1. Two Sum

Given an array of integers `nums` and an integer `target`, return *indices of the two numbers such that they add up to `target`*.

You may assume that each input would have ***exactly*** **one solution**, and you may not use the *same* element twice.

You can return the answer in any order.

Example 1:

```
Input: nums = [2,7,11,15], target = 9
Output: [0,1]
Explanation: Because nums[0] + nums[1] == 9, we return [0, 1].
```

Example 2:

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

Example 3:

```
Input: nums = [3,3], target = 6
Output: [0,1]
```

Constraints:

- $2 <= \texttt{nums.length} <= 10^4$
- $-10^9 <= \texttt{nums[i]} <= 10^9$
- $-10^9 <= \texttt{target} <= 10^9$
- **Only one valid answer exists.**

Follow-up: Can you come up with an algorithm that is less than $O(n^2)$ time complexity?

```
class Solution {
    public int[] twoSum(int[] nums, int target) {
        HashMap<Integer,Integer> hm=new HashMap<>();
        int [] arr=new int[2];
        for(int i=0;i<nums.length;i++)
        {
            if(hm.containsKey(target-nums[i]))
            {
                arr[0]=hm.get(target-nums[i]);
                arr[1]=i;
            }
            else
            {
                hm.put(nums[i],i);
            }
        }
        return arr;
    }
}
```

11. Container With Most Water

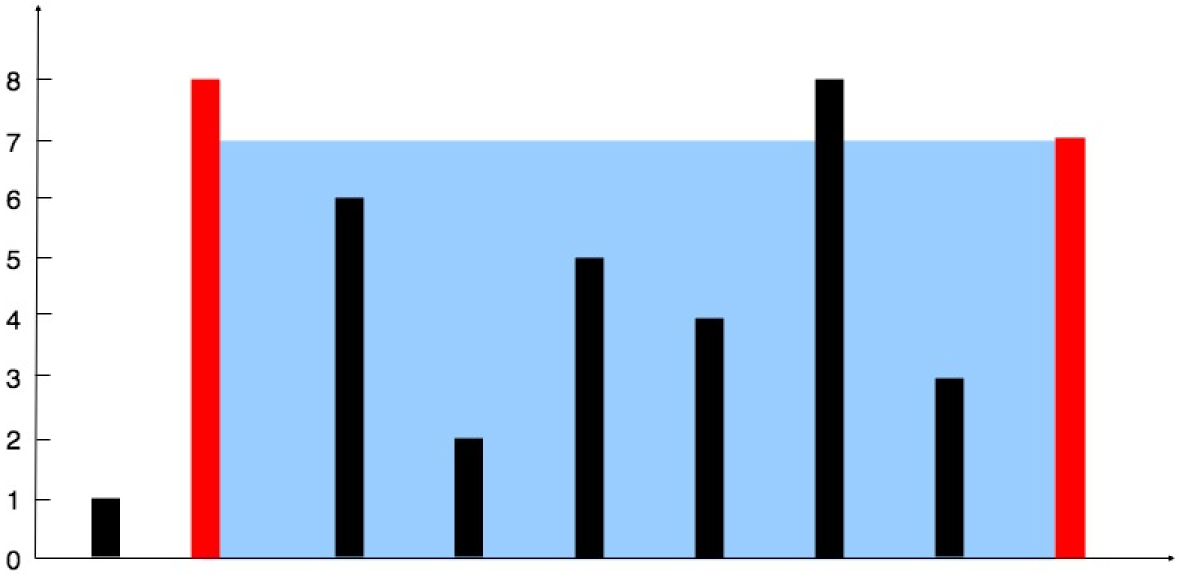
You are given an integer array `height` of length `n`. There are `n` vertical lines drawn such that the two endpoints of the `i`$^{\text{th}}$ line are `(i, 0)` and `(i, height[i])`.

Find two lines that together with the x-axis form a container, such that the container contains the most water.

Return *the maximum amount of water a container can store*.

Notice that you may not slant the container.

Example 1:



```
Input: height = [1,8,6,2,5,4,8,3,7]
Output: 49
Explanation: The above vertical lines are represented by array [1,8,6,2,5,4,8
```

Example 2:

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

Constraints:

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

---

1. O(N)

```
class Solution {
 public int maxArea(int[] height) {
     int right=height.length-1;
     int left=0;
     int maxVolume=Integer.MIN_VALUE;
     while(left<right)
     {
         int width=right-left;
         int heightAtIndex=(height[left]<height[right])?height[left]:height[right];
         if((width*heightAtIndex)>maxVolume)
         {
             maxVolume=width*heightAtIndex;
         }
         else if(height[left]<height[right])
         {
             left++;
         }
         else
         {
             right--;
         }
     }

     return maxVolume;
 }
}
```

2. O(N^2)

```
class Solution {
 public int maxArea(int[] height) {
     int n=height.length;

     int maxVolume=Integer.MIN_VALUE;
     for(int i=0;i<n;i++)
     {
         for(int j=i+1;j<n;j++)
         {
         int minHeight=(height[i]<height[j])?height[i]:height[j];
         int volume=minHeight*(j-i);
         if(volume>maxVolume)
         maxVolume=volume;

         }
     }
     return maxVolume;
 }
}
```

15. 3Sum

Given an integer array nums, return all the triplets [nums[i], nums[j], nums[k]] such that i != j , i != k , and j != k , and nums[i] + nums[j] + nums[k] == 0 .

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]]
Explanation:
nums[0] + nums[1] + nums[2] = (-1) + 0 + 1 = 0.
nums[1] + nums[2] + nums[4] = 0 + 1 + (-1) = 0.
nums[0] + nums[3] + nums[4] = (-1) + 2 + (-1) = 0.
The distinct triplets are [-1,0,1] and [-1,-1,2].
Notice that the order of the output and the order of the triplets does not ma
```

Example 2:

```
Input: nums = [0,1,1]
Output: []
Explanation: The only possible triplet does not sum up to 0.
```

Example 3:

```
Input: nums = [0,0,0]
Output: [[0,0,0]]
Explanation: The only possible triplet sums up to 0.
```

Constraints:

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

1. time: O(N^2), space: O(N)

```
public List<List<Integer>> threeSum(int[] nums) {
 Arrays.sort(nums);
 List<List<Integer>> list = new ArrayList<List<Integer>>();
 for(int i = 0; i < nums.length-2; i++) {
     if(i > 0 && (nums[i] == nums[i-1])) continue; // avoid duplicates
     for(int j = i+1, k = nums.length-1; j<k;) {
         if(nums[i] + nums[j] + nums[k] == 0) {
             list.add(Arrays.asList(nums[i],nums[j],nums[k]));
             j++;k--;
             while((j < k) && (nums[j] == nums[j-1]))j++;// avoid duplicates
             while((j < k) && (nums[k] == nums[k+1]))k--;// avoid duplicates
         }else if(nums[i] + nums[j] + nums[k] > 0) k--;
         else j++;
     }
 }
 return list;
}
```

2. time: O(N^3) , space:O(N)

```
class Solution {
 public List<List<Integer>> threeSum(int[] nums) {
     Arrays.sort(nums);
     ArrayList<List<Integer>> al=new ArrayList<>();
     HashSet<List<Integer>> hs=new HashSet<>();
     for(int i=0;i<nums.length;i++)
     {
         for(int j=i+1;j<nums.length;j++)
         {
             for(int k=j+1;k<nums.length;k++)
             {
             if(nums[i]+nums[j]+nums[k]==0)
             {
                 hs.add(new ArrayList<>(List.of(nums[i],nums[j],nums[k])));
             }
             }
         }
     }
     hs.stream().forEach(i->al.add(i));
     return al;
 }
}
```

20. Valid Parentheses

Given a string `s` containing just the characters `'('`, `')'`, `'{'`, `'}'`, `'['` and `']'`, determine if the input string is valid.

An input string is valid if:

1. Open brackets must be closed by the same type of brackets.
2. Open brackets must be closed in the correct order.
3. Every close bracket has a corresponding open bracket of the same type.

Example 1:

```
Input: s = "()"
Output: true
```

Example 2:

```
Input: s = "()[]{}"
Output: true
```

Example 3:

```
Input: s = "(]"
Output: false
```

Constraints:

- `1 <= s.length <=` 10^4
- `s` consists of parentheses only `'()[]{}'`.

```
class Solution {
    public boolean isValid(String s) {
    Stack<Character> stack = new Stack<Character>();
    for (char c : s.toCharArray()) {
        if (c == '(')
            stack.push(')');
        else if (c == '{')
            stack.push('}');
        else if (c == '[')
            stack.push(']');
        else if (stack.isEmpty() || stack.pop() != c)
            return false;
    }
    return stack.isEmpty();
}
}
```

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 lowercase English letters.

1.

```
public List<List<String>> groupAnagrams(String[] strs) {
    if (strs == null || strs.length == 0) return new ArrayList<>();
    Map<String, List<String>> map = new HashMap<>();
    for (String s : strs) {
        char[] ca = s.toCharArray();
        Arrays.sort(ca);
        String keyStr = String.valueOf(ca);
        if (!map.containsKey(keyStr)) map.put(keyStr, new ArrayList<>());
        map.get(keyStr).add(s);
    }
    return new ArrayList<>(map.values());
}
```

2.

```
import java.util.*;
class Solution {
    public List<List<String>> groupAnagrams(String[] strs) {
       HashMap<String,ArrayList<String>> hm=new HashMap<>();
       for(int i=0;i<strs.length;i++)
       {
        //each string in array
        //key is sorted string
        String sortedStringChar=sortString(strs[i]);
        if(!hm.containsKey(sortedStringChar))
        {
            ArrayList<String> alString=new ArrayList<>();
            alString.add(strs[i]);
            hm.put(sortedStringChar,alString);
        }
        else
        {
            ArrayList<String> al=hm.get(sortedStringChar);
            al.add(strs[i]);
            hm.put(sortedStringChar,al);
        }
       }
       System.out.println(hm);
       Collection<ArrayList<String>> valuesCollection =hm.values();
       ArrayList<List<String>> result=new ArrayList<List<String>>();
       valuesCollection.stream().forEach(i->{result.add(i);});
       return result;
    }
    public static String sortString(String inputString)
    {
        // Converting input string to character array
        char tempArray[] = inputString.toCharArray();

        // Sorting temp array using
        Arrays.sort(tempArray);

        // Returning new sorted string
        return new String(tempArray);
    }
}
```

125. Valid Palindrome

A phrase is a **palindrome** if, after converting all uppercase letters into lowercase letters and removing all non-alphanumeric characters, it reads the same forward and backward. Alphanumeric characters include letters and numbers.

Given a string `s`, return `true` *if it is a* ***palindrome****, or* `false` *otherwise*.

Example 1:

```
Input: s = "A man, a plan, a canal: Panama"
Output: true
Explanation: "amanaplanacanalpanama" is a palindrome.
```

Example 2:

```
Input: s = "race a car"
Output: false
Explanation: "raceacar" is not a palindrome.
```

Example 3:

```
Input: s = " "
Output: true
Explanation: s is an empty string "" after removing non-alphanumeric characte
Since an empty string reads the same forward and backward, it is a palindrome
```

Constraints:

- `1 <= s.length <= 2 * ` 10^5
- `s` consists only of printable ASCII characters.

1.Array generic operation

```
class Solution {
    public boolean isPalindrome(String s) {
        s=s.toLowerCase().trim();
        s=s.replaceAll(" ","");
        char[] ch=s.toCharArray();
        ArrayList<Character>arrayList=new ArrayList<>();
        for(char i:ch)
        {
            if((i>='a'&& i<='z')||(i>='0'&&i<='9'))
                arrayList.add(i);
        }
         for(int i=0;i<(arrayList.size()-1);i++)
        {
            if(arrayList.get(i)!=arrayList.get(arrayList.size() - 1 - i))
                return false;
        }
        return true;
    }
}
```

2. 2 pointer Note: Character.isLetterOrDigit(cHead) -> allows alphanumeric

```
public class Solution {
 public boolean isPalindrome(String s) {
     if (s.isEmpty()) {
         return true;
     }
     int head = 0, tail = s.length() - 1;
     char cHead, cTail;
     while(head <= tail) {
         cHead = s.charAt(head);
         cTail = s.charAt(tail);
         if (!Character.isLetterOrDigit(cHead)) {
             head++;
         } else if(!Character.isLetterOrDigit(cTail)) {
             tail--;
         } else {
             if (Character.toLowerCase(cHead) != Character.toLowerCase
(cTail)) {
                 return false;
             }
             head++;
             tail--;
         }
     }

     return true;
 }
}
```

128. Longest Consecutive Sequence

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

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

Example 1:

```
Input: nums = [100,4,200,1,3,2]
Output: 4
Explanation: The longest consecutive elements sequence is [1, 2, 3,
4]. Therefore its length is 4.
```

Example 2:

```
Input: nums = [0,3,7,2,5,8,4,6,0,1]
Output: 9
```

Constraints:

- `0 <= nums.length <=` 10^5
- -10^9 `<= nums[i] <=` 10^9

```
class Solution {
    public int longestConsecutive(int[] nums) {
        // finding the number of digits that are continous
        Arrays.sort(nums);
        //get the result as maximum continous numbers
        int result=1;
        //continous number count for each count
        int conti=1;
        for(int i=0;i<nums.length-1;i++)
        {
            //continous number
            if(nums[i+1]==nums[i]+1)
            {
                conti++;
                result=conti>result?conti:result;
            }
            //same number
            else if(nums[i+1]==nums[i])
            {
                continue;
            }
            //reset for not continous
            else
            {
                conti=1;
            }
        }
        //if length if nums is zero then no continous number
        if(nums.length==0)
            return 0;
        else
        return result;

    }
}
```

150. Evaluate Reverse Polish Notation

You are given an array of strings `tokens` that represents an arithmetic expression in a Reverse Polish Notation (http://en.wikipedia.org/wiki/Reverse_Polish_notation).

Evaluate the expression. Return *an integer that represents the value of the expression*.

Note that:

- The valid operators are `'+'`, `'-'`, `'*'`, and `'/'`.
- Each operand may be an integer or another expression.
- The division between two integers always **truncates toward zero**.

- There will not be any division by zero.
- The input represents a valid arithmetic expression in a reverse polish notation.
- The answer and all the intermediate calculations can be represented in a **32-bit** integer.

Example 1:

```
Input: tokens = ["2","1","+","3","*"]
Output: 9
Explanation: ((2 + 1) * 3) = 9
```

Example 2:

```
Input: tokens = ["4","13","5","/","+"]
Output: 6
Explanation: (4 + (13 / 5)) = 6
```

Example 3:

```
Input: tokens = ["10","6","9","3","+","-11","*","/","*","17","+","5","+"]
Output: 22
Explanation: ((10 * (6 / ((9 + 3) * -11))) + 17) + 5
= ((10 * (6 / (12 * -11))) + 17) + 5
= ((10 * (6 / -132)) + 17) + 5
= ((10 * 0) + 17) + 5
= (0 + 17) + 5
= 17 + 5
= 22
```

Constraints:

- 1 <= tokens.length <= 10^4
- tokens[i] is either an operator: "+", "-", "*", or "/", or an integer in the range [-200, 200].

```
public class Solution {
    public int evalRPN(String[] tokens) {
        int a,b;
        Stack<Integer> S = new Stack<Integer>();
        for (String s : tokens) {
            if(s.equals("+")) {
                S.add(S.pop()+S.pop());
            }
            else if(s.equals("/")) {
                b = S.pop();
                a = S.pop();
                S.add(a / b);
            }
            else if(s.equals("*")) {
                S.add(S.pop() * S.pop());
            }
            else if(s.equals("-")) {
                b = S.pop();
                a = S.pop();
                S.add(a - b);
            }
            else {
                S.add(Integer.parseInt(s));
            }
        }
        return S.pop();
    }
}
```

Note:

to convert string to integer

```
    Integer.parseInt(s)
```

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(int 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.

You must implement a solution with `O(1)` time complexity for each function.

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` .

```
class MinStack {
        Stack<Integer> st;
        PriorityQueue<Integer> pq;
    public MinStack() {
        st=new Stack<>();
        pq=new PriorityQueue<>();
    }

    public void push(int val) {
        st.push(val);
        pq.add(val);
    }

    public void pop() {
        pq.remove(st.pop());

    }

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

    public int getMin() {
        return pq.peek();
    }
}

/**
 * Your MinStack object will be instantiated and called as such:
 * MinStack obj = new MinStack();
 * obj.push(val);
 * obj.pop();
 * int param_3 = obj.top();
 * int param_4 = obj.getMin();
 */
```

167. Two Sum II - Input Array Is Sorted

Given a **1-indexed** array of integers `numbers` that is already ***sorted in non-decreasing order***, find two numbers such that they add up to a specific `target` number. Let these two numbers be `numbers[index`$_1$`]` and `numbers[index`$_2$`]` where `1 <= index`$_1$ `< index`$_2$ `<= numbers.length`.

Return *the indices of the two numbers,* `index`$_1$ *and* `index`$_2$ *,* ***added by one*** *as an integer array* `[index`$_1$`, index`$_2$`]` *of length 2.*

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

Your solution must use only constant extra space.

Example 1:

Input: `numbers = [2,7,11,15], target = 9`
Output: `[1,2]`
Explanation: The sum of 2 and 7 is 9. Therefore, $index_1$ = 1, $index_2$ = 2. We re

Example 2:

Input: `numbers = [2,3,4], target = 6`
Output: `[1,3]`
Explanation: The sum of 2 and 4 is 6. Therefore $index_1$ = 1, $index_2$ = 3. We ret

Example 3:

Input: `numbers = [-1,0], target = -1`
Output: `[1,2]`
Explanation: The sum of -1 and 0 is -1. Therefore $index_1$ = 1, $index_2$ = 2. We

Constraints:

- `2 <= numbers.length <=` $3 * 10^4$
- `-1000 <= numbers[i] <= 1000`
- `numbers` is sorted in **non-decreasing order**.
- `-1000 <= target <= 1000`
- The tests are generated such that there is **exactly one solution**.

1.

```
class Solution {
   public int[] twoSum(int[] numbers, int target) {
    int l = 0, r = numbers.length - 1;
    while (numbers[l] + numbers[r] != target) {
        if (numbers[l] + numbers[r] > target) r--;
        else l++;
    }
    return new int[]{l + 1, r + 1};
}
}
```

2.

```
class Solution {
    public int[] twoSum(int[] numbers, int target) {
        int [] result=new int[2];
        int j=numbers.length-1;
        int i=0;
        while(i<j)
        {
            if(numbers[i]+numbers[j]==target)
            {
                result[0]=i+1;
                result[1]=j+1;
               break;
            }
            else if(numbers[i]+numbers[j]>target)
            {
                j--;
            }
            else
            {
                i++;
            }
        }
        return result;
    }
}
```

217. Contains Duplicate

Given an integer array `nums`, return `true` if any value appears **at least twice** in the array, and return `false` if every element is distinct.

Example 1:

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

Example 2:

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

Example 3:

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

Constraints:

- 1 <= nums.length <= 10^5
- -10^9 <= nums[i] <= 10^9

```
class Solution {
    public boolean containsDuplicate(int[] nums) {
    HashSet<Integer> flag = new HashSet<Integer>();

    for(int i : nums) {
        if(!flag.add(i)) {
            return true;
        }
    }
    return false;
    }
}
```

238. Product of Array Except Self

Given an integer array `nums` , return *an array* `answer` *such that* `answer[i]` *is equal to the product of all the elements of* `nums` *except* `nums[i]` .

The product of any prefix or suffix of `nums` is **guaranteed** to fit in a **32-bit** integer.

You must write an algorithm that runs in `O(n)` time and without using the division operation.

Example 1:

```
Input: nums = [1,2,3,4]
Output: [24,12,8,6]
```

Example 2:

```
Input: nums = [-1,1,0,-3,3]
Output: [0,0,9,0,0]
```

Constraints:

- 2 <= nums.length <= 10^5
- -30 <= nums[i] <= 30

- The product of any prefix or suffix of `nums` is **guaranteed** to fit in a **32-bit** integer.

Follow up: Can you solve the problem in `O(1)` extra space complexity? (The output array **does not** count as extra space for space complexity analysis.)

```
class Solution {
    public int[] productExceptSelf(int[] nums) {
         int n = nums.length;
        int[] res = new int[n];
        // Calculate lefts and store in res.
        int left = 1;
        for (int i = 0; i < n; i++) {
            if (i > 0)
                left = left * nums[i - 1];
            res[i] = left;
        }
        // Calculate rights and the product from the end of the array.
        int right = 1;
        for (int i = n - 1; i >= 0; i--) {
            if (i < n - 1)
                right = right * nums[i + 1];
            res[i] *= right;
        }
        return res;

    }
}
```

242. Valid Anagram

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

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

Example 1:

```
Input: s = "anagram", t = "nagaram"
Output: true
```

Example 2:

```
Input: s = "rat", t = "car"
Output: false
```

Constraints:

- 1 <= s.length, t.length <= 5 * 10^4
- s and t consist of lowercase English letters.

Follow up: What if the inputs contain Unicode characters? How would you adapt your solution to such a case?

```
class Solution {
    public boolean isAnagram(String s, String t) {
        int arr[]=new int[30];
        for(int i=0;i<s.length();i++)
        {
            int ch=(int)(s.charAt(i)-'a');
            arr[ch]++;
        }
        for(int i=0;i<t.length();i++)
        {
            int ch=(int)(t.charAt(i)-'a');
            arr[ch]--;
        }
        // int t=0;
        for(int i=0;i<30;i++)
        {
            if(arr[i]!=0)
            return false;
        }
        return true;

    }
}
```

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.length <= 10^5
- -10^4 <= nums[i] <= 10^4
- `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> map = new HashMap<>();
        for (int n : nums) {
            map.put(n, map.getOrDefault(n,0) + 1);
        }

        PriorityQueue<int[]> pq = new PriorityQueue<>((a,b) -> Integer.comp
are(a[1], b[1]));
        for (Map.Entry<Integer, Integer> e : map.entrySet()) {
            pq.add(new int[]{e.getKey(), e.getValue()});
            while (pq.size() > k) {
                pq.poll();
            }
        }

        int[] result = new int[k];
        for (int i = 0; i < k; i++) {
            result[i] = pq.poll()[0];
        }

        return result;
    }
}
```

739. Daily Temperatures

Given an array of integers `temperatures` represents the daily temperatures, return *an array* `answer` *such that* `answer[i]` *is the number of days you have to wait after the* i^{th} *day to get a warmer temperature*. If there is no future day for which this is possible, keep `answer[i] == 0` instead.

Example 1:

```
Input: temperatures = [73,74,75,71,69,72,76,73]
Output: [1,1,4,2,1,1,0,0]
```

Example 2:

```
Input: temperatures = [30,40,50,60]
Output: [1,1,1,0]
```

Example 3:

```
Input: temperatures = [30,60,90]
Output: [1,1,0]
```

Constraints:

- `1 <= temperatures.length <=` 10^5
- `30 <= temperatures[i] <= 100`

---

1. Stacks

```
public int[] dailyTemperatures(int[] temperatures) {
 Stack<Integer> stack = new Stack<>();
 int[] ret = new int[temperatures.length];
 for(int i = 0; i < temperatures.length; i++) {
     while(!stack.isEmpty() && temperatures[i] > temperatures[stack.peek()]) {
         int idx = stack.pop();
         ret[idx] = i - idx;
     }
     stack.push(i);
 }
 return ret;
}
```

2. Array

```
public int[] dailyTemperatures(int[] temperatures) {
 int[] stack = new int[temperatures.length];
 int top = -1;
 int[] ret = new int[temperatures.length];
 for(int i = 0; i < temperatures.length; i++) {
     while(top > -1 && temperatures[i] > temperatures[stack[top]]) {
         int idx = stack[top--];
         ret[idx] = i - idx;
     }
     stack[++top] = i;
 }
 return ret;
}
```