val;
        int ans[]=new int[2];
        ans[0]=(int)((val+sadd)/2);
        ans[1]=(int)(sadd-ans[0]);
        return ans;
    }
}
```

```
//check else if condition if you not understand

class Solution {
    public int[] findErrorNums(int[] nums) {
        int ans[]=new int[2];
        for(int i=0;i<nums.length;i++)
        {
            int temp=Math.abs(nums[i]);
            ans[1]=ans[1]^(i+1)^(temp);
            if(nums[temp-1]<0)
                ans[0]=temp;
            else if(nums[temp-1]>0)
                nums[temp-1]=-nums[temp-1];
        }
        ans[1]=ans[1]^ans[0];
        return ans;
    }
}
```

```
class Solution {
    public int[] findErrorNums(int[] nums) {
        int xor=0;
        int n=nums.length;
        int ans[]=new int[2];
        for(int i=0;i<n;i++)
            xor=xor^nums[i]^(i+1);
        int rightmostbit=(xor)& ~(xor-1);
        int one=0;
        int two=0;
        for(int i=0;i<n;i++)
        {
            if((rightmostbit &nums[i])!=0)
                one^=nums[i];
            else
                two^=nums[i];
            if(((i+1)&rightmostbit)!=0)
                one^=(i+1);
            else
                two^=(i+1);
        }
        int count=0;
        for(int i=0;i<n;i++)
            if(one==nums[i])
                count++;
        if(count==2)
        {
            ans[0]=one;
            ans[1]=two;
        }
        else
        {
            ans[0]=two;
            ans[1]=one;
        }
        return ans;
    }
}
```

703. Kth Largest Element in a Stream

Design a class to find the k^{th} largest element in a stream. Note that it is the k^{th} largest element in the sorted order, not the k^{th} distinct element.

Implement `KthLargest` class:

- `KthLargest(int k, int[] nums)` Initializes the object with the integer `k` and the stream of integers `nums`.
- `int add(int val)` Returns the element representing the k^{th} largest element in the stream.

Example 1:

```
Input
["KthLargest", "add", "add", "add", "add", "add"]
[[3, [4, 5, 8, 2]], [3], [5], [10], [9], [4]]
Output
[null, 4, 5, 5, 8, 8]

Explanation
KthLargest kthLargest = new KthLargest(3, [4, 5, 8, 2]);
kthLargest.add(3);   // return 4
kthLargest.add(5);   // return 5
kthLargest.add(10);  // return 5
kthLargest.add(9);   // return 8
kthLargest.add(4);   // return 8
```

Constraints:

- `1 <= k <= 10^4`
- `0 <= nums.length <= 10^4`
- `-10^4 <= nums[i] <= 10^4`
- `-10^4 <= val <= 10^4`
- At most `10^4` calls will be made to `add`.
- It is guaranteed that there will be at least `k` elements in the array when you search for the k^{th} element.

```
Explanation:
We will maintain min heap or priority queue
Now we have to find kth largest for every entry
we will maintain heap of size k in min priorityqueue
This will ensure that only k max elemnts in queue
and peek() will be smallest among k i.e. kth largest
every time we will push elemnt in heap or priority

and if size if greater than k then pop (actuallywe have to check if q size great than
k as initial q may be k-1 size)
```

```
class KthLargest {
    PriorityQueue<Integer> q;
    int k;
    public KthLargest(int k, int[] nums) {
        this.k=k;
        q=new PriorityQueue<>();
        for(int num:nums)
            q.offer(num);
        while(q.size()>k)
            q.poll();
    }

    public int add(int val) {
        q.offer(val);
        if(q.size()>k)
          q.poll();
        return q.peek();
    }
}
```

---

823. Binary Trees With Factors

Given an array of unique integers, `arr`, where each integer `arr[i]` is strictly greater than `1`.

We make a binary tree using these integers, and each number may be used for any number of times. Each non-leaf node's value should be equal to the product of the values of its children.

Return *the number of binary trees we can make*. The answer may be too large so return the answer **modulo** `10^9 + 7`.

Example 1:

```
Input: arr = [2,4]
Output: 3
Explanation: We can make these trees: [2], [4], [4, 2, 2]
```

Example 2:

```
Input: arr = [2,4,5,10]
Output: 7
Explanation: We can make these trees: [2], [4], [5], [10], [4, 2, 2], [10, 2, 5], [1
0, 5, 2].
```

Constraints:

- `1 <= arr.length <= 1000`
- `2 <= arr[i] <= 10^9`
- All the values of `arr` are **unique**.

```
/*sort the array and use HashMap to record each Integer, and the number of trees with
that Integer as root
 (1) each integer A[i] will always have one tree with only itself
 (2) if A[i] has divisor (a) in the map, and if A[i]/a also in the map
     then a can be the root of its left subtree, and A[i]/a can be the root of its ri
ght subtree;
     the number of such tree is map.get(a) * map.get(A[i]/a)
(3) sum over the map

Note:
If the root node of the tree (with value v) has children with values x and y (and x *
y == v), then there are map.get(x) * map.get(y) ways to make this tree.
 */
```

```
class Solution {
    public int numFactoredBinaryTrees(int[] arr) {
        HashMap<Integer,Long> h=new HashMap<>();
        int n=arr.length;
        int mod=1000000007;
        int ans=0;
        Arrays.sort(arr);
        for(int i=0;i<n;i++)
        {
            long count=1L;
            for(int j=0;j<i;j++)
            {
                if(arr[i]%arr[j]==0 && h.containsKey(arr[i]/arr[j]))
                    count+=(h.get(arr[i]/arr[j])*h.get(arr[j]));
            }
            ans=(ans%mod+(int)(count%mod))%mod;
            h.put(arr[i],count);
        }
        return ans;
    }
}
```

---

895. Maximum Frequency Stack

Design a stack-like data structure to push elements to the stack and pop the most frequent element from the stack.

Implement the `FreqStack` class:

- `FreqStack()` constructs an empty frequency stack.
- `void push(int val)` pushes an integer `val` onto the top of the stack.
- `int pop()` removes and returns the most frequent element in the stack.
  - If there is a tie for the most frequent element, the element closest to the stack's top is removed and returned.

Example 1:

```
Input
["FreqStack", "push", "push", "push", "push", "push", "push", "pop", "pop", "pop", "pop
[[], [5], [7], [5], [7], [4], [5], [], [], [], []]
Output
[null, null, null, null, null, null, null, 5, 7, 5, 4]

Explanation
FreqStack freqStack = new FreqStack();
freqStack.push(5); // The stack is [5]
freqStack.push(7); // The stack is [5,7]
freqStack.push(5); // The stack is [5,7,5]
freqStack.push(7); // The stack is [5,7,5,7]
freqStack.push(4); // The stack is [5,7,5,7,4]
freqStack.push(5); // The stack is [5,7,5,7,4,5]
freqStack.pop();   // return 5, as 5 is the most frequent. The stack becomes [5,7,5,7,4
freqStack.pop();   // return 7, as 5 and 7 is the most frequent, but 7 is closest to th
freqStack.pop();   // return 5, as 5 is the most frequent. The stack becomes [5,7,4].
freqStack.pop();   // return 4, as 4, 5 and 7 is the most frequent, but 4 is closest to
```

Constraints:

- `0 <= val <= 10^9`
- At most `2 * 10^4` calls will be made to `push` and `pop`.
- It is guaranteed that there will be at least one element in the stack before calling `pop`.

```
General Idea:
At every Instant(call)  we required most frequqnt element
So we have to change frequqncy at every call
So we required HashMap(h) which store frequency of element
Not that we have to update hashmap(h) after every call(push and pop)

Now We have to maintain top frequqncy of elements
We can use TreeMap <frequqncy ,stack>
TreeMap will be frequqncy and stack of  elements of corresponding frequency
Suppose if we have 5 as element and frquency of 2 then it will be present in stack at
key 2 in treemap
Now if new elemnt 5 have to push
then we will increase frequency of 5 in hashmap(h)
and push 5 in stack of tree map at key 3(already freq was 2 + 1)

Now for pop
we have to access max frequqncy and treemap store values according to accesnding orde
r of key
Now last element in treemap will be Highest frequency
So we will pop element from stack at last posnof treemap
Note that we will also remove key at last if stack at that key become empty
```

```
class FreqStack {
    TreeMap<Integer,Stack<Integer>> tm;
    HashMap<Integer,Integer> h;
    public FreqStack() {
        tm=new TreeMap<>();
        h=new HashMap<>();
    }

    public void push(int x) {
        int freq=h.getOrDefault(x,0)+1;
        h.put(x,freq);
        if(!tm.containsKey(freq))
            tm.put(freq,new Stack<>());
        tm.get(freq).push(x);
    }

    public int pop() {
        int last_key=tm.lastKey();
        int val=tm.get(last_key).pop();
        h.put(val,last_key-1);
        if(tm.get(last_key).size()==0)
            tm.remove(last_key);
        return val;
    }
}
```

We can use HashMap and varible maxfrequency instead of treemap for efficient aprroaxh

```
class FreqStack {
    HashMap<Integer,Stack<Integer>> tm;
    HashMap<Integer,Integer> h;
    int max_freq=-1;
    public FreqStack() {
        tm=new HashMap<>();
        h=new HashMap<>();
    }

    public void push(int x) {
        int freq=h.getOrDefault(x,0)+1;
        h.put(x,freq);
        if(!tm.containsKey(freq))
            tm.put(freq,new Stack<>());
        tm.get(freq).push(x);
        if(max_freq<=freq)
            max_freq=freq;
    }

    public int pop() {
        int val=tm.get(max_freq).pop();
        h.put(val,max_freq-1);
        if(tm.get(max_freq).size()==0)
            max_freq--;
        return val;
    }
}
```

921. Minimum Add to Make Parentheses Valid

Given a string `S` of `'('` and `')'` parentheses, we add the minimum number of parentheses (`'('` or `')'`, and in any positions) so that the resulting parentheses string is valid.

Formally, a parentheses string is valid if and only if:

- It is the empty string, or
- It can be written as `AB` (`A` concatenated with `B`), where `A` and `B` are valid strings, or
- It can be written as `(A)`, where `A` is a valid string.

Given a parentheses string, return the minimum number of parentheses we must add to make the resulting string valid.

Example 1:

```
Input: "())"
Output: 1
```

Example 2:

```
Input: "((("
Output: 3
```

Example 3:

```
Input: "()"
Output: 0
```

Example 4:

```
Input: "()))(("
Output: 4
```

Note:

1. `S.length <= 1000`
2. `S` only consists of `'('` and `')'` characters.

```
/*
Explanation:
Algorithm:
  1) We will push if we found open bracket in stack
  2) if we found close bracket
     1) We eill check if it have open bracket in stack or not if yes pop froom stack
     2) else we have to insert open bracket so we have to increment ans
  3) at last if stack is not empty then we required close bracket for open bracket in
stack
     so we add stack of size to ans

Note: Actually we can use count variable to store count of open bracket instead of st
ack
*/
```

```
class Solution {
    public int minAddToMakeValid(String S) {
        int n=S.length();
        int ans=0;
        int count=0;
        for(int i=0;i<n;i++)
        {
            char curr=S.charAt(i);
            if(curr=='(')
                count++;
            if(curr==')')
            {
                if(count==0)
                    ans++;
                else
                    count--;
            }
        }
        return ans+count;
    }
}
```

```
//Stack solution
 public int minAddToMakeValid(String S) {
        Stack<Character> stack = new Stack<Character> ();
        int counter = 0;
        for(int i = 0; i < S.length(); i++) {
            char temp = S.charAt(i);
            if(temp == '(') {
                stack.push(temp);
            }
            else if(!stack.isEmpty()) {
                stack.pop();
            }
            else counter++;
        }
        return counter + stack.size();
    }
```

```
https://practice.geeksforgeeks.org/problems/maximum-index-1587115620/1

/*
Explanation:
Brute force:O(n^2)
Another approach will be Sorting (nlogn)
O(n) approach:
Test Case:
9
34 8 10 3 2 80 30 33 1

//For Intuition see editorial in gfg
We will create min and max array:
min array will be minimum elements at i in arr from o to ith index(current index)
max array will be maximum element at i  in arr from i to n-i-1 index
Basically in General min array will consist minimum till current index
and max array will contain maximum from end till current index
Test Case:
9
34 8 10 3 2 80 30 33 1

min array: 34  8  8  3  2  2  2  2  1
max array: 80 80 80 80 80 80 33 33  1

Now we will use two pointer apparoach and increment i or j according to comparison of
min[i] and max[j] (See code)

*/
```

```
class MaxDifference{

    // Function to find maximum difference of j-1
    // arr[]: input array
    // n: size of array
    static int maxIndexDiff(int arr[], int n) {
        int min[]=new int[n];
        int max[]=new int[n];
        int ans=0;
        int low=Integer.MAX_VALUE;
        int high=Integer.MIN_VALUE;
        for(int i=0;i<n;i++)
        {
            low=Math.min(low,arr[i]);
            min[i]=low;
            high=Math.max(high,arr[n-i-1]);
            max[n-i-1]=high;
        }
        int i=0,j=0;
        while(j<n && i<n)
        {
            if(min[i]>max[j])
               i++;
            else
            {
                ans=Math.max(ans,j-i);
                j++;
            }
        }
        return ans;
    }
}
```

933. Number of Recent Calls

You have a `RecentCounter` class which counts the number of recent requests within a certain time frame.

Implement the `RecentCounter` class:

- `RecentCounter()` Initializes the counter with zero recent requests.
- `int ping(int t)` Adds a new request at time `t`, where `t` represents some time in milliseconds, and returns the number of requests that has happened in the past `3000` milliseconds (including the new request). Specifically, return the number of requests that have happened in the inclusive range `[t - 3000, t]`.

It is **guaranteed** that every call to `ping` uses a strictly larger value of `t` than the previous call.

Example 1:

```
Input
["RecentCounter", "ping", "ping", "ping", "ping"]
[[], [1], [100], [3001], [3002]]
Output
[null, 1, 2, 3, 3]

Explanation
RecentCounter recentCounter = new RecentCounter();
recentCounter.ping(1);     // requests = [1], range is [-2999,1], return 1
recentCounter.ping(100);   // requests = [1, 100], range is [-2900,100], return 2
recentCounter.ping(3001);  // requests = [1, 100, 3001], range is [1,3001], return 3
recentCounter.ping(3002);  // requests = [1, 100, 3001, 3002], range is [2,3002], retur
```

Constraints:

- `1 <= t <= 10^9`
- Each test case will call `ping` with **strictly increasing** values of `t`.
- At most `10^4` calls will be made to `ping`.

```
/*
Explanation: Queue Question
Brute force will be find range[min,max] for every ping()
and check how many numbers will be in given range

But In question mentioned that every next call t value will be greater than previous
As we have incresing input so we will not required numbers which were already not inc
luded in range
Suppose we have maintain range [min,max]
for every t will have min range always greater than previous min
so we will exclude all numbers before min
As we are removing numbers from front we will use queue
*/
```

```
class RecentCounter {
    Queue<Integer> q;
    int min;
    int ans;
    public RecentCounter() {
        q=new LinkedList<>();
        min=-3000;
    }

    public int ping(int t) {
        min=t-3000;
        q.offer(t);
        while(!q.isEmpty() && q.peek()<min)
        {
            q.poll();
        }
        return q.size();
    }
}
```

---

946. Validate Stack Sequences

Given two sequences pushed and popped **with distinct values**, return true if and only if this could have been the result of a sequence of push and pop operations on an initially empty stack.

Example 1:

```
Input: pushed = [1,2,3,4,5], popped = [4,5,3,2,1]
Output: true
Explanation: We might do the following sequence:
push(1), push(2), push(3), push(4), pop() -> 4,
push(5), pop() -> 5, pop() -> 3, pop() -> 2, pop() -> 1
```

Example 2:

```
Input: pushed = [1,2,3,4,5], popped = [4,3,5,1,2]
Output: false
Explanation: 1 cannot be popped before 2.
```

Constraints:

- `0 <= pushed.length == popped.length <= 1000`
- `0 <= pushed[i], popped[i] < 1000`
- pushed is a permutation of popped .
- pushed and popped have distinct values.

```
Explanation:
We can push numbers without restriction as input is given
So we have to validate while popping elements from popped array
There are two cases:
  1) if current element needed to pop from array is yet to be pushed
      i.e. we have to push elements in stack till we found current popped element in
push array
            if we doesnot found element then return false else move to next iteration
2) if(current element is already pushed
    i) then either current element should be at top of stack then we will continue
    ii) else it will follow case 1) and return false.
```

```
class Solution {
    public boolean validateStackSequences(int[] pushed, int[] popped) {
        int n=pushed.length;
        int m=popped.length;
        int i=0;
        int j=0;
        Stack<Integer> st=new Stack<>();
        while(i<m)
        {
            if(!st.isEmpty() && popped[i]==st.peek())
            {
                i++;
                st.pop();
            }
            else
            {
                while(j<n && pushed[j]!=popped[i])
                {
                    st.push(pushed[j]);
                    j++;
                }
                if(j==n)
                    return false;
                i++;
                j++;
            }
        }
        return true;
    }
}
```

---

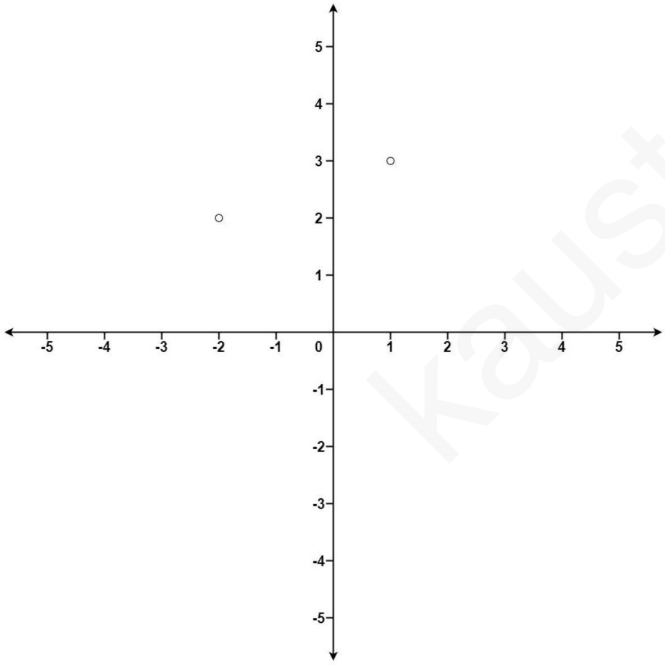
973. K Closest Points to Origin

Given an array of `points` where `points[i] = [xi, yi]` represents a point on the **X-Y** plane and an integer `k`, return the `k` closest points to the origin `(0, 0)`.

The distance between two points on the **X-Y** plane is the Euclidean distance (i.e, $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$).

You may return the answer in **any order**. The answer is **guaranteed** to be **unique** (except for the order that it is in).

Example 1:



Input: points = [[1,3],[-2,2]], k = 1
Output: [[-2,2]]
Explanation:
The distance between (1, 3) and the origin is sqrt(10).
The distance between (-2, 2) and the origin is sqrt(8).
Since sqrt(8) < sqrt(10), (-2, 2) is closer to the origin.
We only want the closest k = 1 points from the origin, so the answer is just [[-2,2]].

Example 2:

Input: points = [[3,3],[5,-1],[-2,4]], k = 2
Output: [[3,3],[-2,4]]
Explanation: The answer [[-2,4],[3,3]] would also be accepted.

Constraints:

- 1 <= k <= points.length <= 10^4
- $-10^4 < x_i, y_i < 10^4$

Efiicient solution:
Using heap : KLog(N)
See Kth top elelmt solution or implementation of heap

```
Using min heap or Sorting(Nlogn)
class Solution {
    public int[][] kClosest(int[][] points, int k) {
        PriorityQueue<int[]> q=new PriorityQueue<>((a,b) ->{
            int sqr1=a[0]*a[0]+a[1]*a[1];
            int sqr2=b[0]*b[0]+b[1]*b[1];
            if(sqr1>sqr2)
                return 1;
            return -1;
        });
        for(int i=0;i<points.length;i++)
            q.offer(points[i]);
        int ans[][]=new int[k][2];
        int i=0;
        while(i<k)
        {
            ans[i]=q.poll();
            i++;
        }
        return ans;
    }
}
```

```
Sorting solution
public int[][] kClosest(int[][] points, int K) {
    Arrays.sort(points, (p1, p2) -> p1[0] * p1[0] + p1[1] * p1[1] - p2[0] * p2[0] - p
2[1] * p2[1]);
    return Arrays.copyOfRange(points, 0, K);
}
```

```
Max heap Solution : NLog(k)
We will just maintain heap of k min elemnts

class Solution {
    public int[][] kClosest(int[][] points, int k) {
        PriorityQueue<int[]> q=new PriorityQueue<>((a,b) ->{
            int sqr1=a[0]*a[0]+a[1]*a[1];
            int sqr2=b[0]*b[0]+b[1]*b[1];
            if(sqr1<sqr2)
                return 1;
            return -1;
        });
        for(int i=0;i<k;i++)
            q.offer(points[i]);
        int ans[][]=new int[k][2];
        for(int i=k;i<points.length;i++)
        {
            q.offer(points[i]);
            q.poll();
        }
        int i=0;
        while(!q.isEmpty())
        {
            ans[i]=q.poll();
            i++;
        }
        return ans;
    }
}
```

1356. Sort Integers by The Number of 1 Bits

Given an integer array `arr` . You have to sort the integers in the array in ascending order by the number of **1's** in their binary representation and in case of two or more integers have the same number of **1's** you have to sort them in ascending order.

Return *the sorted array*.

Example 1:

```
Input: arr = [0,1,2,3,4,5,6,7,8]
Output: [0,1,2,4,8,3,5,6,7]
Explantion: [0] is the only integer with 0 bits.
[1,2,4,8] all have 1 bit.
[3,5,6] have 2 bits.
[7] has 3 bits.
The sorted array by bits is [0,1,2,4,8,3,5,6,7]
```

Example 2:

```
Input: arr = [1024,512,256,128,64,32,16,8,4,2,1]
Output: [1,2,4,8,16,32,64,128,256,512,1024]
Explantion: All integers have 1 bit in the binary representation, you should just sort
```

Example 3:

```
Input: arr = [10000,10000]
Output: [10000,10000]
```

Example 4:

```
Input: arr = [2,3,5,7,11,13,17,19]
Output: [2,3,5,17,7,11,13,19]
```

Example 5:

```
Input: arr = [10,100,1000,10000]
Output: [10,100,10000,1000]
```

Constraints:

- `1 <= arr.length <= 500`
- `0 <= arr[i] <= 10^4`

```
class Solution {
    public int[] sortByBits(int[] arr) {
        int bit[]=new int[10001];
        for(int i=1;i<10001;i++)
        {
            bit[i]=bit[i>>1]+(i&1);
        }
        PriorityQueue<Integer>q=new PriorityQueue<>((a,b) ->bit[a]==bit[b]?a-b:bit[a]
-bit[b]);
        for(int val:arr)
            q.add(val);
        int ans[]=new int[arr.length];
        for(int i=0;i<arr.length;i++)
            ans[i]=q.poll();
        return ans;
    }
}
```

```
//Using Bit manipulation technique
class Solution {
    public int[] sortByBits(int[] arr) {
        int len=arr.length;
        int ans[]=new int[len];
        int bit[]=new int[10001];
        for(int i=1;i<10001;i++)
        {
            bit[i]=bit[i>>1]+(i&1);
        }
        int index=0;
        for(int i=0;i<=32;i++)
        {
            int start=index;
            for(int j=0;j<len;j++)
            {
                if(bit[arr[j]]==i)
                {
                    ans[index]=arr[j];
                    index++;
                }
            }
            Arrays.sort(ans,start,index);
        }
        return ans;
    }
}
```

1460. Make Two Arrays Equal by Reversing Sub-arrays

Given two integer arrays of equal length `target` and `arr`.

In one step, you can select any **non-empty sub-array** of `arr` and reverse it. You are allowed to make any number of steps.

Return *True* if you can make `arr` equal to `target`, or *False* otherwise.

Example 1:

```
Input: target = [1,2,3,4], arr = [2,4,1,3]
Output: true
Explanation: You can follow the next steps to convert arr to target:
1- Reverse sub-array [2,4,1], arr becomes [1,4,2,3]
2- Reverse sub-array [4,2], arr becomes [1,2,4,3]
3- Reverse sub-array [4,3], arr becomes [1,2,3,4]
There are multiple ways to convert arr to target, this is not the only way to do so.
```

Example 2:

```
Input: target = [7], arr = [7]
Output: true
Explanation: arr is equal to target without any reverses.
```

Example 3:

```
Input: target = [1,12], arr = [12,1]
Output: true
```

Example 4:

```
Input: target = [3,7,9], arr = [3,7,11]
Output: false
Explanation: arr doesn't have value 9 and it can never be converted to target.
```

Example 5:

```
Input: target = [1,1,1,1,1], arr = [1,1,1,1,1]
Output: true
```

Constraints:

- `target.length == arr.length`
- `1 <= target.length <= 1000`
- `1 <= target[i] <= 1000`
- `1 <= arr[i] <= 1000`

Obsrvation: We observe that we can convert goven array into target array if element in both array are equal

```
Using Frequency Array
class Solution {
    public boolean canBeEqual(int[] target, int[] arr) {
        int freq[]=new int[1001];
        for(int i=0;i<arr.length;i++)
        {
            freq[arr[i]]++;
            freq[target[i]]--;
        }
        for(int i=0;i<1001;i++)
            if(freq[i]!=0)
                return false;
        return true;
    }
}
```

```
Using sorting
class Solution {
    public boolean canBeEqual(int[] target, int[] arr) {
        Arrays.sort(target);
        Arrays.sort(arr);
        for(int i=0;i<arr.length;i++)
        {
            if(arr[i]!=target[i])
                return false;
        }
        return true;
    }
}
```

1461. Check If a String Contains All Binary Codes of Size K

Given a binary string `s` and an integer `k`.

Return `true` *if every binary code of length* `k` *is a substring of* `s`. Otherwise, return `false`.

Example 1:

```
Input: s = "00110110", k = 2
Output: true
Explanation: The binary codes of length 2 are "00", "01", "10" and "11". They can be al
```

Example 2:

```
Input: s = "00110", k = 2
Output: true
```

Example 3:

```
Input: s = "0110", k = 1
Output: true
Explanation: The binary codes of length 1 are "0" and "1", it is clear that both exist
```

Example 4:

```
Input: s = "0110", k = 2
Output: false
Explanation: The binary code "00" is of length 2 and doesn't exist in the array.
```

Example 5:

```
Input: s = "0000000001011100", k = 4
Output: false
```

Constraints:

- `1 <= s.length <= 5 * 10^5`
- `s[i]` is either `'0'` or `'1'`.
- `1 <= k <= 20`

```
/*
Explanation:
Brute force approach:
we will generate all substring of given length k for binaray string
Total string : 2^k
time complexity wil be 2^k for generating strig
and using kmp algorithm checking each substring present in string or not O(n)
Total Time complexity:  (2^k * n)

Now how to optimize??
One thing we know that we have to found only all substring i.e 2^k present in string
or not
i.e we are concern with number of substrings  not actual substring
Now we will calculate how many distinct substring of given length k can be formed in
string of lengtn n
that can be found in O(n)*k (By traversing string of length n)

We will use hashset for storing distinct substring of length k/
if lenght of hashset is 2^k then return true

*/

//Can be otimized by using bit set and hashset of integer not string
//current complexity: O(n)*O(k)
class Solution {
    public boolean hasAllCodes(String s, int k) {
        int n=s.length();
        int total=(int)Math.pow(2,k);
        if(n<total)
            return false;
        HashSet<String> h=new HashSet<>();
        StringBuilder curr=new StringBuilder();
        for(int i=0;i<k;i++)
        {
            curr.append(s.charAt(i));
        }
        h.add(curr.toString());
        for(int i=k;i<n;i++)
        {
            curr.deleteCharAt(0);
            curr.append(s.charAt(i));
            h.add(curr.toString());
            if(h.size()==total)
                return true;
        }
        if(h.size()==total)
            return true;
        return false;
    }
}
```

1640. Check Array Formation Through Concatenation

You are given an array of **distinct** integers `arr` and an array of integer arrays `pieces`, where the integers in `pieces` are **distinct**. Your goal is to form `arr` by concatenating the arrays in `pieces` **in any order**. However, you are **not** allowed to reorder the integers in each array `pieces[i]`.

Return `true` *if it is possible to form the array* `arr` *from* `pieces`. Otherwise, return `false`.

Example 1:

```
Input: arr = [85], pieces = [[85]]
Output: true
```

Example 2:

```
Input: arr = [15,88], pieces = [[88],[15]]
Output: true
Explanation: Concatenate [15] then [88]
```

Example 3:

```
Input: arr = [49,18,16], pieces = [[16,18,49]]
Output: false
Explanation: Even though the numbers match, we cannot reorder pieces[0].
```

Example 4:

```
Input: arr = [91,4,64,78], pieces = [[78],[4,64],[91]]
Output: true
Explanation: Concatenate [91] then [4,64] then [78]
```

Example 5:

```
Input: arr = [1,3,5,7], pieces = [[2,4,6,8]]
Output: false
```

Constraints:

- `1 <= pieces.length <= arr.length <= 100`
- `sum(pieces[i].length) == arr.length`
- `1 <= pieces[i].length <= arr.length`
- `1 <= arr[i], pieces[i][j] <= 100`
- The integers in `arr` are **distinct**.
- The integers in `pieces` are **distinct** (i.e., If we flatten pieces in a 1D array, all the integers in this array are distinct).

```
We have to check two condition
1) If element in pieces is present in array or not
2) found index of first element of each pieces in an array
    and check whether all elements in pieces comes in array continuos or not
```

```
class Solution {
    public boolean canFormArray(int[] arr, int[][] pieces) {
        HashMap<Integer,Integer> h=new HashMap<>();
        for(int i=0;i<arr.length;i++)
            h.put(arr[i],i);
        for(int i=0;i<pieces.length;i++)
        {
                if(!h.containsKey(pieces[i][0]))
                  return false;
                int index=h.get(pieces[i][0]);
                int j=0;
                for(j=0;j<pieces[i].length && index<arr.length;j++)
                {
                    if(!h.containsKey(pieces[i][j]))
                        return false;
                    if(arr[index++]!=pieces[i][j])
                        return false;
                }
                if(j!=pieces[i].length)
                    return false;
        }
        return true;
    }
}
```

1759. Count Number of Homogenous Substrings

Given a string `s`, return *the number of **homogenous** substrings of* `s`. Since the answer may be too large, return it **modulo** $10^9 + 7$.

A string is **homogenous** if all the characters of the string are the same.

A **substring** is a contiguous sequence of characters within a string.

Example 1:

```
Input: s = "abbcccaa"
Output: 13
Explanation: The homogenous substrings are listed as below:
"a"   appears 3 times.
"aa"  appears 1 time.
"b"   appears 2 times.
"bb"  appears 1 time.
"c"   appears 3 times.
"cc"  appears 2 times.
"ccc" appears 1 time.
3 + 1 + 2 + 1 + 3 + 2 + 1 = 13.
```

Example 2:

```
Input: s = "xy"
Output: 2
Explanation: The homogenous substrings are "x" and "y".
```

Example 3:

```
Input: s = "zzzzz"
Output: 15
```

Constraints:

- `1 <= s.length <= 10^5`
- `s` consists of lowercase letters.

```
class Solution {
    public int countHomogenous(String s) {
        int ans=0;
        long count=1;
        int n=s.length();
        int mod=1000000007;
        for(int i=0;i<n-1;i++)
        {
            int curr=s.charAt(i);
            int prev=s.charAt(i+1);
            if(curr==prev)
            {
                count++;
            }
            else
            {
                ans=(ans%mod+(int)(((count*(count+1))/2)%mod))%mod;
                count=1;
            }
        }
        ans=(ans%mod+(int)(((count*(count+1))/2)%mod))%mod;
        return (int)ans;
    }
}
```

```
//clean code
class Solution {
    public int countHomogenous(String s) {
        int ans=0;
        int count=0;
        int n=s.length();
        int mod=1000000007;
        for(int i=0;i<n;i++)
        {
            if(i>0 && s.charAt(i)==s.charAt(i-1))
            {
                count++;
            }
            else
            {
                count=1;
            }
            ans=(ans%mod+count%mod)%mod;
        }
        return ans;
    }
}
```

```
https://www.interviewbit.com/problems/distinct-numbers-in-window/
Distinct numbers in window


public class Solution {
    public int[] dNums(int[] A, int B) {
        int n=A.length;
        if(B>n)
          return new int[0];
        int ans[]=new int[n-B+1];
        HashMap<Integer,Integer> h=new HashMap<>();
        for(int i=0;i<B;i++)
          h.put(A[i],h.getOrDefault(A[i],0)+1);
        ans[0]=h.size();
        int in=1;
        for(int i=B;i<n;i++)
        {
            h.put(A[i-B],h.get(A[i-B])-1);
            if(h.get(A[i-B])==0)
            {
                h.remove(A[i-B]);
            }
            if(h.containsKey(A[i]))
            {
                h.put(A[i],h.get(A[i])+1);
            }
            else
            {
                h.put(A[i],1);
            }
            ans[in]=h.size();
            in++;
        }
        return ans;
    }
}
```

```
https://practice.geeksforgeeks.org/problems/maximum-distinct-elements-after-removing-k-elements5906/1#

//Similar or relative harder version of distribute candies leetcode 575 question
/*
Explanation:
As we required maximum distinct-elements so first we have to get rid of duplicate elements
Now we will store number and frequency in hashmap
Now there are two cases:
case 1:
  we have enough number of duplicates for given k
  i.e k should be smaller and equal to duplicate elements
  how to find duplicate elements?
  duplicate elements=total elements - distinct-elements
  duplicate elements=n- size of hashmap
Case 2:
  K is bigger than all duplicate elements
  now we anyhow make k completely exhausted i.e. k==0
  hence we have to delete distinct-elements also
  simple we will delete total elements-K
*/

class Complete{
    int maxDistinctNum(int a[], int n, int K)
    {
        HashMap<Integer,Integer> h=new HashMap<>();
        for(int num:a)
          h.put(num,h.getOrDefault(num,0)+1);
        int hsize=h.size();
        if(K<=(n-hsize))
          return hsize;
        return n-K;
    }


}
```

1773. Count Items Matching a Rule

You are given an array `items`, where each `items[i] = [type_i, color_i, name_i]` describes the type, color, and name of the `i^th` item. You are also given a rule represented by two strings, `ruleKey` and `ruleValue`.

The `i^th` item is said to match the rule if **one** of the following is true:

- `ruleKey == "type"` and `ruleValue == type_i`.
- `ruleKey == "color"` and `ruleValue == color_i`.
- `ruleKey == "name"` and `ruleValue == name_i`.

Return *the number of items that match the given rule*.

Example 1:

```
Input: items = [["phone","blue","pixel"],["computer","silver","lenovo"],["phone","gold"
Output: 1
Explanation: There is only one item matching the given rule, which is ["computer","silv
```

Example 2:

```
Input: items = [["phone","blue","pixel"],["computer","silver","phone"],["phone","gold",
Output: 2
Explanation: There are only two items matching the given rule, which are ["phone","blue
```

Constraints:

- `1 <= items.length <= 10^4`
- `1 <= type_i.length, color_i.length, name_i.length, ruleValue.length <= 10`
- `ruleKey` is equal to either `"type"`, `"color"`, or `"name"`.
- All strings consist only of lowercase letters.

```
class Solution {
    public int countMatches(List<List<String>> items, String ruleKey, String ruleValu
e) {
        HashMap<String,Integer> h=new HashMap<>();
        h.put("type",0);
        h.put("color",1);
        h.put("name",2);
        int index=h.get(ruleKey);
        int ans=0;
        for(int i=0;i<items.size();i++)
        {
            if((items.get(i).get(index)).equals(ruleValue))
                ans++;
        }
        return ans;
    }
}
```

1792. Maximum Average Pass Ratio

There is a school that has classes of students and each class will be having a final exam. You are given a 2D integer array `classes`, where `classes[i] = [pass_i, total_i]`. You know beforehand that in the `i^th` class, there are `total_i` total students, but only `pass_i` number of students will pass the exam.

You are also given an integer `extraStudents`. There are another `extraStudents` brilliant students that are **guaranteed** to pass the exam of any class they are assigned to. You want to assign each of the `extraStudents` students to a class in a way that **maximizes** the **average** pass ratio across **all** the classes.

The **pass ratio** of a class is equal to the number of students of the class that will pass the exam divided by the total number of students of the class. The **average pass ratio** is the sum of pass ratios of all the classes divided by the number of the classes.

Return *the **maximum** possible average pass ratio after assigning the* `extraStudents` *students.* Answers within `10^-5` of the actual answer will be accepted.

Example 1:

```
Input: classes = [[1,2],[3,5],[2,2]], extraStudents = 2
Output: 0.78333
Explanation: You can assign the two extra students to the first class. The average pass
```

Example 2:

```
Input: classes = [[2,4],[3,9],[4,5],[2,10]], extraStudents = 4
Output: 0.53485
```

Constraints:

- $1 <= classes.length <= 10^5$
- $classes[i].length == 2$
- $1 <= pass_i <= total_i <= 10^5$
- $1 <= extraStudents <= 10^5$

Explanation: Pay attention to how much the pass ratio changes when you add a student to the class. If you keep adding students,The more students you add to a class, the smaller the change in pass ratio becomes. Since the change in the pass ratio is always decreasing with the more students you add, then the very first student you add to each class is the one that makes the biggest change in the pass ratio. Because each class's pass ratio is weighted equally, it's always optimal to put the student in the class that makes the biggest change among all the other classes. Keep a max heap of the current class sizes and order them by the change in pass ratio. For each extra student, take the top of the heap, update the class size, and put it back in the heap.

```
class Solution {
    public double maxAverageRatio(int[][] classes, int extraStudents) {
        PriorityQueue<double[]> q=new PriorityQueue<>((x,y)-> Double.compare(y[0],x
[0]));
        int n=classes.length;
        double ans=0;
        for(int i=0;i<n;i++)
        {
            double num=classes[i][0];
            double deno=classes[i][1];
            double profit=((num+1)/(deno+1))-(num/deno);
            q.offer(new double[]{profit,num+1,deno+1});
            ans+=(num/deno);
        }

        while(extraStudents-- >0)
        {
            double curr[]=q.poll();
            ans+=curr[0];
            double numerator=curr[1];
            double denominator=curr[2];
            double future_profit=((numerator+1)/(denominator+1))-(numerator/denominat
or);
            q.offer(new double[]{future_profit,numerator+1,denominator+1});
        }
        return ans/n;
    }
}
```