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 <= nums.length <= 10^4`
- `-10^9 <= nums[i] <= 10^9`
- `-10^9 <= 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;
    }
}
```

---

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);
    }
}
```

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();
 */
```

---

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;
    }
}
```