

# 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:

- $2 \leq \text{nums.length} \leq 10^4$
- $-10^9 \leq \text{nums}[i] \leq 10^9$
- $-10^9 \leq \text{target} \leq 10^9$
- **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;
    }
}

```

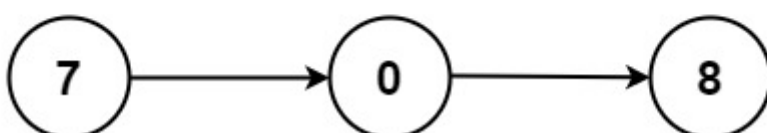
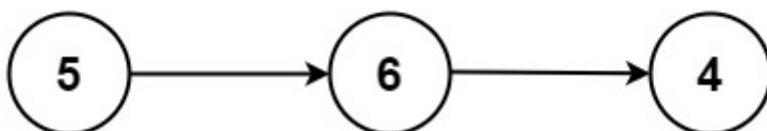
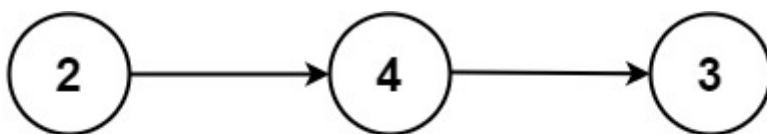
## 2. Add Two Numbers [↗](#)



You are given two **non-empty** linked lists representing two non-negative integers. The digits are stored in **reverse order**, and each of their nodes contains a single digit. Add the two numbers and return the sum as a linked list.

You may assume the two numbers do not contain any leading zero, except the number 0 itself.

**Example 1:**



**Input:** l1 = [2,4,3], l2 = [5,6,4]

**Output:** [7,0,8]

**Explanation:** 342 + 465 = 807.

**Example 2:**

**Input:** l1 = [0], l2 = [0]

**Output:** [0]

**Example 3:**

**Input:** l1 = [9,9,9,9,9,9,9], l2 = [9,9,9,9]

**Output:** [8,9,9,9,0,0,0,1]

**Constraints:**

- The number of nodes in each linked list is in the range [1, 100] .
  - $0 \leq \text{Node.val} \leq 9$
  - It is guaranteed that the list represents a number that does not have leading zeros.
-

```
/*
    Name - Add Two Numbers
    Link - https://leetcode.com/problems/add-two-numbers/
    Time Complexity -  $O(m + n)$ 
    Space Complexity -  $O(1)$ 
    Note - make use of modulo (get remainder) and division (get quotient)
*/
```

```
class Solution {

    public ListNode addTwoNumbers(ListNode first, ListNode second) {
        int q = 0;
        int r = 0;
        int sum = 0;
        ListNode head = null;
        ListNode temp = null;
        while (first != null || second != null) {
            sum =
                q +
                (
                    ((first != null) ? first.val : 0) +
                    ((second != null) ? second.val : 0)
                );
            r = sum % 10;
            q = sum / 10;
            ListNode newNode = new ListNode(r);
            if (head == null) {
                head = newNode;
            } else {
                temp = head;
                while (temp.next != null) {
                    temp = temp.next;
                }
                temp.next = newNode;
                newNode.next = null;
            }
            if (first != null) {
                first = first.next;
            }
            if (second != null) {
                second = second.next;
            }
        }
        if (q > 0) {
            ListNode newNode = new ListNode(q);
            temp = head;
            while (temp.next != null) {
                temp = temp.next;
            }
            temp.next = newNode;
            newNode.next = null;
        }
    }
}
```

```
    }  
    return head;  
}  
}
```

### 3. Longest Substring Without Repeating Characters



Given a string `s`, find the length of the **longest substring** without repeating characters.

#### Example 1:

**Input:** `s = "abcabcbb"`

**Output:** 3

**Explanation:** The answer is "abc", with the length of 3.

#### Example 2:

**Input:** `s = "bbbbbb"`

**Output:** 1

**Explanation:** The answer is "b", with the length of 1.

#### Example 3:

**Input:** `s = "pwwkew"`

**Output:** 3

**Explanation:** The answer is "wke", with the length of 3.

Notice that the answer must be a substring, "pwke" is a subsequence and not a substring.

#### Constraints:

- $0 \leq s.length \leq 5 \times 10^4$
- `s` consists of English letters, digits, symbols and spaces.

1. Note :  $j \Rightarrow$  In short, when we see a duplicate character in the current substring, move the left pointer past the first occurrence of this character recorded in the map. (This cuts out unnecessary steps

```

public class Solution {
    public int lengthOfLongestSubstring(String s) {
        int result = 0;
        int[] cache = new int[256];
        for (int i = 0, j = 0; i < s.length(); i++) {
            j = (cache[s.charAt(i)] > 0) ? Math.max(j, cache[s.charAt(i)])
: j;

            cache[s.charAt(i)] = i + 1;
            result = Math.max(result, i - j + 1);
        }
        return result;
    }
}

```

2.

```

public int lengthOfLongestSubstring(String s) {
    if (s.length()==0) return 0;
    HashMap<Character, Integer> map = new HashMap<Character, Integer>
();
    int max=0;
    for (int i=0, j=0; i<s.length(); ++i){
        if (map.containsKey(s.charAt(i))){
            j = Math.max(j,map.get(s.charAt(i))+1);// just updating val
ue of j to new char index
        }
        map.put(s.charAt(i),i);
        max = Math.max(max,i-j+1);
    }
    return max;
}

```

## 11. Container With Most Water



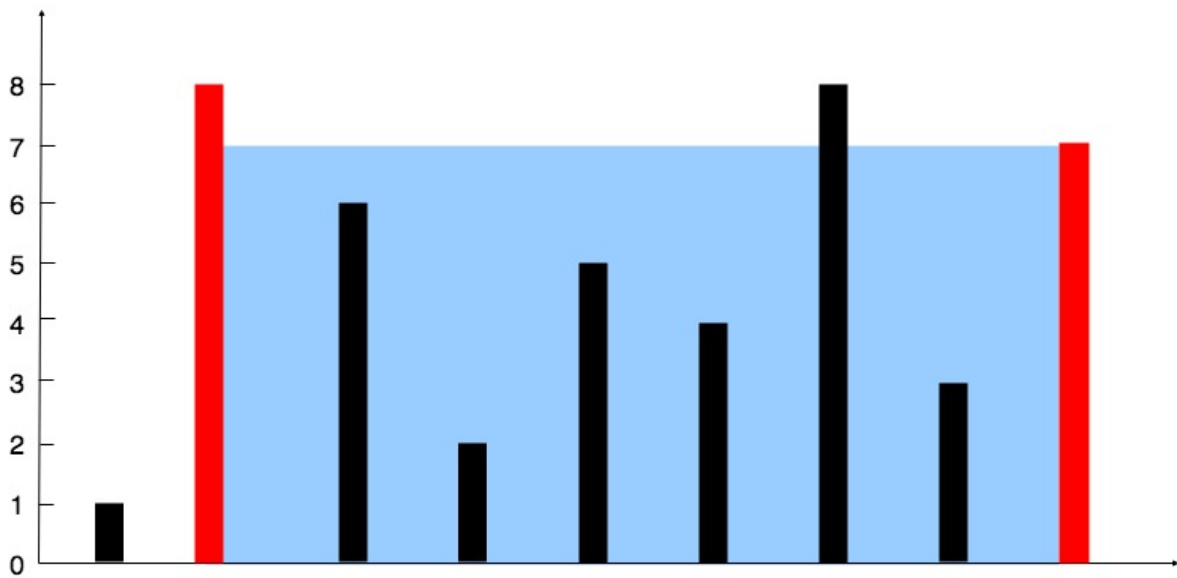
You are given an integer array `height` of length `n`. There are `n` vertical lines drawn such that the two endpoints of the  $i^{\text{th}}$  line are  $(i, 0)$  and  $(i, \text{height}[i])$ .

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 [1,8,6,2,5,4,8,3,7]

#### Example 2:

**Input:** height = [1,1]

**Output:** 1

#### Constraints:

- $n == \text{height.length}$
- $2 \leq n \leq 10^5$
- $0 \leq \text{height}[i] \leq 10^4$

---

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  $i \neq j$ ,  $i \neq k$ , and  $j \neq k$ , 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 matter.

### 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:

- $3 \leq \text{nums.length} \leq 3000$
- $-10^5 \leq \text{nums}[i] \leq 10^5$

---

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;
    }
}

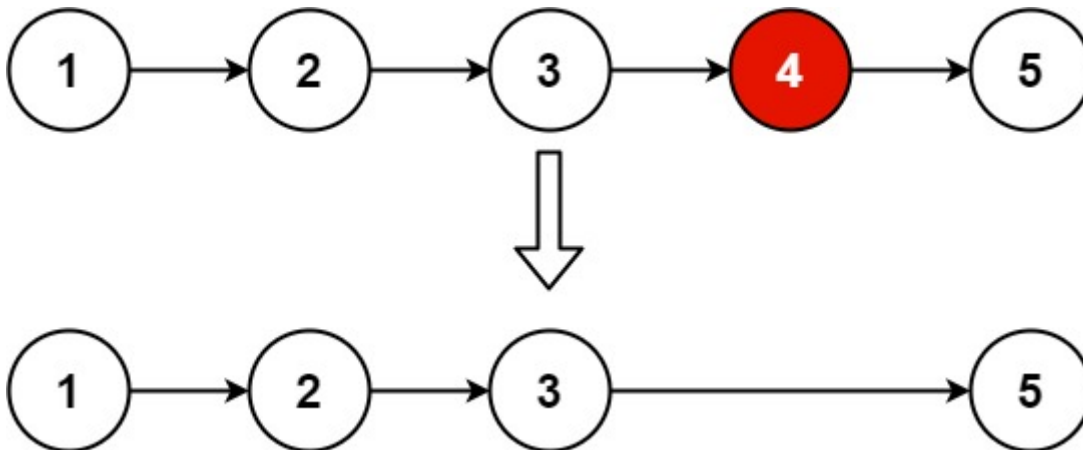
```

## 19. Remove Nth Node From End of List [↗](#)



Given the head of a linked list, remove the  $n^{\text{th}}$  node from the end of the list and return its head.

**Example 1:**



**Input:** head = [1,2,3,4,5], n = 2

**Output:** [1,2,3,5]

**Example 2:**

**Input:** head = [1], n = 1

**Output:** []

**Example 3:**

**Input:** head = [1,2], n = 1

**Output:** [1]

**Constraints:**

- The number of nodes in the list is sz .
- $1 \leq sz \leq 30$
- $0 \leq \text{Node.val} \leq 100$
- $1 \leq n \leq sz$

**Follow up:** Could you do this in one pass?

---

```
class Solution {  
  
    public ListNode removeNthFromEnd(ListNode head, int n) {  
        if (head == null || head.next == null) return null;  
  
        ListNode temp = new ListNode(0);  
        temp.next = head;  
        ListNode first = temp, second = temp;  
  
        while (n > 0) {  
            second = second.next;  
            n--;  
        }  
  
        while (second.next != null) {  
            second = second.next;  
            first = first.next;  
        }  
  
        first.next = first.next.next;  
        return temp.next;  
    }  
}
```

---

## 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:**

- $1 \leq s.length \leq 10^4$
- 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();
    }
}
```

## 21. Merge Two Sorted Lists

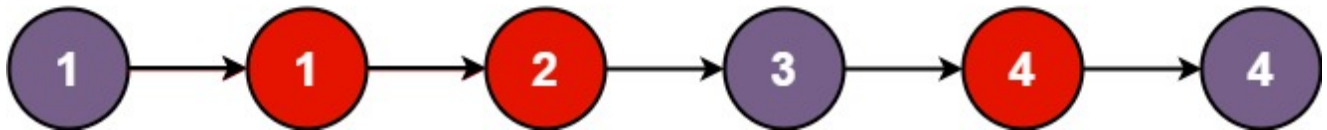
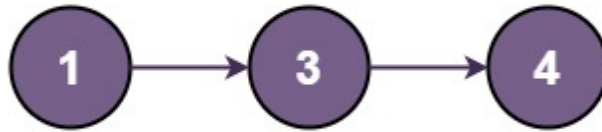
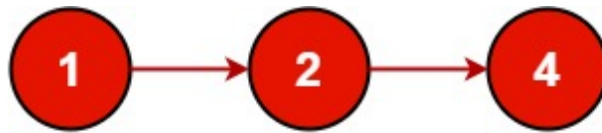


You are given the heads of two sorted linked lists `list1` and `list2`.

Merge the two lists in a one **sorted** list. The list should be made by splicing together the nodes of the first two lists.

Return *the head of the merged linked list*.

**Example 1:**



**Input:** list1 = [1,2,4], list2 = [1,3,4]  
**Output:** [1,1,2,3,4,4]

**Example 2:**

**Input:** list1 = [], list2 = []  
**Output:** []

**Example 3:**

**Input:** list1 = [], list2 = [0]  
**Output:** [0]

**Constraints:**

- The number of nodes in both lists is in the range [0, 50] .
  - $-100 \leq \text{Node.val} \leq 100$
  - Both list1 and list2 are sorted in **non-decreasing** order.
-

```

/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next;
 * }
 */
class Solution {
    public ListNode mergeTwoLists(ListNode list1, ListNode list2) {
        if(list1 == null) return list2;
        if(list2 == null) return list1;
        if(list1.val < list2.val){
            list1.next = mergeTwoLists(list1.next, list2);
            return list1;
        } else{
            list2.next = mergeTwoLists(list1, list2.next);
            return list2;
        }
    }
}

```

## 33. Search in Rotated Sorted Array [↗](#)



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

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

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

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

### Example 1:

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

**Output:** `4`

### Example 2:

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

**Output:** -1

**Example 3:**

**Input:** nums = [1], target = 0

**Output:** -1

**Constraints:**

- $1 \leq \text{nums.length} \leq 5000$
- $-10^4 \leq \text{nums}[i] \leq 10^4$
- All values of nums are **unique**.
- nums is an ascending array that is possibly rotated.
- $-10^4 \leq \text{target} \leq 10^4$

Do note the conditions used

```
public class Solution {
    public int search(int[] nums, int target) {
        int start = 0;
        int end = nums.length - 1;
        while (start <= end){
            int mid = (start + end) / 2;
            if (nums[mid] == target)
                return mid;

            if (nums[start] <= nums[mid]){
                if (target < nums[mid] && target >= nums[start])
                    end = mid - 1;
                else
                    start = mid + 1;
            }

            if (nums[mid] <= nums[end]){
                if (target > nums[mid] && target <= nums[end])
                    start = mid + 1;
                else
                    end = mid - 1;
            }
        }
        return -1;
    }
}
```



## 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:

- $1 \leq \text{strs.length} \leq 10^4$
- $0 \leq \text{strs}[i].\text{length} \leq 100$
- `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);
    }
}

```

## 70. Climbing Stairs



You are climbing a staircase. It takes  $n$  steps to reach the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

### Example 1:

**Input:**  $n = 2$

**Output:** 2

**Explanation:** There are two ways to climb to the top.

1. 1 step + 1 step
2. 2 steps

### Example 2:

**Input:**  $n = 3$

**Output:** 3

**Explanation:** There are three ways to climb to the top.

1. 1 step + 1 step + 1 step
2. 1 step + 2 steps
3. 2 steps + 1 step

### Constraints:

- $1 \leq n \leq 45$

---

## Recursion

### Does it implement recursion ?

If asking for total possible steps, min/max possible value.

### How to implement recursion?

- think in terms of indexes
- set the base condition
- apply possible operations on the indexes
- for total: add all operations
- for min/max: min/max of the operation values

## Dynamic Programming (DP)

Optimise the recursion problem with

1. Memoization: storing values in array in top down (recursion tree) for quick access.
2. Tabulation: use the bottom-up approach which is to save the base condition value in the array and then apply the for loop till the  $n$ , compute the value of next by prev calculated and stored value of array.
3. Optimise space complexity on top of Tabulation: if there is condition that refers to previous values then no need of array, instead use the variables to store prev (two generally) values and update it's prev with next in every loop.

Memoization: top down approach. Store values in array to avoid repeated calculation.

```

class Solution {
    public int fun(int n,int []dpArray)
    {
        if(n==0 || n==1) return 1;
        if(dpArray[n]!=-1) return dpArray[n];

        int singleStep= fun(n-1,dpArray);
        int twoStep= fun(n-2,dpArray);
        return dpArray[n]=singleStep+twoStep;
    }
    public int climbStairs(int n) {
        int []dpArray=new int [n+1];
        for(int i=0;i<=n;i++)
        {
            dpArray[i]=-1;
        }
        return fun(n,dpArray);
    }
}

```

Tabulation:

```

class Solution {

    public int climbStairs(int n) {
        int []dpArray=new int [n+1];
        dpArray[0]=1;
        dpArray[1]=1;
        for(int i=2;i<=n;i++)
        {
            dpArray[i]=dpArray[i-1]+dpArray[i-2];
        }
        return dpArray[n];
    }
}

```

Optimised space in tabulation:

```
class Solution {  
  
    public int climbStairs(int n) {  
        int prev2=1;  
        int prev1=1;  
        for(int i=2;i<=n;i++)  
        {  
            int output=prev1+prev2;  
            prev2=prev1;  
            prev1=output;  
        }  
        return prev1;  
    }  
}
```

## 74. Search a 2D Matrix [↗](#)



You are given an  $m \times n$  integer matrix `matrix` with the following two properties:

- Each row is sorted in non-decreasing order.
- The first integer of each row is greater than the last integer of the previous row.

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

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

**Example 1:**

1	3	5	7
10	11	16	20
23	30	34	60

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

**Output:** `true`

**Example 2:**

1	3	5	7
10	11	16	20
23	30	34	60

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

**Output:** false

#### Constraints:

- $m == \text{matrix.length}$
- $n == \text{matrix}[i].\text{length}$
- $1 \leq m, n \leq 100$
- $-10^4 \leq \text{matrix}[i][j], \text{target} \leq 10^4$

```
class Solution {
    public boolean searchMatrix(int[][] matrix, int target) {
        for(int i=0;i<matrix.length;i++)
        {
            if(matrix[i][matrix[0].length-1]>=target && matrix[i][0]<=target)
            {
                if(Arrays.binarySearch(matrix[i],target)>=0)
                    return true;
            }
        }
        return false;
    }
}
```

## 76. Minimum Window Substring [↗](#)



Given two strings *s* and *t* of lengths *m* and *n* respectively, return the **minimum window substring** of *s* such that every character in *t* (**including duplicates**) is included in the window. If there is no such substring, return the empty string `""`.


The testcases will be generated such that the answer is **unique**.

**Example 1:**

**Input:** `s = "ADOBECODEBANC", t = "ABC"`

**Output:** `"BANC"`

**Explanation:** The minimum window substring "BANC" includes 'A', 'B', and 'C' f



**Example 2:**

**Input:** `s = "a", t = "a"`

**Output:** `"a"`

**Explanation:** The entire string s is the minimum window.

**Example 3:**

**Input:** `s = "a", t = "aa"`

**Output:** `""`

**Explanation:** Both 'a's from t must be included in the window.  
Since the largest window of s only has one 'a', return empty string.

**Constraints:**

- `m == s.length`
- `n == t.length`
- `1 <= m, n <= 105`
- `s` and `t` consist of uppercase and lowercase English letters.

**Follow up:** Could you find an algorithm that runs in  $O(m + n)$  time?

---

```

class Solution {

    //sliding window
    public String minWindow(String s, String t) {
        HashMap<Character, Integer> map = new HashMap<>();

        for (char x : t.toCharArray()) {
            map.put(x, map.getOrDefault(x, 0) + 1);
        }

        int matched = 0;
        int start = 0;
        int minLen = s.length() + 1;
        int subStr = 0;
        for (int endWindow = 0; endWindow < s.length(); endWindow++) {
            char right = s.charAt(endWindow);
            if (map.containsKey(right)) {
                map.put(right, map.get(right) - 1);
                if (map.get(right) == 0) matched++;
            }

            while (matched == map.size()) {
                if (minLen > endWindow - start + 1) {
                    minLen = endWindow - start + 1;
                    subStr = start;
                }
                char deleted = s.charAt(start++);
                if (map.containsKey(deleted)) {
                    if (map.get(deleted) == 0) matched--;
                    map.put(deleted, map.get(deleted) + 1);
                }
            }
        }
        return minLen > s.length() ? "" : s.substring(subStr, subStr + minLen);
    }
}

```

## 104. Maximum Depth of Binary Tree

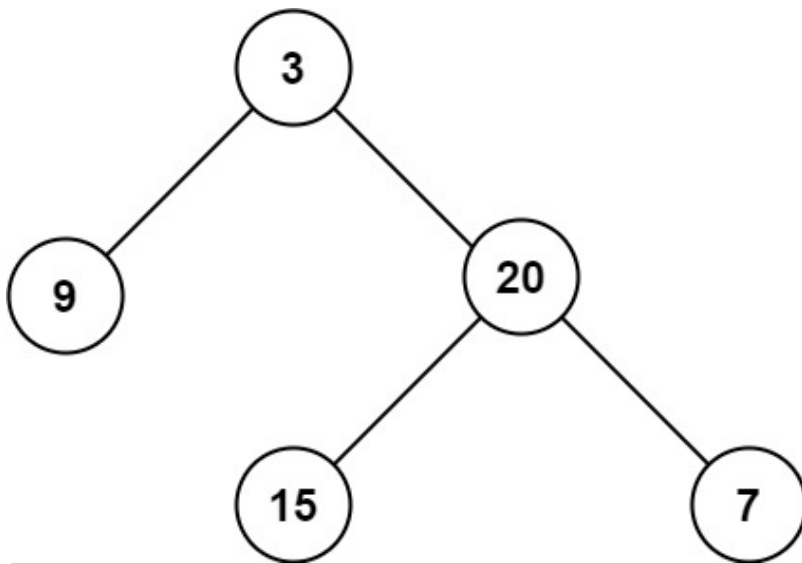


Given the `root` of a binary tree, return *its maximum depth*.

A binary tree's **maximum depth** is the number of nodes along the longest path from the root node down to the farthest leaf node.

**Example 1:**





**Input:** root = [3,9,20,null,null,15,7]  
**Output:** 3

**Example 2:**

**Input:** root = [1,null,2]  
**Output:** 2

**Constraints:**

- The number of nodes in the tree is in the range  $[0, 10^4]$  .
  - $-100 \leq \text{Node.val} \leq 100$
-

```

/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {
    public int maxDepth(TreeNode root) {
//         if(root==null)
//             return 0;
//         int depth=0;
//         Queue<TreeNode> q= new LinkedList<>();
//         q.add(root);
//         while(q.size()!=0)
//         {
//             int initialSize=q.size();
//             for(int i=0;i<initialSize;i++)
//             {
//                 System.out.println("remove node:"+ q.peek().val);
//                 TreeNode node=q.remove();
//                 if(node.left!=null)
//                 {
//                     q.add(node.left);
//                     System.out.println("left node:"+node.left.val);
//                 }
//                 if(node.right!=null)
//                 {
//                     q.add(node.right);
//                     System.out.println("right node:"+node.right.val);
//                 }
//             }
//             System.out.println("depth before==" +depth);

//             depth++;
//             System.out.println("depth after removed==" +depth);

//         }
//         return depth;
//         // Base Condition
//         if(root == null) return 0;
//         // Hypothesis
//         int left = maxDepth(root.left);

```

```
        int right = maxDepth(root.right);  
        // Induction  
        return Math.max(left, right) + 1;  
    }  
}
```

## 121. Best Time to Buy and Sell Stock



You are given an array `prices` where `prices[i]` is the price of a given stock on the  $i^{\text{th}}$  day.

You want to maximize your profit by choosing a **single day** to buy one stock and choosing a **different day in the future** to sell that stock.

Return *the maximum profit you can achieve from this transaction*. If you cannot achieve any profit, return `0`.

### Example 1:

**Input:** `prices = [7,1,5,3,6,4]`

**Output:** `5`

**Explanation:** Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 5.  
Note that buying on day 2 and selling on day 1 is not allowed because you must buy before you sell.

### Example 2:

**Input:** `prices = [7,6,4,3,1]`

**Output:** `0`

**Explanation:** In this case, no transactions are done and the max profit = 0.

### Constraints:

- $1 \leq \text{prices.length} \leq 10^5$
- $0 \leq \text{prices}[i] \leq 10^4$

Sliding window : <https://www.youtube.com/watch?v=GcW4mgmgSbw> (<https://www.youtube.com/watch?v=GcW4mgmgSbw>)

```

class Solution {
    public int maxProfit(int[] prices) {
        int lsf = Integer.MAX_VALUE;
        int op = 0;
        int pist = 0;

        for(int i = 0; i < prices.length; i++){
            lsf=(lsf<prices[i])?lsf:prices[i];
            op=(op>prices[i]-lsf)?op:prices[i]-lsf;
        }
        return op;
    }
}

```

Sliding window/ two pointer

```

class Solution {
    public int maxProfit(int[] prices) {
        int op = 0;
        int start = 0;
        int end = 1;
        while(end<prices.length)
        {
            if(prices[start]<prices[end])
            {
                op=Math.max(op,prices[end]-prices[start]);
            }
            else
            {
                start=end;
            }
            end++;
        }
        return op;
    }
}

```

## 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 characters. Since an empty string reads the same forward and backward, it is a palindrome.

#### Constraints:

- $1 \leq s.length \leq 2 * 10^5$
- 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:

- $0 \leq \text{nums.length} \leq 10^5$
- $-10^9 \leq \text{nums}[i] \leq 10^9$

```
class Solution {
    public int longestConsecutive(int[] nums) {
        // finding the number of digits that are continuous
        Arrays.sort(nums);
        //get the result as maximum continuous numbers
        int result=1;
        //continuous number count for each count
        int conti=1;
        for(int i=0;i<nums.length-1;i++)
        {
            //continuous 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 continuous
            else
            {
                conti=1;
            }
        }
        //if length of nums is zero then no continuous number
        if(nums.length==0)
            return 0;
        else
            return result;
    }
}
```

## 141. Linked List Cycle

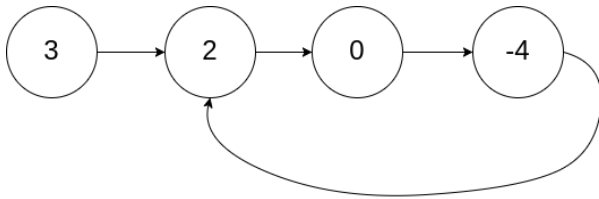


Given `head` , the head of a linked list, determine if the linked list has a cycle in it.

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.**

Return `true` if there is a cycle in the linked list. Otherwise, return `false`.

#### Example 1:

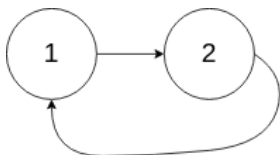


**Input:** `head = [3,2,0,-4], pos = 1`

**Output:** `true`

**Explanation:** There is a cycle in the linked list, where the tail connects to

#### Example 2:



**Input:** `head = [1,2], pos = 0`

**Output:** `true`

**Explanation:** There is a cycle in the linked list, where the tail connects to

#### Example 3:



**Input:** `head = [1], pos = -1`

**Output:** `false`

**Explanation:** There is no cycle in the linked list.

#### Constraints:

- The number of the nodes in the list is in the range  $[0, 10^4]$ .
- $-10^5 \leq \text{Node.val} \leq 10^5$
- `pos` is `-1` or a **valid index** in the linked-list.

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



```

/**
 * Definition for singly-linked list.
 * class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
 * }
 */
public class Solution {
    public boolean hasCycle(ListNode head) {
        ListNode fast=head;
        ListNode slow=head;
        while(fast!=null && slow!=null && fast.next!=null)
        {
            slow=slow.next;
            if(fast.next!=null)
                fast=fast.next.next;

            if(fast==slow)
                return true;
        }
        return false;
    }
}

```

## 143. Reorder List [↗](#)



You are given the head of a singly linked-list. The list can be represented as:

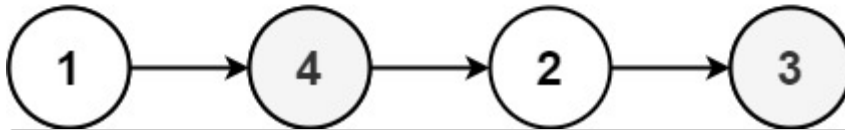
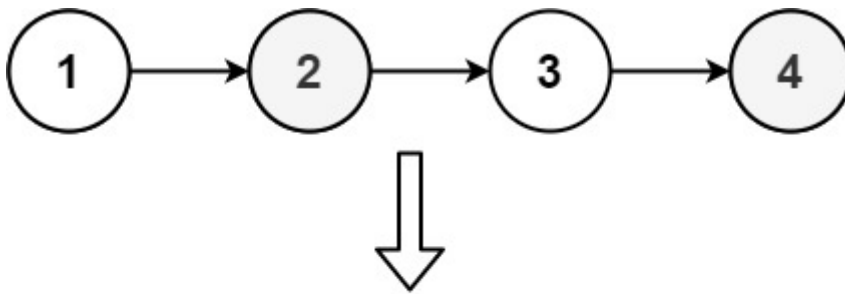
$$L_0 \rightarrow L_1 \rightarrow \dots \rightarrow L_{n-1} \rightarrow L_n$$

Reorder the list to be on the following form:

$$L_0 \rightarrow L_n \rightarrow L_1 \rightarrow L_{n-1} \rightarrow L_2 \rightarrow L_{n-2} \rightarrow \dots$$

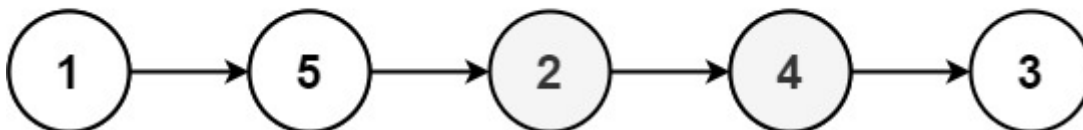
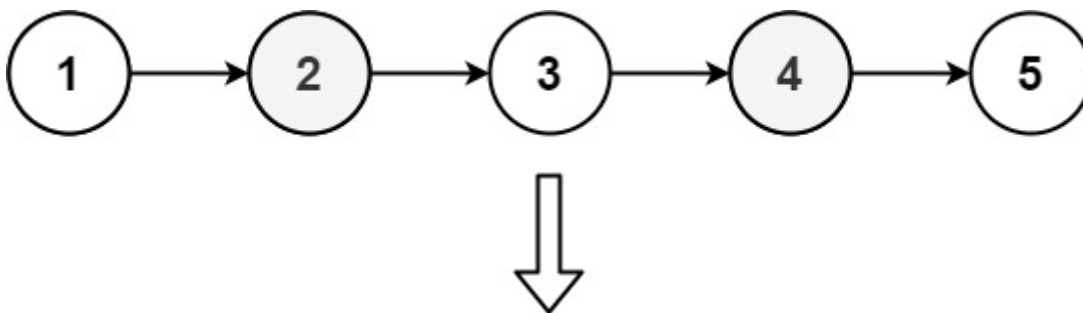
You may not modify the values in the list's nodes. Only nodes themselves may be changed.

**Example 1:**



**Input:** head = [1,2,3,4]  
**Output:** [1,4,2,3]

**Example 2:**



**Input:** head = [1,2,3,4,5]  
**Output:** [1,5,2,4,3]

**Constraints:**

- The number of nodes in the list is in the range  $[1, 5 \cdot 10^4]$ .
  - $1 \leq \text{Node.val} \leq 1000$
-

```
public void reorderList(ListNode head) {

    Stack<ListNode> stack = new Stack<>();
    ListNode curr = head;
    while (curr != null) {
        stack.push(curr);
        curr = curr.next;
    }

    ListNode headCurr = head;

    int size = stack.size();
    while (stack.size() > Math.ceil((double)size/2)) {
        ListNode next = stack.pop();
        ListNode nextNext = headCurr.next;
        next.next = null;
        headCurr.next = next;
        next.next = nextNext;
        headCurr = nextNext;
    }

    if (headCurr != null)
        headCurr.next = null;

}
```

```

public void reorderList(ListNode head) {
    if (head == null) {
        return;
    }

    // Find the middle node
    ListNode slow = head, fast = head.next;
    while (fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;
    }

    // Reverse the second half
    ListNode head2 = reverse(slow.next);
    slow.next = null;

    // Link the two halves together
    while (head != null && head2 != null) {
        ListNode tmp1 = head.next;
        ListNode tmp2 = head2.next;
        head2.next = head.next;
        head.next = head2;
        head = tmp1;
        head2 = tmp2;
    }
}

private ListNode reverse(ListNode n) {
    ListNode prev = null;
    ListNode cur = n;
    while (cur != null) {
        ListNode tmp = cur.next;
        cur.next = prev;
        prev = cur;
        cur = tmp;
    }
    return prev;
}

```

## 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](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:

- $1 \leq \text{tokens.length} \leq 10^4$
  - $\text{tokens}[i]$  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)
```

## 153. Find Minimum in Rotated Sorted Array



Suppose an array of length  $n$  sorted in ascending order is **rotated** between 1 and  $n$  times. For example, the array `nums = [0,1,2,4,5,6,7]` might become:

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

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

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

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

#### Example 1:

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

**Output:** `1`

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

#### Example 2:

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

**Output:** `0`

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

#### Example 3:

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

**Output:** `11`

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

#### Constraints:

- $n == \text{nums.length}$
  - $1 \leq n \leq 5000$
  - $-5000 \leq \text{nums}[i] \leq 5000$
  - All the integers of `nums` are **unique**.
  - `nums` is sorted and rotated between 1 and `n` times.
-

```

public class Solution {
    public int findMin(int[] num) {
        if (num == null || num.length == 0) {
            return 0;
        }
        if (num.length == 1) {
            return num[0];
        }
        int start = 0, end = num.length - 1;
        while (start < end) {
            int mid = (start + end) / 2;
            if (mid > 0 && num[mid] < num[mid - 1]) {
                return num[mid];
            }
            if (num[start] <= num[mid] && num[mid] > num[end]) {
                start = mid + 1;
            } else {
                end = mid - 1;
            }
        }
        return num[start];
    }
}

```

```

class Solution {
    public int findMin(int[] nums) {
        Arrays.sort(nums);
        return nums[0];
    }
}

```

## 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:

- $-2^{31} \leq \text{val} \leq 2^{31} - 1$
  - Methods `pop`, `top` and `getMin` operations will always be called on **non-empty** stacks.
  - At most  $3 \cdot 10^4$  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[index1]` and `numbers[index2]` where  $1 \leq \text{index}_1 < \text{index}_2 \leq \text{numbers.length}$ .

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

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

Your solution must use only constant extra space.

#### Example 1:

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

**Output:** [1,2]

**Explanation:** The sum of 2 and 7 is 9. Therefore,  $\text{index}_1 = 1$ ,  $\text{index}_2 = 2$ . We return [1, 2].

#### Example 2:

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

**Output:** [1,3]

**Explanation:** The sum of 2 and 4 is 6. Therefore  $\text{index}_1 = 1$ ,  $\text{index}_2 = 3$ . We return [1, 3].

#### Example 3:

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

**Output:** [1,2]

**Explanation:** The sum of -1 and 0 is -1. Therefore  $\text{index}_1 = 1$ ,  $\text{index}_2 = 2$ . We return [1, 2].

#### Constraints:

- $2 \leq \text{numbers.length} \leq 3 * 10^4$
- $-1000 \leq \text{numbers}[i] \leq 1000$
- numbers is sorted in **non-decreasing order**.
- $-1000 \leq \text{target} \leq 1000$
- 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;
    }
}

```

## 200. Number of Islands



Given an  $m \times n$  2D binary grid `grid` which represents a map of '1' s (land) and '0' s (water), return *the number of islands*.

An **island** is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

### Example 1:

```

Input: grid = [
  ["1","1","1","1","0"],
  ["1","1","0","1","0"],
  ["1","1","0","0","0"],
  ["0","0","0","0","0"]
]
Output: 1

```

### Example 2:

**Input:** grid = [  
 ["1", "1", "0", "0", "0"],  
 ["1", "1", "0", "0", "0"],  
 ["0", "0", "1", "0", "0"],  
 ["0", "0", "0", "1", "1"]  
]

**Output:** 3

**Constraints:**

- m == grid.length
  - n == grid[i].length
  - 1 <= m, n <= 300
  - grid[i][j] is '0' or '1'.
-

```

class Solution {
    char[][] g;
    int m;
    int n;
    public int numIslands(char[][] grid) {
        g=grid;
        if(g.length==0)
            return 0;
        m=g.length;
        n=g[0].length;
        int count=0;
        for(int i=0;i<m;i++)
        {
            for(int j=0;j<n;j++)
            {
                if(g[i][j]=='1')
                {
                    dfs(i,j);
                    count++;
                }
            }
        }
        return count;
    }
    public void dfs(int i,int j)
    {
        if(i<0 || i>=m || j<0 || j>=n || g[i][j]!='1')
            return;
        g[i][j]='0';
        dfs(i+1,j);
        dfs(i-1,j);
        dfs(i,j+1);
        dfs(i,j-1);
    }
}

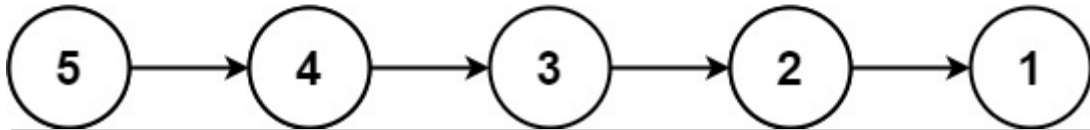
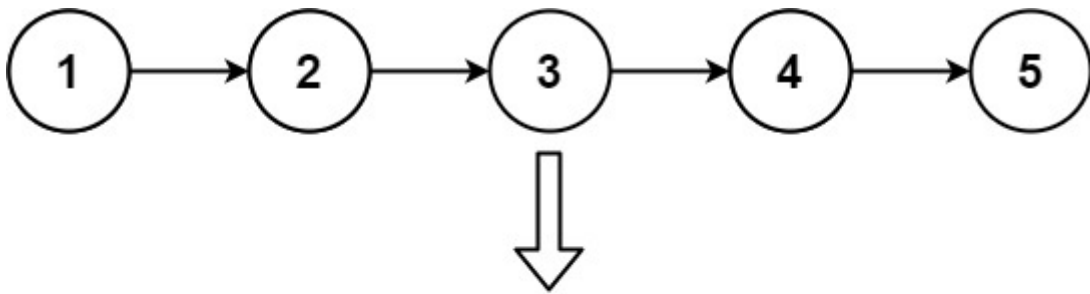
```

## 206. Reverse Linked List



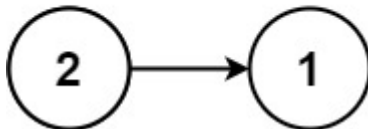
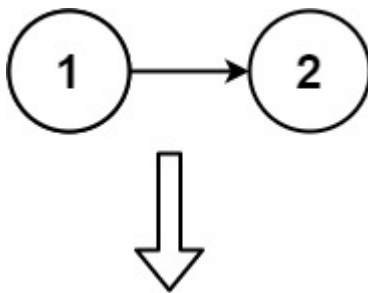
Given the `head` of a singly linked list, reverse the list, and return *the reversed list*.

**Example 1:**



**Input:** head = [1,2,3,4,5]  
**Output:** [5,4,3,2,1]

**Example 2:**



**Input:** head = [1,2]  
**Output:** [2,1]

**Example 3:**

**Input:** head = []  
**Output:** []

**Constraints:**

- The number of nodes in the list is the range [0, 5000] .
- $-5000 \leq \text{Node.val} \leq 5000$

**Follow up:** A linked list can be reversed either iteratively or recursively. Could you implement both?

---

```

/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next;
 * }
 */
class Solution {
    public ListNode reverseList(ListNode head) {
        if(head==null||head.next==null)
            return head;
        ListNode right=head.next;
        ListNode left=head;
        ListNode prev=null;
        while(right!=null)
        {
            left.next=prev;
            prev=left;
            left=right;
            right=right.next;
        }
        left.next=prev;
        head=left;
        return head;
    }
}

```

## 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:**

- $1 \leq \text{nums.length} \leq 10^5$
- $-10^9 \leq \text{nums}[i] \leq 10^9$

```
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:**

- $2 \leq \text{nums.length} \leq 10^5$
- $-30 \leq \text{nums}[i] \leq 30$
- 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:**

- $1 \leq s.length, t.length \leq 5 * 10^4$
- 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;
    }
}
```

## 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*.

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

### Example 1:

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

### Example 2:

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

### Constraints:

- $1 \leq n \leq 10^5$
- `nums.length == n + 1`
- $1 \leq \text{nums}[i] \leq n$
- 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 in linear runtime complexity?
-

```

class Solution {
    public int findDuplicate(int[] nums) {
        int arr[]=new int[nums.length];
        for(int i=0;i<nums.length;i++)
        {
            arr[nums[i]-1]++;
        }
        for(int i=0;i<arr.length;i++)
        {
            if(arr[i]>1)
                return i+1;
        }
        return -1;
    }
}

```

```

//Fast and slow pointer approach
// Time Complexity: O(n)
// Space Complexity: O(1)

class Solution {

    public int findDuplicate(int[] nums) {
        int fast = nums[0];
        int slow = nums[0];
        boolean first = true;
        while (first || fast != slow) {
            if (first) first = false;
            slow = nums[slow];
            fast = nums[nums[fast]];
            if (fast == slow) break;
        }
        int slow2 = nums[0];
        while (slow2 != slow) {
            if (first) first = false;
            slow2 = nums[slow2];
            slow = nums[slow];
            if (slow2 == slow) return slow;
        }
        return slow;
    }
}

```

## 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:

- $1 \leq \text{nums.length} \leq 10^5$
- $-10^4 \leq \text{nums}[i] \leq 10^4$
- 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.compare(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;
    }
}
```

---

## 424. Longest Repeating Character Replacement ▼

You are given a string `s` and an integer `k`. You can choose any character of the string and change it to any other uppercase English character. You can perform this operation at most `k` times.

Return *the length of the longest substring containing the same letter you can get after performing the above operations*.

### Example 1:

**Input:** `s = "ABAB", k = 2`

**Output:** 4

**Explanation:** Replace the two 'A's with two 'B's or vice versa.

### Example 2:

**Input:** `s = "AABABBA", k = 1`

**Output:** 4

**Explanation:** Replace the one 'A' in the middle with 'B' and form "AABBBBA". The substring "BBBB" has the longest repeating letters, which is 4.

### Constraints:

- $1 \leq s.length \leq 10^5$
- `s` consists of only uppercase English letters.
- $0 \leq k \leq s.length$

---

need to check for the less frequent characters to be  $< k$ . `lessFreq = lengthOfTheWindow - mostFrequentCharacterFromMap`

```

class Solution {

    public int characterReplacement(String s, int k) {
        int[] arr = new int[26];
        int ans = 0;
        int max = 0;
        int i = 0;
        for (int j = 0; j < s.length(); j++) {
            arr[s.charAt(j) - 'A']++;
            max = Math.max(max, arr[s.charAt(j) - 'A']);
            if (j - i + 1 - max > k) {
                arr[s.charAt(i) - 'A']--;
                i++;
            }
            ans = Math.max(ans, j - i + 1);
        }
        return ans;
    }
}

```

## 567. Permutation in String



Given two strings `s1` and `s2`, return `true` if `s2` contains a permutation of `s1`, or `false` otherwise.

In other words, return `true` if one of `s1`'s permutations is the substring of `s2`.

### Example 1:

**Input:** `s1 = "ab", s2 = "eidbaooo"`

**Output:** `true`

**Explanation:** `s2` contains one permutation of `s1` ("ba").

### Example 2:

**Input:** `s1 = "ab", s2 = "eidboao"`

**Output:** `false`

### Constraints:

- $1 \leq s1.length, s2.length \leq 10^4$
- `s1` and `s2` consist of lowercase English letters.



```

class Solution {

    public boolean checkInclusion(String s1, String s2) {
        if (s2.length() < s1.length()) return false;
        int[] arr = new int[26];
        //add the values to the hash array
        for (int i = 0; i < s1.length(); i++) {
            arr[s1.charAt(i) - 'a']++;
        }
        int i = 0;
        int j = 0;
        //point j to it's position
        for (; j < s1.length(); j++) {
            arr[s2.charAt(j) - 'a']--;
        }
        j--;
        if (isEmpty(arr)) return true;
        while (j < s2.length()) {
            arr[s2.charAt(i) - 'a']++;
            i++;
            j++;
            if (j < s2.length()) arr[s2.charAt(j) - 'a']--;
            if (isEmpty(arr)) return true;
        }
        return isEmpty(arr);
    }

    public boolean isEmpty(int[] arr) {
        for (int i = 0; i < arr.length; i++) {
            if (arr[i] != 0) return false;
        }
        return true;
    }
}

```

## 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  $i^{\text{th}}$  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:

- $1 \leq \text{temperatures.length} \leq 10^5$
- $30 \leq \text{temperatures}[i] \leq 100$

#### 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;
}
```

---

## 704. Binary Search



Given an array of integers `nums` which is sorted in ascending order, and an integer `target` , write a function to search `target` in `nums` . If `target` exists, then return its index. Otherwise, return `-1` .

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

### Example 1:

**Input:** `nums = [-1,0,3,5,9,12]`, `target = 9`

**Output:** `4`

**Explanation:** 9 exists in `nums` and its index is 4

### Example 2:

**Input:** `nums = [-1,0,3,5,9,12]`, `target = 2`

**Output:** `-1`

**Explanation:** 2 does not exist in `nums` so return -1

### Constraints:

- $1 \leq \text{nums.length} \leq 10^4$
- $-10^4 < \text{nums}[i], \text{target} < 10^4$
- All the integers in `nums` are **unique**.
- `nums` is sorted in ascending order.

---

```
public int search(int[] nums, int target) {  
    if(Arrays.binarySearch(nums,target)>=0)  
        return Arrays.binarySearch(nums,target);  
    else  
        return -1;  
}
```

```

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

        int low = 0;
        int high = nums.length - 1;
        while(low <= high){
            int mid = low + (high - low) / 2;
            if(nums[mid] == target){
                return mid;
            }
            else if(nums[mid] < target){
                low = mid + 1;
            }
            else{
                high = mid - 1;
            }
        }
        return -1;
    }
}

```

## 853. Car Fleet



There are  $n$  cars going to the same destination along a one-lane road. The destination is  $target$  miles away.

You are given two integer array `position` and `speed`, both of length  $n$ , where `position[i]` is the position of the  $i^{th}$  car and `speed[i]` is the speed of the  $i^{th}$  car (in miles per hour).

A car can never pass another car ahead of it, but it can catch up to it and drive bumper to bumper **at the same speed**. The faster car will **slow down** to match the slower car's speed. The distance between these two cars is ignored (i.e., they are assumed to have the same position).

A **car fleet** is some non-empty set of cars driving at the same position and same speed. Note that a single car is also a car fleet.

If a car catches up to a car fleet right at the destination point, it will still be considered as one car fleet.

Return *the **number of car fleets** that will arrive at the destination*.

**Example 1:**

**Input:** target = 12, position = [10,8,0,5,3], speed = [2,4,1,1,3]

**Output:** 3

**Explanation:**

The cars starting at 10 (speed 2) and 8 (speed 4) become a fleet, meeting each other at 12.  
The car starting at 0 does not catch up to any other car, so it is a fleet by itself.  
The cars starting at 5 (speed 1) and 3 (speed 3) become a fleet, meeting each other at 7.  
Note that no other cars meet these fleets before the destination, so the answer is 3.

**Example 2:**

**Input:** target = 10, position = [3], speed = [3]

**Output:** 1

**Explanation:** There is only one car, hence there is only one fleet.

**Example 3:**

**Input:** target = 100, position = [0,2,4], speed = [4,2,1]

**Output:** 1

**Explanation:**

The cars starting at 0 (speed 4) and 2 (speed 2) become a fleet, meeting each other at 5.  
Then, the fleet (speed 2) and the car starting at 4 (speed 1) become one fleet, meeting each other at 100.

**Constraints:**

- $n == \text{position.length} == \text{speed.length}$
- $1 \leq n \leq 10^5$
- $0 < \text{target} \leq 10^6$
- $0 \leq \text{position}[i] < \text{target}$
- All the values of position are **unique**.
- $0 < \text{speed}[i] \leq 10^6$

---

1. TreeMap: Note: Sorting in stored values

```

public int carFleet(int target, int[] pos, int[] speed) {
    Map<Integer, Double> m = new TreeMap<>(Collections.reverseOrder
());
    for (int i = 0; i < pos.length; ++i)
        m.put(pos[i], (double)(target - pos[i]) / speed[i]);
    int res = 0; double cur = 0;
    for (double time : m.values()) {
        if (time > cur) {
            cur = time;
            res++;
        }
    }
    return res;
}

```

2. Note: 2D array sorting

```

public int carFleet(int target, int[] pos, int[] speed) {
    int N = pos.length, res = 0;
    double[][] cars = new double[N][2];
    for (int i = 0; i < N; ++i)
        cars[i] = new double[] {pos[i], (double)(target - pos[i]) / s
peed[i]};
    Arrays.sort(cars, (a, b) -> Double.compare(a[0], b[0]));
    double cur = 0;
    for (int i = N - 1; i >= 0 ; --i) {
        if (cars[i][1] > cur) {
            cur = cars[i][1];
            res++;
        }
    }
    return res;
}

```

## 875. Koko Eating Bananas



Koko loves to eat bananas. There are  $n$  piles of bananas, the  $i^{\text{th}}$  pile has `piles[i]` bananas. The guards have gone and will come back in  $h$  hours.

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

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

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

**Example 1:**

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

**Output:** 4

**Example 2:**

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

**Output:** 30

**Example 3:**

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

**Output:** 23

**Constraints:**

- $1 \leq \text{piles.length} \leq 10^4$
- $\text{piles.length} \leq h \leq 10^9$
- $1 \leq \text{piles}[i] \leq 10^9$

```
class Solution {
    public int minEatingSpeed(int[] piles, int H) {
        int low = 1, high = 1000000000, k = 0;
        while (low <= high) {
            //mid
            k = (low + high) / 2;
            //required result
            int h = 0;
            //Adding time to eat all piles with current rate of k
            for (int i = 0; i < piles.length; i++)
                h += Math.ceil(1.0 * piles[i] / k);
            //if h>H (time consumed more than available)
            if (h > H)
                low = k + 1;
            else
                high = k - 1;
        }
        return low;
    }
}
```