

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]`

Output: 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?

Interview Bit

```
public class Solution {  
    // DO NOT MODIFY THE ARGUMENTS WITH "final" PREFIX. IT IS READ ONLY  
    public int[] twoSum(final int[] A, int B) {  
  
        HashMap<Integer, Integer> map = new HashMap<>();  
        int[] ans = new int[2];  
        for (int i = 0; i < A.length; i++)  
        {  
            int req = B - A[i]; //correct  
            if (!map.containsKey(req)) //map me req ni h to apne ko daal do  
            {  
                if (!map.containsKey(A[i])) // agr mai khud present hun to  
                an na dalo qki index badh jaega  
                map.put(A[i], i);  
            }  
            else  
            {  
                ans[0] = map.get(req)+1;  
                ans[1] = i+1;  
                return ans; //first milte he return  
            }  
        }  
        return new int[0]; // ni mila to empty array  
    }  
}
```

```

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

        int i;
        int[] arr = new int[2];
        HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();

        for (i = 0; i < nums.length; i++)
        {
            map.put(target-nums[i], i);
        }

        for (i = 0; i < nums.length; i++)
        {
            if(map.containsKey(nums[i]) && map.get(nums[i]) != i)
            {
                arr[0] = map.get(nums[i]);
                arr[1] = i;
                break;
            }
        }

        return arr;
    }
}

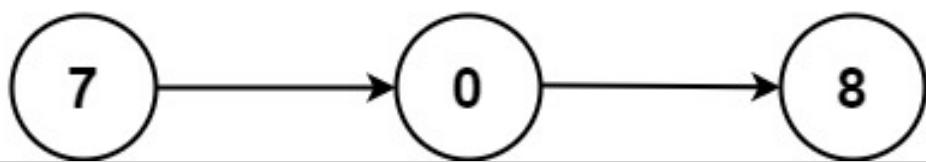
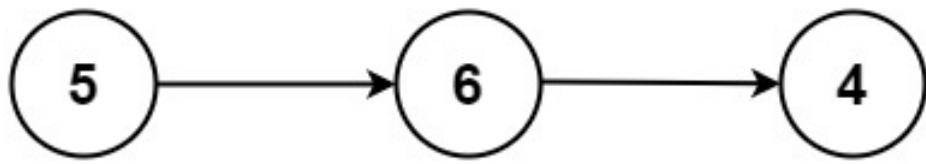
```

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.

```
class Solution {
    public ListNode addTwoNumbers(ListNode l1, ListNode l2)
    {

        //we are creating dummyhead to keep track of head
        ListNode dummyhead = new ListNode (-1);
        ListNode currNode = dummyhead;

        int sum = 0, carry = 0;

        while (l1 != null && l2 != null)
        {
            sum = l1.val + l2.val + carry;
            carry = 0;
            if (sum >= 10)
            {
                carry = sum/10;
                sum = sum % 10;
            }
            ListNode tempNode = new ListNode(sum);
            currNode.next = tempNode;
            currNode = currNode.next;
            l1 = l1.next;
            l2 = l2.next;
        }

        while (l1 != null)
        {
            sum = l1.val + carry;
            carry = 0;
            if (sum >= 10)
            {
                carry = sum/10;
                sum = sum % 10;
            }
            ListNode tempNode = new ListNode (sum);
            currNode.next = tempNode;
            currNode = currNode.next;
            l1 = l1.next;
        }

        while (l2 != null)
        {
            sum = l2.val + carry;
            carry = 0;
```

```

        if (sum >= 10)
        {
            carry = sum/10;
            sum = sum % 10;
        }
        ListNode tempNode = new ListNode(sum);
        currNode.next = tempNode;
        currNode = currNode.next;
        l2 = l2.next;
    }

    if (carry != 0)
    {
        ListNode tempNode = new ListNode(carry);
        currNode.next = tempNode;
        currNode = currNode.next;
    }

    return dummyhead.next;
}

}

```

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

Example 4:

Input: s = ""

Output: 0

Constraints:

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

```

class Solution {
    public int lengthOfLongestSubstring(String s)
    {
        Set<Character> set = new HashSet<>();

        char[] ch = s.toCharArray();
        int j = 0;
        int max = Integer.MIN_VALUE;

        for (int i = 0; i < ch.length;) //abcabcbb
        {
            if (!set.contains(ch[i])) //set does not contain... add it
            {
                set.add(ch[i]);
                i++;
            }
            else
            {
                max = Math.max(max, set.size()); //first duplicate aaya h...
                usse pahle tk ka size save kar lia
                while (set.contains(ch[i]) && j < i) //till we have not rem
                oved duplicate keep on removing
                {
                    set.remove(ch[j]);
                    j++;
                }
            }
        }
        max = Math.max(max, set.size()); //for "" and " "
        return max;
    }
}

```

4. Median of Two Sorted Arrays ↗

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

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

Example 1:

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

Output: 2.00000

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

Example 2:

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

Output: 2.50000

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

Example 3:

Input: nums1 = [0,0], nums2 = [0,0]

Output: 0.00000

Example 4:

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

Output: 1.00000

Example 5:

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

Output: 2.00000

Constraints:

- $\text{nums1.length} == m$
- $\text{nums2.length} == n$
- $0 \leq m \leq 1000$
- $0 \leq n \leq 1000$
- $1 \leq m + n \leq 2000$
- $-10^6 \leq \text{nums1}[i], \text{nums2}[i] \leq 10^6$

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

        int size1 = nums1.length;
        int size2 = nums2.length;
        int resultSize = size1 + size2;
        int result[] = new int[resultSize];
        int i = 0, j = 0, k = 0;

        while(i < size1 && j < size2)
        {
            if (nums1[i] < nums2[j])
            {
                result[k] = nums1[i];
                i++;k++;
            }
            else
            {
                result[k] = nums2[j];
                j++;k++;
            }
        }

        while (i < size1)
        {
            result[k] = nums1[i];
            i++;k++;
        }
        while (j < size2)
        {
            result[k] = nums2[j];
            j++;k++;
        }

        int index = (1 + resultSize)/2;
        double medain = 0.0;
        if (resultSize % 2 == 0)
        {
            medain = (double)(result[index-1] + result[index])/2;
        }
        else
        {
            medain = result[index -1];
        }
        return medain;
    }
}
```

```
    }  
}
```

5. Longest Palindromic Substring ↗

Given a string `s`, return *the longest palindromic substring* in `s`.

Example 1:

```
Input: s = "babad"  
Output: "bab"  
Note: "aba" is also a valid answer.
```

Example 2:

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

Example 3:

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

Example 4:

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

Constraints:

- `1 <= s.length <= 1000`
- `s` consist of only digits and English letters.

```

//272 ms, faster than 20.36%
class Solution {
    public String longestPalindrome(String s)
    {
        if (s == null || s == "") return null;

        int l = s.length();
        if (l == 1) return s;
        else if (l == 2 && s.charAt(0) == s.charAt(1)) return s;

        int start = 0, end = 0;
        boolean dp[][] = new boolean[l][l];//for checking last 2 char..it stores result for middle

        for (int gap = 0; gap < l; gap++)
        {
            for (int i = 0; i+gap < l; i++)
            {
                if (gap == 0)
                {
                    dp[i][i+gap] = true; //single character
                    start = i;
                    end = i+gap+1;
                    // ans = s.substring(i, i+gap+1);
                }
                else if (gap == 1)
                {
                    if (s.charAt(i) == s.charAt(i+gap))//2 characters
                    {
                        dp[i][i+gap] = true;
                        // ans = s.substring(i, i+gap+1);
                        start = i;
                        end = i+gap+1;
                    }
                    else
                    {
                        dp[i][i+gap] = false;
                    }
                }
                else if (gap > 1)//first & last character check kar rai..bech ka atrix se mil ra
                {
                    if (s.charAt(i) == s.charAt(i+gap) && dp[i+1][i+gap-1])//dp mddle ka bataega
                    {
                        dp[i][i+gap] = true;
                    }
                }
            }
        }
    }
}

```

```

        // ans = s.substring(i, i+gap+1);
        start = i;
        end = i+gap+1; //instead of doing substring, store
index
    }
    else
    {
        dp[i][i+gap] = false;
    }

}
}
}
// ans = s.substring(start, end);
return s.substring(start, end);
}
}

```

6. Zigzag Conversion ↗

The string "PAYPALISHIRING" is written in a zigzag pattern on a given number of rows like this:
 (you may want to display this pattern in a fixed font for better legibility)

P	A	H	N			
A	P	L	S	I	I	G
Y	I	R				

And then read line by line: "PAHNAPLSIIGYIR"

Write the code that will take a string and make this conversion given a number of rows:

```
string convert(string s, int numRows);
```

Example 1:

Input: s = "PAYPALISHIRING", numRows = 3
Output: "PAHNAPLSIIGYIR"

Example 2:

Input: s = "PAYPALISHIRING", numRows = 4

Output: "PINALSIGYAHRPI"

Explanation:

P	I	N
A	L S	I G
Y A	H R	
P	I	

Example 3:

Input: s = "A", numRows = 1

Output: "A"

Constraints:

- $1 \leq s.length \leq 1000$
 - s consists of English letters (lower-case and upper-case), ',', and '.'.
 - $1 \leq \text{numRows} \leq 1000$
-

```
/*
Array Size: 3
PAHN
APLSIIG
YIR
*****
Array Size: 4
PIN
ALSIG
YAHR
PI
*/
class Solution {
    public String convert(String s, int numRows)
    {
        StringBuffer sb = new StringBuffer();
        //array of list
        ArrayList<Character>[] listArray = new ArrayList[numRows];//jtni rows utni array bana dia

        for (int i = 0; i < listArray.length; i++)
        {
            listArray[i] = new ArrayList<Character>();
        }
        //System.out.println("Array Size: " + listArray.length);
        char ch[] = s.toCharArray();

        int row = 0;
        for (int i = 0; i < ch.length());//char array
        {
            row = 0;
            while (row < numRows && i < ch.length)//up --> down
            {
                listArray[row].add(ch[i]);
                row++;
                i++;
            }
            row = numRows - 2;//qki last row ni fill kar rai ulta jate time

            while (row > 0 && i < ch.length) //down --> up
            {
                listArray[row].add(ch[i]);
                row--;
                i++;
            }
        }
    }
}
```

```
        }

        for (List<Character> l : listArray)
        {
            for (char c : l)
            {
                sb.append(c);
            }
        }
        return sb.toString();
    }
}
```

7. Reverse Integer ↗



Given a signed 32-bit integer x , return x with its digits reversed. If reversing x causes the value to go outside the signed 32-bit integer range $[-2^{31}, 2^{31} - 1]$, then return 0.

Assume the environment does not allow you to store 64-bit integers (signed or unsigned).

Example 1:

```
Input: x = 123
Output: 321
```

Example 2:

```
Input: x = -123
Output: -321
```

Example 3:

```
Input: x = 120
Output: 21
```

Example 4:

```
Input: x = 0
Output: 0
```

Constraints:

- $-2^{31} \leq x \leq 2^{31} - 1$

```
class Solution {
    public int reverse(int x) {

        if (x < Integer.MIN_VALUE || x > Integer.MAX_VALUE)
            return 0;

        boolean isNegative = false;
        if (x<0)
        {
            isNegative = true;
            x = x*(-1);
        }

        long r = 0;
        while (x > 0)
        {
            r = r*10 + (x%10);
            if(r > Integer.MAX_VALUE || r < Integer.MIN_VALUE)
                return 0;

            x = x/10;
        }

        if (isNegative)
        {
            return (int)(-r);
        }

        return (int)r;
    }
}
```

9. Palindrome Number ↗

Given an integer x , return `true` if x is palindrome integer.

An integer is a **palindrome** when it reads the same backward as forward. For example, 121 is palindrome while 123 is not.

Example 1:

Input: x = 121

Output: true

Example 2:

Input: x = -121

Output: false

Explanation: From left to right, it reads -121. From right to left, it become

Example 3:

Input: x = 10

Output: false

Explanation: Reads 01 from right to left. Therefore it is not a palindrome.

Example 4:

Input: x = -101

Output: false

Constraints:

- $-2^{31} \leq x \leq 2^{31} - 1$

Follow up: Could you solve it without converting the integer to a string?

```

class Solution {
    public boolean isPalindrome(int x) {
        if (x == 0) return true;

        //if x ends with a zero, it's never a palindrome (you can't have a
        leading zero in the input number
        if (x < 0 || x%10 == 0) return false;

        int rev = 0;

        while (x > rev) //reverse tk he compare karo: compare half number
        {
            rev = (rev*10) + (x %10);
            x /= 10;
        }
        //even case; odd case : odd case me rev k pass ek extra number hoga
        use hata k compare karna h islia rev/10
        if (x == rev || x == rev/10) return true;

        return false;
    }
}

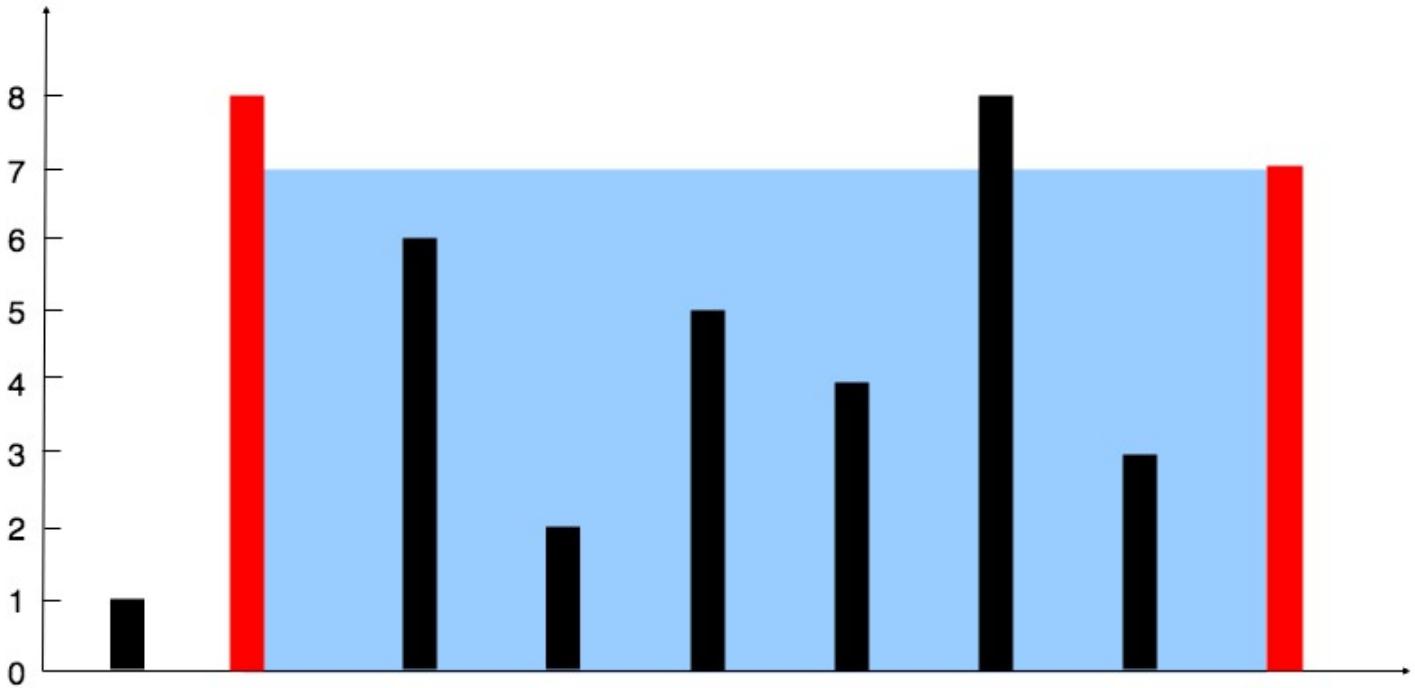
```

11. Container With Most Water ↗

Given n non-negative integers a_1, a_2, \dots, a_n , where each represents a point at coordinate (i, a_i) . n vertical lines are drawn such that the two endpoints of the line i is at (i, a_i) and $(i, 0)$. Find two lines, which, together with the x-axis forms a container, such that the container contains the most water.

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

Example 3:

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

Output: 16

Example 4:

Input: height = [1,2,1]

Output: 2

Constraints:

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

```
class Solution {
    public int maxArea(int[] height) {

        //2 pointer

        int i = 0, j = height.length-1;
        int maxWater = Integer.MIN_VALUE;
        int minHeight = 0;
        int area = 0;
        while (i < j)
        {
            minHeight = Math.min (height[i], height[j]);
            area = minHeight * (j-i);
            maxWater = Math.max(maxWater, area);

            if (height[i] < height[j])
            {
                i++;
            }
            else j--;
        }
        return maxWater;
    }
}
```

```
class Solution {
    public int maxArea(int[] height) {

        int lp = 0;
        int rp = height.length-1;
        int maxWater = Integer.MIN_VALUE;
        int minVal, water;

        while (lp < rp)
        {
            if (height[lp] < height[rp])
            {
                minVal = height[lp];

                water = (minVal) * (rp-lp);
                maxWater = Math.max(maxWater, water);
                lp++;
            }
            else if (height[lp] > height[rp])
            {
                minVal = height[rp];

                water = (minVal) * (rp-lp);
                System.out.println("water "+water);
                maxWater = Math.max(maxWater, water);
                rp--;
            }
            else if (height[lp] == height[rp])
            {
                minVal = height[rp];
                water = (minVal) * (rp-lp);
                maxWater = Math.max(maxWater, water);

                if (height[lp+1] > height[rp-1]) lp++;
                else rp--;
            }
        }

        return maxWater;
    }
}
```

12. Integer to Roman ↗

Roman numerals are represented by seven different symbols: I , V , X , L , C , D and M .

Symbol	Value
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

For example, 2 is written as II in Roman numeral, just two one's added together. 12 is written as XII , which is simply X + II . The number 27 is written as XXVII , which is XX + V + II .

Roman numerals are usually written largest to smallest from left to right. However, the numeral for four is not IIII . Instead, the number four is written as IV . Because the one is before the five we subtract it making four. The same principle applies to the number nine, which is written as IX . There are six instances where subtraction is used:

- I can be placed before V (5) and X (10) to make 4 and 9.
- X can be placed before L (50) and C (100) to make 40 and 90.
- C can be placed before D (500) and M (1000) to make 400 and 900.

Given an integer, convert it to a roman numeral.

Example 1:

```
Input: num = 3
Output: "III"
```

Example 2:

```
Input: num = 4
Output: "IV"
```

Example 3:

Input: num = 9

Output: "IX"

Example 4:

Input: num = 58

Output: "LVIII"

Explanation: L = 50, V = 5, III = 3.

Example 5:

Input: num = 1994

Output: "MCMXCIV"

Explanation: M = 1000, CM = 900, XC = 90 and IV = 4.

Constraints:

- 1 <= num <= 3999

Input: 2345 **Calc:** num/1000 = 2, (num%1000)/100 = 3, (num%100)/10 = 4, num%10 = 5

Output: "MMCCCXLV"

```
class Solution {

    public String intToRoman(int num)
    {
        String thousands[] = {"", "M", "MM", "MMM"};
        String hundreds[] = {"", "C", "CC", "CCC", "CD", "D", "DC", "DCC",
"DCCC", "CM"};
        String tens[] = {"", "X", "XX", "XXX", "XL", "L", "LX", "LXX", "LXXX",
"XC"};
        String ones[] = {"", "I", "II", "III", "IV", "V", "VI", "VII", "VIII",
"IX"};

        System.out.print("num/1000 = " + num/1000 + " (num%1000)/100 = " +
(num%1000)/100+ " (num%100)/10 = " + (num%100)/10+ " num%10 = " +
num%10);
        return thousands[num/1000] + hundreds[(num%1000)/100] + tens[(num%
100)/10] + ones[num%10];
    }
}
```

14. Longest Common Prefix ↗



Write a function to find the longest common prefix string amongst an array of strings.

If there is no common prefix, return an empty string "" .

Example 1:

```
Input: strs = ["flower", "flow", "flight"]
```

```
Output: "fl"
```

Example 2:

```
Input: strs = ["dog", "racecar", "car"]
```

```
Output: ""
```

Explanation: There is no common prefix among the input strings.

Constraints:

- $1 \leq \text{strs.length} \leq 200$
- $0 \leq \text{strs[i].length} \leq 200$
- strs[i] consists of only lower-case English letters.

```
//pahle word ko pura prifix maan lo, fir next word k acc last se ek ek character kam karte jao
class Solution {
    public String longestCommonPrefix(String[] strs) {

        String lcp = "";
        if (strs == null || strs.length == 0) return lcp;

        lcp = strs[0]; //taking 1st string as prefix

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

            while (strs[i].indexOf(lcp) != 0) {
                lcp = lcp.substring(0, lcp.length()-1); //reduce size of lcp..jb tk lcp pura na aa jae word me
            }
        }
        return lcp;
    }
}
```

```
class Solution {
    public String longestCommonPrefix(String[] strs) {

        int lenOfArr = strs.length;
        if (lenOfArr == 0)
        {
            return "";
        }
        String first = strs[0];
        String newString = "";

        for (int i = 1; i < lenOfArr; i++)
        {
            String compare = strs[i];

            for (int j = 0; j < compare.length() && j < first.length(); j++)
            {
                if(first.charAt(j) != compare.charAt(j))
                {
                    if (j == 0)
                    return "";

                    break;
                }
                else
                {
                    newString = newString + first.charAt(j);
                }
            }
            first = newString;
            newString = "";
        }
        return first;
    }
}
```

Prepare

```

class Solution {
    public String longestCommonPrefix(String[] strs) {

        if (strs == null || strs.length == 0) return "";

        StringBuffer ans = new StringBuffer();
        int index = 0;

        for (char c : strs[0].toCharArray()) //1st string ko char by char l
e rai
        {
            for (int i = 1; i < strs.length; i++)
            {
                //ya to ek string pura ho gaya,,,ya to char mismatch ho gay
a
                if (index >= strs[i].length() || c != strs[i].charAt(ind
x)) return ans.toString();
            }
            ans.append(c);
            index++;
        }
        return ans.toString();
    }
}

```

PREPARE

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]]`

Example 2:

Input: nums = []

Output: []

Example 3:

Input: nums = [0]

Output: []

Constraints:

- $0 \leq \text{nums.length} \leq 3000$
 - $-10^5 \leq \text{nums}[i] \leq 10^5$
-

```

//25 ms, faster than 46.90%
class Solution
{
    //3 pointer
    public List<List<Integer>> threeSum(int[] nums) // -4,-1,-1,0,1,2
    {
        List<List<Integer>> ans = new ArrayList<>();
        List<Integer> subList = new ArrayList<>();
        int len = nums.length;
        Arrays.sort(nums);

        for (int i = 0; i < len; i++)//a+b+c = 0
        {
            while(i != 0 && i < len && nums[i] == nums[i-1])//same number
            {
                i++;
            }
            if (i >= len-1) break; //out of bound
            int a = nums[i]; // first num
            int target = -1*a; //b+c = -a

            int lp = i+1;
            int hp = len-1;
            // System.out.println(" Traget = " + target);

            // -4 Traget = 4
            // -1 Traget = 1
            // -1 Traget = 0
            // 0 Traget = -1
            // 1 Traget = -2
            // 2 ?

            while (lp < hp)
            {
                if(nums[lp] + nums[hp] == target)//we got pair
                {
                    subList.add(nums[i]);
                    subList.add(nums[lp]);
                    subList.add(nums[hp]); //added 3 numbers in subList
                    ans.add(subList); //added in main list
                    subList = new ArrayList<>(); //new subList
                    lp++;
                    hp--;
                    //if previous num is same
                    while(lp < hp && nums[lp] == nums[lp-1])
                    lp++;
                    //if number is same..matlab process ho chuka h
                }
            }
        }
    }
}

```

```

        while(lp > hp && nums[hp] == nums[hp+1])
            hp--;
    } // -4, -1, -1, 0, 1, 2
    else if(nums[lp] + nums[hp] < target) // increase lp qki array sorted h
    {
        lp++;
        while(lp < hp && nums[lp] == nums[lp-1]) lp++; // to avoid same number
    }
    else if(nums[lp] + nums[hp] > target) // decrease hp qki array sorted h
    {
        hp--;
        while(lp < hp && nums[hp] == nums[hp+1]) hp--; // to avoid same number
    }
}

// ans.add(subList);
}
return ans;
}

}

```

16. 3Sum Closest ↗

Given an integer array `nums` of length `n` and an integer `target`, find three integers in `nums` such that the sum is closest to `target`.

Return *the sum of the three integers*.

You may assume that each input would have exactly one solution.

Example 1:

Input: nums = [-1,2,1,-4], target = 1

Output: 2

Explanation: The sum that is closest to the target is 2. (-1 + 2 + 1 = 2).

Example 2:

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

Output: 0

Constraints:

- $3 \leq \text{nums.length} \leq 1000$
 - $-1000 \leq \text{nums}[i] \leq 1000$
 - $-10^4 \leq \text{target} \leq 10^4$
-

```

//abs value coz if we take 0,1,3 then 0 is close to 1 0-1 = -1 where as 3-1
=2
class Solution {
    public int threeSumClosest(int[] nums, int target) {

        Arrays.sort(nums);
        int close = nums[0]+ nums[1] + nums[2];

        for(int i = 0; i < nums.length-2; i++) {
            int start = i+1, end = nums.length-1;

            while (start < end){
                int currSum = nums[i] + nums[start]+ nums[end];
                //corner case just to optimise..same he mil gaya to or q de
khe
                if (currSum == target){
                    close = currSum;
                    break;
                }
                else if (currSum < target) start++;
                else end--;
                if(Math.abs(currSum - target) < Math.abs(close - target)){
                    close = currSum;
                }
            }
        }
        return close;
    }
}

```

17. Letter Combinations of a Phone Number ↗

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

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



Example 1:

```
Input: digits = "23"
Output: ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"]
```

Example 2:

```
Input: digits = ""
Output: []
```

Example 3:

```
Input: digits = "2"
Output: ["a", "b", "c"]
```

Constraints:

- $0 \leq \text{digits.length} \leq 4$
- $\text{digits}[i]$ is a digit in the range $['2', '9']$.

```

class Solution {
    public List<String> letterCombinations(String digits) {

        List<String> list = new ArrayList<String>();
        if (digits == null || digits.length() == 0) return list;

        HashMap<Character, String> map = new HashMap<>();
        map.put('2', "abc"); map.put('3', "def"); map.put('4', "ghi"); map.
        put('5', "jkl");
        map.put('6', "mno"); map.put('7', "pqrs"); map.put('8', "tuv"); ma
        p.put('9', "wxyz");

        Queue<String> queue = new LinkedList<>();
        queue.add("");

        for (int i = 0; i < digits.length(); i++) { //23
            char c = digits.charAt(i); //2
            String smap = map.get(c); //abc
            int size = queue.size(); //"""

            while (size > 0) { //1
                String inQ = queue.poll(); //"""

                for (int k = 0; k < smap.length(); k++) { //abc
                    queue.add(inQ+smap.charAt(k)); //a b c
                }
                size--;
            }
        }

        while (!queue.isEmpty()){
            list.add(queue.poll());
        }

        return list;
    }
}

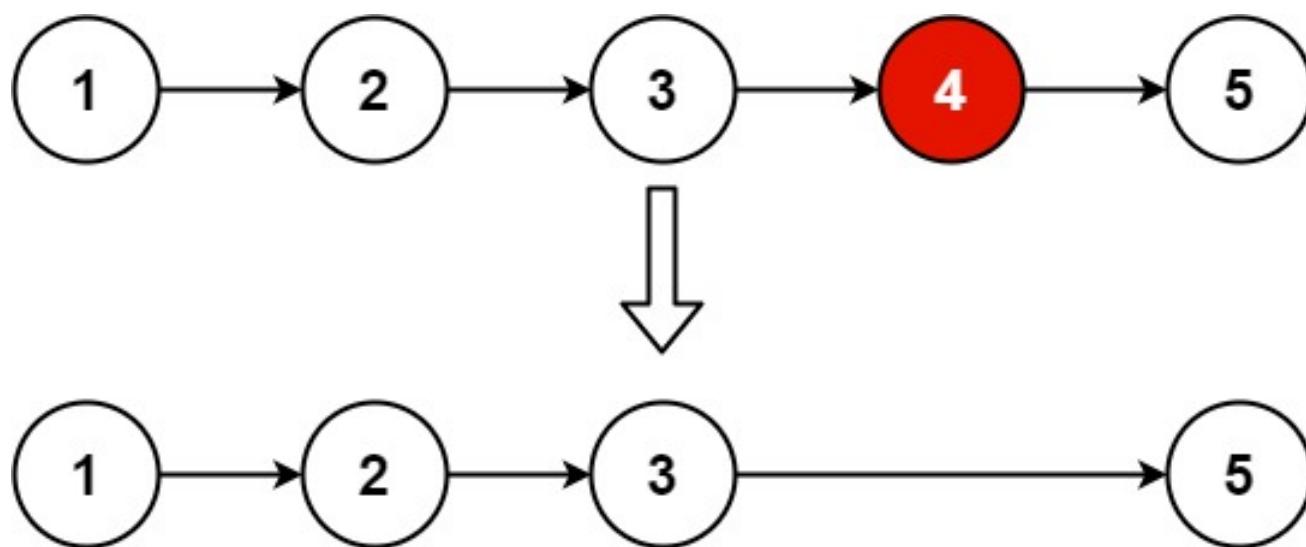
```

19. Remove Nth Node From End of List ↗



Given the head of a linked list, remove the n^{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)
    {
        ListNode fastPointer = head;
        ListNode slowPointer = head;

        while (n > 0 )
        {
            fastPointer = fastPointer.next;
            n--;
        }
        if (fastPointer == null)
        {
            head = head.next;
            return head;
        }
        while (fastPointer.next != null)
        {
            slowPointer = slowPointer.next;
            fastPointer = fastPointer.next;
        }
        //delete now
        ListNode tempNode = slowPointer.next;
        slowPointer.next = tempNode.next;
        tempNode.next = null;
        return head;
    }
}

```

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.

Example 1:

Input: s = "()"

Output: true

Example 2:

Input: s = "()[]{}"

Output: true

Example 3:

Input: s = "()"

Output: false

Example 4:

Input: s = "([)]"

Output: false

Example 5:

Input: s = "{[()]}"

Output: true

Constraints:

- $1 \leq s.length \leq 10^4$
- s consists of parentheses only '()[]{}' .

```

class Solution {
    public boolean isValid(String s) {

        char[] chars = s.toCharArray();
        Stack<Character> st = new Stack<>();

        for (char c : chars)
        {
            if (c == '(' || c == '[' || c == '{') st.push(c);
            //check for empty everytime
            else if (c == ')' && !st.isEmpty() && st.peek() == '(') st.pop();
            else if (c == ']' && !st.isEmpty() && st.peek() == '[') st.pop();
            else if (c == '}' && !st.isEmpty() && st.peek() == '{') st.pop();

            else return false; //nothing was pushed or popped
        }

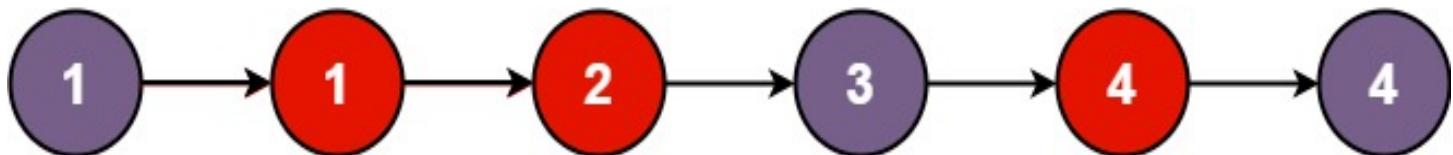
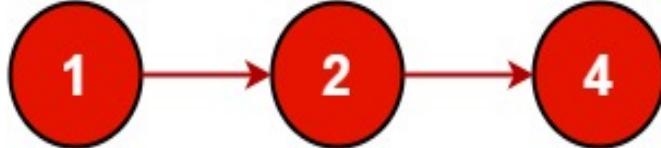
        return st.isEmpty() ? true : false; //for single input '[' ']'
    }
}

```

21. Merge Two Sorted Lists ↗

Merge two sorted linked lists and return it as a **sorted** list. The list should be made by splicing together the nodes of the first two lists.

Example 1:



Input: l1 = [1,2,4], l2 = [1,3,4]

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

Example 2:

Input: l1 = [], l2 = []

Output: []

Example 3:

Input: l1 = [], l2 = [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 l1 and l2 are sorted in **non-decreasing** order.

```
class Solution {
    public ListNode mergeTwoLists(ListNode l1, ListNode l2)
    {
        ListNode currNode = null;
        ListNode head = null;

        if (l1 == null && l2 == null)
        {
            return null;
        }
        else if(l1 != null && l2 == null)
        {
            //return l1 coz l2 is empty
            return l1;
        }
        else if (l2 != null && l1 == null)
        {
            //return l2 coz l1 is empty
            return l2;
        }

        else if (l1 != null && l2 != null)
        {
            if (l1.val < l2.val)
            {
                currNode = l1;
                l1 = l1.next;
            }
            else
            {
                currNode = l2;
                l2 = l2.next;
            }
            head = currNode;
        }

        while (l1 != null && l2 != null)
        {
            if (l1.val < l2.val)
            {
                currNode.next = l1;
                l1 = l1.next;
                //currNode = currNode.next;
            }
            else
            {

```

```
        currNode.next = l2;
        l2 = l2.next;

    }
    currNode = currNode.next;
}
if (l1 == null)
{
    currNode.next = l2;
    /*while (l2 != null)
    {
        currNode.next = l2;
        l2 = l2.next;
        currNode = currNode.next;
    }*/
}
else if (l2 == null)
{
    currNode.next = l1;
    /*while (l1 != null)
    {
        currNode.next = l1;
        l1 = l1.next;
        currNode = currNode.next;
    } */
}
return head;

}
}
```

Practice

```
/**  
 * 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 l1, ListNode l2) {  
  
        //corner cases  
        if (l1 == null && l2 == null) return null;  
        else if (l1 == null) return l2;  
        else if (l2 == null) return l1;  
  
        ListNode ans = new ListNode();  
        ListNode ansHead = ans;  
        ListNode c11 = l1, c12 = l2;  
  
        while (c11 != null && c12 != null){  
  
            if (c11.val < c12.val){  
  
                ans.next = c11;  
                ans = ans.next;  
                c11 = c11.next;  
  
            } else {  
                ans.next = c12;  
                ans = ans.next;  
                c12 = c12.next;  
            }  
        }  
        if (c11 == null){  
            ans.next = c12;  
        } else {  
            ans.next = c11;  
        }  
        return ansHead.next;  
    }  
}
```

23. Merge k Sorted Lists ↗

You are given an array of `k` linked-lists `lists`, each linked-list is sorted in ascending order.

Merge all the linked-lists into one sorted linked-list and return it.

Example 1:

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

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

Explanation: The linked-lists are:

```
[  
    1->4->5,  
    1->3->4,  
    2->6
```

merging them into one sorted list:
`1->1->2->3->4->4->5->6`

Example 2:

Input: `lists = []`

Output: `[]`

Example 3:

Input: `lists = [[]]`

Output: `[]`

Constraints:

- `k == lists.length`
- `0 <= k <= 10^4`
- `0 <= lists[i].length <= 500`
- `-10^4 <= lists[i][j] <= 10^4`
- `lists[i]` is sorted in **ascending order**.
- The sum of `lists[i].length` won't exceed `10^4`.

```

//we can also put head.next while taking out the head...this way we can reduce heap size
//Runtime : O(n*mlog(n*m))
//space : O(n*m)
//n = number of list, m = max number of nodes

//second Approach: mentioned in Kevin's comments
//TC: O(nlogk)
class Solution {
    public ListNode mergeKLists(ListNode[] lists) {

        PriorityQueue<ListNode> pq = new PriorityQueue<>((a,b) -> a.val-b.val);

        for (ListNode l : lists) { //since we got all heads
            if (l != null) pq.add(l); //add all head,,null check imp

        }
        ListNode ans = new ListNode(-1);
        ListNode dummmHead = ans;//dummyhead
        while (!pq.isEmpty()){
            ListNode node = pq.remove();
            ans.next = node;
            ans =ans.next;
            if (node.next != null) pq.add(node.next); //jb bhi koi node remove ho rahi h uske next node ko priority queue me daal do,,null check imp
        }

        return dummmHead.next;
    }
}

```

```

//we can also put head.next while taking out the head...this way we can reduce heap size
//Runtime : O(nmlog(nm)) //space : O(n*m) //n = number of list, m = max number of nodes

```

```

class Solution {
    public ListNode mergeKLists(ListNode[] lists) {

        PriorityQueue<Integer> pq = new PriorityQueue<>();

        for (ListNode l : lists) { //since we got all heads
            while (l != null) {
                pq.add(l.val);
                l = l.next;
            }
        }
        ListNode ans = new ListNode(-1);
        ListNode dummmHead = ans;//dummyhead
        while (!pq.isEmpty()){
            ans.next = new ListNode(pq.remove());
            ans =ans.next;
        }

        return dummmHead.next;

    }
}

```

/**

- 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; }
- }
- //102 ms, faster than 14.76% class Solution { public ListNode mergeKLists(ListNode[] lists)
{

```
if (lists.length == 0) return null;
ListNode headFirst = lists[0];

if (lists.length == 1 && headFirst == null) return null;

ArrayList<Integer> next = null;
for (int i = 1; i < lists.length; i++)
{
    ListNode nextFirst = lists[i];
    headFirst = mergeTwoLists (headFirst, nextFirst);
}
return headFirst;
```

```
} public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
```

```
ListNode currNode = null;
ListNode head = null;

if (l1 == null && l2 ==null)
{
    return null;
}
else if(l1 != null && l2 == null)
{
    //return l1 coz l2 is empty
    return l1;
}
else if (l2 != null && l1 == null)
{
    //return l2 coz l1 is empty
    return l2;
}

else if (l1 != null && l2 != null)
{
    if (l1.val < l2.val)
    {
        currNode = l1;
        l1 = l1.next;
    }
    else
    {
        currNode = l2;
        l2 = l2.next;
    }
    head = currNode;
}

while (l1 != null && l2 != null)
{
    if (l1.val < l2.val)
    {
        currNode.next = l1;
        l1 = l1.next;
        //currNode = currNode.next;
    }
    else
    {
        currNode.next = l2;
        l2 = l2.next;
    }
}
```

```

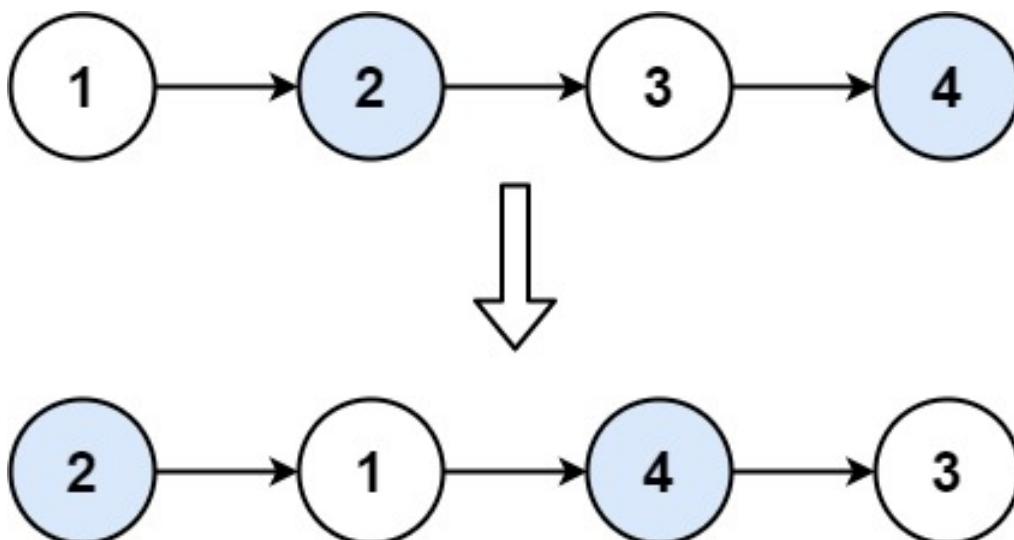
        }
        currNode = currNode.next;
    }
    if (l1 == null)
    {
        currNode.next = l2;
    }
    else if (l2 == null)
    {
        currNode.next = l1;
    }
    return head;
}
}
```

```

## 24. Swap Nodes in Pairs ↗

Given a linked list, swap every two adjacent nodes and return its head. You must solve the problem without modifying the values in the list's nodes (i.e., only nodes themselves may be changed.)

### Example 1:



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

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

### Example 2:

**Input:** head = []

**Output:** []

### Example 3:

**Input:** head = [1]

**Output:** [1]

### Constraints:

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

```

/**
 * 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 swapPairs(ListNode head) //1--2--3--4 2--1--3--4 2--1-
-4--3
 {
 //corner cases
 if (head == null) return null;
 if (head.next == null) return head; // [1]

 ListNode newHead = head.next;
 ListNode prev = new ListNode(0); // prev is dummyhead jo head ko poi
nt kar ra
 prev.next = head;
 ListNode currNode = prev; // start hum dummy se karege islia curr no
de ko dummy pr rakha h

 while (currNode.next != null && currNode.next.next != null)
 {
 ListNode fn = currNode.next; // 1 first node
 ListNode sn = currNode.next.next; // 2 second node

 prev.next = sn; // dummy--2

 fn.next = sn.next; // 1--3
 sn.next = fn; // 2--1
 prev = fn; // prev = 1
 currNode = prev; // curr = prev = 1
 }
 return newHead;
 }
}

```



Given an integer array `nums` sorted in **non-decreasing order**, remove the duplicates **in-place** ([https://en.wikipedia.org/wiki/In-place\\_algorithm](https://en.wikipedia.org/wiki/In-place_algorithm)) such that each unique element appears only **once**. The **relative order** of the elements should be kept the **same**.

Since it is impossible to change the length of the array in some languages, you must instead have the result be placed in the **first part** of the array `nums`. More formally, if there are `k` elements after removing the duplicates, then the first `k` elements of `nums` should hold the final result. It does not matter what you leave beyond the first `k` elements.

Return `k` *after placing the final result in the first `k` slots of `nums`*.

Do **not** allocate extra space for another array. You must do this by **modifying the input array in-place** ([https://en.wikipedia.org/wiki/In-place\\_algorithm](https://en.wikipedia.org/wiki/In-place_algorithm)) with O(1) extra memory.

### Custom Judge:

The judge will test your solution with the following code:

```
int[] nums = [...]; // Input array
int[] expectedNums = [...]; // The expected answer with correct length

int k = removeDuplicates(nums); // Calls your implementation

assert k == expectedNums.length;
for (int i = 0; i < k; i++) {
 assert nums[i] == expectedNums[i];
}
```

If all assertions pass, then your solution will be **accepted**.

### Example 1:

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

**Output:** `2, nums = [1,2,_]`

**Explanation:** Your function should return `k = 2`, with the first two elements of `nums` being `1` and `2`. It does not matter what you leave beyond the returned `k` (hence they are underlined).

### Example 2:

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

**Output:** 5, nums = [0,1,2,3,4,\_,\_,\_,\_,\_]

**Explanation:** Your function should return k = 5, with the first five elements

It does not matter what you leave beyond the returned k (hence they are under

### Constraints:

- $0 \leq \text{nums.length} \leq 3 * 10^4$
- $-100 \leq \text{nums}[i] \leq 100$
- nums is sorted in **non-decreasing** order.

```
class Solution {
 public int removeDuplicates(int[] nums) {

 if (nums.length == 0) return 0;
 int k = 0; //maintain non duplicate till k

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

 if (nums[k] != nums[i]) {
 nums[k+1] = nums[i]; //store in k+1 to make k+1 index unique
 k++; //till k it is non duplicate
 }
 }
 return k+1;
 }
}
```

```

class Solution {
 public int removeDuplicates(int[] nums) {

 int l = nums.length;
 int c = l;
 int temp = 1;

 for(int i = 0; i < l-1; i++)
 {
 if(nums[i] == nums[i+1])
 {
 c--;
 }
 else
 {
 nums[temp] = nums[i+1];
 temp++;
 }
 }
 return c;
 }
}

```

## Practice

```

class Solution {
 public int removeDuplicates(int[] nums) {

 if (nums == null || nums.length ==0) return 0;

 //track the place to change, counter stores number to be compared
 int track = 1, counter = nums[0];
 for (int i = 1; i < nums.length; i++)
 {
 if (nums[i] != counter)
 {
 counter = nums[i];
 nums[track] = nums[i]; // in place modification
 track++;
 }
 }
 return track;
 }
}

```

## 27. Remove Element ↗



Given an integer array `nums` and an integer `val`, remove all occurrences of `val` in `nums` **in-place** ([https://en.wikipedia.org/wiki/In-place\\_algorithm](https://en.wikipedia.org/wiki/In-place_algorithm)). The relative order of the elements may be changed.

Since it is impossible to change the length of the array in some languages, you must instead have the result be placed in the **first part** of the array `nums`. More formally, if there are `k` elements after removing the duplicates, then the first `k` elements of `nums` should hold the final result. It does not matter what you leave beyond the first `k` elements.

Return `k` *after placing the final result in the first `k` slots of `nums`*.

Do **not** allocate extra space for another array. You must do this by **modifying the input array in-place** ([https://en.wikipedia.org/wiki/In-place\\_algorithm](https://en.wikipedia.org/wiki/In-place_algorithm)) with O(1) extra memory.

### Custom Judge:

The judge will test your solution with the following code:

```
int[] nums = [...]; // Input array
int val = ...; // Value to remove
int[] expectedNums = [...]; // The expected answer with correct length.
 // It is sorted with no values equaling val.

int k = removeElement(nums, val); // Calls your implementation

assert k == expectedNums.length;
sort(nums, 0, k); // Sort the first k elements of nums
for (int i = 0; i < actualLength; i++) {
 assert nums[i] == expectedNums[i];
}
```

If all assertions pass, then your solution will be **accepted**.

### Example 1:

**Input:** nums = [3,2,2,3], val = 3

**Output:** 2, nums = [2,2,\_,\_]

**Explanation:** Your function should return k = 2, with the first two elements c

It does not matter what you leave beyond the returned k (hence they are under

## Example 2:

**Input:** nums = [0,1,2,2,3,0,4,2], val = 2

**Output:** 5, nums = [0,1,4,0,3,\_,\_,\_]

**Explanation:** Your function should return k = 5, with the first five elements

Note that the five elements can be returned in any order.

It does not matter what you leave beyond the returned k (hence they are under

## Constraints:

- $0 \leq \text{nums.length} \leq 100$
- $0 \leq \text{nums}[i] \leq 50$
- $0 \leq \text{val} \leq 100$

---

```
class Solution { public int removeElement(int[] nums, int val) {
```

```
int l = nums.length -1;

for (int i=0; i <= l;)
{
 int temp;
 if (nums[i] == val)
 {
 nums[i] = nums[i] + nums[l];
 nums[l] = nums[i] - nums[l];
 nums[i] = nums[i] - nums[l];
 l--;
 }
 if (nums[i] != val)
 {
 i++;
 }
}
return l+1;

}
```

## 28. Implement strStr() ↗

Implement strStr() (<http://www.cplusplus.com/reference/cstring/strstr/>).

Return the index of the first occurrence of needle in haystack, or `-1` if `needle` is not part of `haystack`.

### Clarification:

What should we return when `needle` is an empty string? This is a great question to ask during an interview.

For the purpose of this problem, we will return 0 when `needle` is an empty string. This is consistent to C's `strstr()` (<http://www.cplusplus.com/reference/cstring/strstr/>) and Java's `indexOf()` ([https://docs.oracle.com/javase/7/docs/api/java/lang/String.html#indexOf\(java.lang.String\)](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html#indexOf(java.lang.String))).

### Example 1:

**Input:** haystack = "hello", needle = "ll"

**Output:** 2

### Example 2:

**Input:** haystack = "aaaaa", needle = "bba"

**Output:** -1

### Example 3:

**Input:** haystack = "", needle = ""

**Output:** 0

### Constraints:

- $0 \leq \text{haystack.length}, \text{needle.length} \leq 5 * 10^4$
  - haystack and needle consist of only lower-case English characters.
-

```
class Solution {
 public int strStr(String haystack, String needle) {

 if (needle.length() == 0) return 0;

 if (haystack.length() < needle.length()) return -1;

 int lenH = haystack.length(), lenN = needle.length();

 for (int i = 0; i <= (lenH-lenN); i++)
 {
 if(isSame(haystack,needle, i, i+lenN-1, 0, lenN-1))
 {
 return i;
 }
 }
 return -1;
 }

 private boolean isSame(String hey, String needle, int start_h, int end_h, int start_n, int end_n) //2 pointer
 {
 while (start_n <= end_n) {

 if (hey.charAt(start_h) != needle.charAt(start_n) || hey.charAt(end_h) != needle.charAt(end_n)) return false;

 start_h++; end_h--;
 start_n++; end_n--;

 }

 return true;
 }
}
```

```

class Solution {
 public int strStr(String haystack, String needle) {

 if (haystack.length() == 0 && needle.length() == 0)
 {
 return 0;
 }
 else if (needle.length() != 0 && haystack.length() == 0) return -1;
 else if (needle.length() == 0 && haystack.length() != 0) return 0;

 int s = 0, e = haystack.length() -1;
 char needleStart = needle.charAt(0);
 char needleEnd = needle.charAt(needle.length()-1);
 int start = -1;

 //if (s-e+1 < needle.length()) return -1;
 while (e-s+1 >= needle.length() && s <= e)
 {
 int i = 0;
 if (haystack.charAt(s) == needleStart)
 {
 start = s;
 while (i < needle.length() && haystack.charAt(s) == needle.charAt(i))
 {
 s++; i++;
 }
 if (i == needle.length()) break;
 else start = -1;
 }s = s-i+1;
 }
 return (start != -1)?start:-1;
 }
}

```

## 31. Next Permutation ↗



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

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

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

### Example 1:

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

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

### Example 2:

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

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

### Example 3:

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

```
Output: [1,5,1]
```

### Example 4:

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

```
Output: [1]
```

### Constraints:

- $1 \leq \text{nums.length} \leq 100$
- $0 \leq \text{nums}[i] \leq 100$

```

/*
traverse from back [158476531]
Find number which is less than i+1 (N1) [N1 = 4]
Find the number just greater(N2) than above number(N1) and swap [N2 = 5][15
8576431]
Reverse all number after N2[158513467]

```

If you dont find N1 that means array is sorted in decending order and then simply print from back

\*/

```

class Solution {
 public void nextPermutation(int[] nums) {
 int index = -1;
 for (int i = nums.length-1; i >0; i--){
 if (nums[i] > nums[i-1]){
 index = i-1;
 break;
 }
 }
 if (index == -1){
 reverse(nums, 0, nums.length-1);
 }
 else {
 //find number just greater
 int num = Integer.MAX_VALUE;
 int numIndex = -1;
 for (int i = index +1; i < nums.length; i++){
 if (nums[i] > nums[index] && nums[i] < num) {
 numIndex = i;
 }
 }
 //swap
 int temp = nums[index];
 nums[index] = nums[numIndex];
 nums[numIndex] = temp;

 //reverse from index+1
 reverse(nums, index+1, nums.length-1);
 }
 }

 private void reverse(int[] nums, int s, int e) {
 while (s < e) {
 int temp = nums[s];
 nums[s] = nums[e];
 nums[e] = temp;
 s++;
 e--;
 }
 }
}
```

```

 nums[s] = nums[e];
 nums[e] = temp;
 s++;
 e--;
 }
}
}

```

## 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 <= k < nums.length`) such that the resulting array is `[nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]]` (**0-indexed**). For example, `[0,1,2,4,5,6,7]` might be rotated at pivot index 3 and become `[4,5,6,7,0,1,2]`.

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

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

### Example 1:

```

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

```

### Example 2:

```

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

```

### Example 3:

```

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

```

### Constraints:

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

```

class Solution {
 public static int search(int[] nums, int target) //4,5,6,7,0,1,2
 {
 int ans = -1;
 int length = nums.length-1;
 int index = findMid(nums);

 if (nums[index] == target)
 {
 return index;
 }
 else
 {
 if (target == nums[length])
 {
 return length;
 }
 else if (target > nums[length])
 {
 ans = binarySearch(nums, 0, index -1, target);
 }
 else if (target < nums[length])
 {
 ans = binarySearch(nums, index + 1, length, target);
 }
 }
 return ans;
 }

 //returns minimum number index
 public static int findMid(int[] num)
 {
 int lengthOfArray = num.length;
 int l = 0, h = lengthOfArray-1, mid;
 while (l <= h)
 {
 mid = l + (h - l)/2;
 int next = (mid + 1) % lengthOfArray;
 int prev = (mid - 1 + lengthOfArray) % lengthOfArray;
 //minimum element will be small from its left and right
 if (num[mid] <= num[prev] && num[mid] <= num[next])
 {
 return mid;
 }
 }
 //means left side is not sorted, so we have to move left side a
 }
}

```

```

rray
 else if (num[mid] < num[h])
 {
 h = mid-1;
 }
 else
 {
 l = mid + 1;
 }
}
return -1;
}

public static int binarySearch(int[] arr, int l, int h, int find)
{
 int mid;
 while (l <= h)
 {
 mid = l + (h - l)/2;

 if (arr[mid] == find)
 {
 return mid;
 }
 else if (arr[mid] > find)
 {
 h = mid -1;
 }
 else if (arr[mid] < find)
 {
 l = mid +1;
 }
 }
 return -1;
}
}

```

## 34. Find First and Last Position of Element in Sorted Array ↗

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

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

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

### Example 1:

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

**Output:** `[3,4]`

### Example 2:

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

**Output:** `[-1,-1]`

### Example 3:

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

**Output:** `[-1,-1]`

### Constraints:

- $0 \leq \text{nums.length} \leq 10^5$
- $-10^9 \leq \text{nums}[i] \leq 10^9$
- `nums` is a non-decreasing array.
- $-10^9 \leq \text{target} \leq 10^9$

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

 int firstIndex = binarySearchReturnFirstIndex(nums, target);
 int lastIndex = binarySearchReturnLastIndex(nums, target);
 int[] result = {firstIndex, lastIndex};

 return result;
 }
 //get first occurrence
 public static int binarySearchReturnFirstIndex(int arr[], int target)
 {
 int index = -1, mid;
 int l = 0, h = arr.length -1;

 while (l <= h)
 {
 mid = l + (h - l)/2;

 if (arr[mid] == target)
 {
 index = mid;
 h = mid -1;
 }
 else if (target > arr[mid])
 {
 l = mid +1;
 }
 else
 {
 h = mid -1;
 }
 }

 return index;
 }

 //get last occurrence
 public static int binarySearchReturnLastIndex(int arr[], int target)
 {
 int index = -1, mid;
 int l = 0, h = arr.length -1;

 while (l <= h)
 {
```

```

 mid = l + (h - 1)/2;

 if (arr[mid] == target)
 {
 index = mid;
 l = mid +1;
 }
 else if (target > arr[mid])
 {
 l = mid +1;
 }
 else
 {
 h = mid -1;
 }
 }

 return index;
}
}

```

## 36. Valid Sudoku ↗

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

1. Each row must contain the digits 1-9 without repetition.
2. Each column must contain the digits 1-9 without repetition.
3. Each of the nine  $3 \times 3$  sub-boxes of the grid must contain the digits 1-9 without repetition.

**Note:**

- A Sudoku board (partially filled) could be valid but is not necessarily solvable.
- Only the filled cells need to be validated according to the mentioned rules.

**Example 1:**

|   |   |   |   |   |   |   |   |  |
|---|---|---|---|---|---|---|---|--|
| 5 | 3 |   | 7 |   |   |   |   |  |
| 6 |   |   | 1 | 9 | 5 |   |   |  |
|   | 9 | 8 |   |   |   | 6 |   |  |
| 8 |   |   | 6 |   |   |   | 3 |  |
| 4 |   | 8 |   | 3 |   |   | 1 |  |
| 7 |   |   | 2 |   |   |   | 6 |  |
|   | 6 |   |   |   | 2 | 8 |   |  |
|   |   | 4 | 1 | 9 |   |   | 5 |  |
|   |   |   | 8 |   |   | 7 | 9 |  |

**Input:** board =

```
[["5","3",".",".","7",".",".",".","."]
 ,["6",".",".","1","9","5",".",".","."]
 ,[".","9","8",".",".",".",".","6","."]
 ,["8",".",".",".","6",".",".",".","3"]
 ,["4",".",".","8",".","3",".",".","1"]
 ,["7",".",".",".","2",".",".",".","6"]
 ,[".","6",".",".",".","2","8","."]
 ,[".",".",".","4","1","9",".",".","5"]
 ,[".",".",".","8",".",".","7","9"]]
```

**Output:** true

## Example 2:

**Input:** board =

```
[["8","3",".",".","7",".",".",".","."]
 ,["6",".",".","1","9","5",".",".","."]
 ,[".","9","8",".",".",".",".","6","."]
 ,["8",".",".",".","6",".",".",".","3"]
 ,["4",".",".","8",".","3",".",".","1"]
 ,["7",".",".",".","2",".",".",".","6"]
 ,[".","6",".",".",".","2","8","."]
 ,[".",".",".","4","1","9",".",".","5"]
 ,[".",".",".","8",".",".","7","9"]]
```

**Output:** false

**Explanation:** Same as Example 1, except with the 5 in the top left corner being

## Constraints:

- board.length == 9

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

```

//**boxNumber = (i/3)*3 + j/3;
class Solution {
 public boolean isValidSudoku(char[][] board) {
 HashSet<String> set = new HashSet<>();

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

 for (int j = 0; j < board[0].length; j++) {

 if (board[i][j] != '.'){

 if (!set.add("r"+ i + " "+board[i][j])) return false;
 if (!set.add("c"+ j + " "+ board[i][j])) return false;
 int boxNumber = (i/3)*3 + j/3;
 if (!set.add("b"+ boxNumber +" "+ board[i][j])) return
false;

 }
 }
 }
 return true;
 }
}

```

## 38. Count and Say ↗

The **count-and-say** sequence is a sequence of digit strings defined by the recursive formula:

- `countAndSay(1) = "1"`
- `countAndSay(n)` is the way you would "say" the digit string from `countAndSay(n-1)` , which is then converted into a different digit string.

To determine how you "say" a digit string, split it into the **minimal** number of groups so that each group is a contiguous section all of the **same character**. Then for each group, say the number of characters, then say the character. To convert the saying into a digit string, replace the counts with a number and concatenate every saying.

For example, the saying and conversion for digit string "3322251" :

"3322251"

two 3's, three 2's, one 5, and one 1

2 3 + 3 2 + 1 5 + 1 1

"23321511"

Given a positive integer  $n$ , return the  $n^{\text{th}}$  term of the **count-and-say sequence**.

### Example 1:

**Input:**  $n = 1$

**Output:** "1"

**Explanation:** This is the base case.

### Example 2:

**Input:**  $n = 4$

**Output:** "1211"

**Explanation:**

countAndSay(1) = "1"

countAndSay(2) = say "1" = one 1 = "11"

countAndSay(3) = say "11" = two 1's = "21"

countAndSay(4) = say "21" = one 2 + one 1 = "12" + "11" = "1211"

### Constraints:

- $1 \leq n \leq 30$

for k ander he char nikalo and ye s=dhyaan rakho ki last character ka count hum miss na kare

```

class Solution {
 public String countAndSay(int n) {

 if (n == 1) return "1";

 String rec = countAndSay(n-1);
 StringBuffer val = new StringBuffer();
 int counter = 0;

 for (int i = 0; i < rec.length(); i++) {
 counter++;
 if ((i == rec.length()-1) || rec.charAt(i) != rec.charAt(i+1))
{
 val.append(counter).append(rec.charAt(i));
 counter = 0;
 }
 }
 return val.toString();
 }
}

```

## 39. Combination Sum ↗

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

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

It is **guaranteed** that the number of unique combinations that sum up to `target` is less than 150 combinations for the given input.

**Example 1:**

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

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

**Explanation:**

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

These are the only two combinations.

### Example 2:

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

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

### Example 3:

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

**Output:** []

### Example 4:

**Input:** candidates = [1], target = 1

**Output:** [[1]]

### Example 5:

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

**Output:** [[1,1]]

### Constraints:

- $1 \leq \text{candidates.length} \leq 30$
- $1 \leq \text{candidates}[i] \leq 200$
- All elements of candidates are **distinct**.
- $1 \leq \text{target} \leq 500$

```

class Solution {
 public List<List<Integer>> combinationSum(int[] candidates, int target)
 {
 List<List<Integer>> combs = new ArrayList<>();
 generateCombinations(candidates, 0, target, new ArrayList(), comb
s);
 return combs;
 }
 private void generateCombinations(int[] nums, int start, int target, Ar
rayList<Integer> currList, List<List<Integer>> combs)
 {
 if (target == 0) {
 combs.add(new ArrayList<>(currList));
 }
 if (target < 0) return;

 for (int i = start; i < nums.length; i++) {
 currList.add(nums[i]);
 generateCombinations(nums, i, target-nums[i], currList, combs);
 currList.remove(currList.size()-1);
 }
 }
}

```

## 40. Combination Sum II ↗

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

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

**Note:** The solution set must not contain duplicate combinations.

**Example 1:**

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

**Output:**

```
[
[1,1,6],
[1,2,5],
[1,7],
[2,6]
]
```

### Example 2:

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

**Output:**

```
[
[1,2,2],
[5]
]
```

### Constraints:

- $1 \leq \text{candidates.length} \leq 100$
- $1 \leq \text{candidates}[i] \leq 50$
- $1 \leq \text{target} \leq 30$

---

**Arrays.sort and if ( $i > \text{start} \&& \text{nums}[i] == \text{nums}[i-1]$ ) continue; reduced time complexity**

```

class Solution {
 public List<List<Integer>> combinationSum2(int[] candidates, int target) {
 Arrays.sort(candidates);
 List<List<Integer>> res = new ArrayList<>();
 Map<Integer, Boolean> memo = new HashMap<>();
 combinationSum(0, candidates, new ArrayList<>(), target, res);
 return res;
 }

 private void combinationSum(int start, int[] nums, ArrayList<Integer> curr, int target, List<List<Integer>> res) {
 if (target == 0) {
 if (!res.contains(curr))
 res.add(new ArrayList<Integer>(curr));
 return;
 }
 if (target < 0) return;

 for (int i = start; i < nums.length; i++) {
 if (i > start && nums[i] == nums[i-1]) continue;
 curr.add(nums[i]);
 combinationSum(i+1, nums, curr, target - nums[i], res);
 curr.remove(curr.size()-1);
 }
 }
}

```

## 41. First Missing Positive ↗

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

You must implement an algorithm that runs in  $O(n)$  time and uses constant extra space.

### Example 1:

**Input:** `nums = [1,2,0]`

**Output:** 3

## **Example 2:**

**Input:** nums = [3,4,-1,1]

**Output:** 2

## **Example 3:**

**Input:** nums = [7,8,9,11,12]

**Output:** 1

## **Constraints:**

- $1 \leq \text{nums.length} \leq 5 * 10^5$
  - $-2^{31} \leq \text{nums}[i] \leq 2^{31} - 1$
-

```

class Solution {
 public int firstMissingPositive(int[] nums) { // [3,4,-1,1]

 int l = nums.length;
 for (int i = 0; i < l;)
 {
 if (nums[i] <= l && nums[i] > 0 && nums[i]-1 != i)
 {
 // System.out.println(Arrays.toString(nums));

 int temp = nums[i];
 nums[i] = nums[temp-1]; // can use nums[i] or temp
 nums[temp-1] = temp; // coz above we have changed nums[i] ka
value

 if (i+1 == nums[i] || temp == nums[i]) i++;//correct elemen
t aa gaya h is index pr ab aage badho [1,1]
 // if (temp == nums[i]) i++;

 }
 else
 {
 i++;
 }
 }
 for (int i = 0; i < l; i++)
 {
 if (i+1 != nums[i]) return i+1;
 }
 return l+1;
 }
}

```

## 42. Trapping Rain Water ↗

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

**Example 1:**



**Input:** height = [0,1,0,2,1,0,1,3,2,1,2,1]

**Output:** 6

**Explanation:** The above elevation map (black section) is represented by array

### Example 2:

**Input:** height = [4,2,0,3,2,5]

**Output:** 9

### Constraints:

- $n == \text{height.length}$
- $1 \leq n \leq 2 * 10^4$
- $0 \leq \text{height}[i] \leq 10^5$

```
class Solution {
 public int trap(int[] height)
 {
 int water = 0;
 int left[] = maxLeft(height);
 int right[] = maxRight(height);

 for (int i = 0; i < height.length; i++)
 {
 int minHt = Math.min(left[i], right[i]);

 if (minHt > height[i]) //building ki ht k baad jo bacha h utna
water
 {
 water += minHt - height[i];
 }
 }

 return water;
 }
 //left max
 public int[] maxLeft(int[] height)
 {
 int left[] = new int[height.length];
 int maxHt = 0;
 for (int i = 0; i < height.length; i++)
 {
 left[i] = maxHt;
 maxHt = Math.max(maxHt, height[i]);
 }
 return left;
 }
 //right max
 public int[] maxRight(int[] height)
 {
 int right[] = new int[height.length];
 int maxHt = 0;
 for (int i = height.length-1; i >= 0 ; i--)
 {
 right[i] = maxHt;
 maxHt = Math.max(maxHt, height[i]);
 }
 return right;
 }
}
```

# 45. Jump Game II ↗



Given an array of non-negative integers `nums`, you are initially positioned at the first index of the array.

Each element in the array represents your maximum jump length at that position.

Your goal is to reach the last index in the minimum number of jumps.

You can assume that you can always reach the last index.

## Example 1:

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

**Output:** 2

**Explanation:** The minimum number of jumps to reach the last index is 2. Jump 1

## Example 2:

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

**Output:** 2

## Constraints:

- $1 \leq \text{nums.length} \leq 10^4$
- $0 \leq \text{nums}[i] \leq 1000$

```

//bade problem last me hai
class Solution {
 public int jump(int[] nums)
 {
 int length = nums.length-1;
 Integer dp[] = new Integer[nums.length];//Integer coz we have to store null

 dp[length] = 1; //last se jane ka 1 rasta vo kahe na jae

 for (int i = length-1; i >= 0; i--) //second last index
 {
 int j = 1;
 while (nums[i] != 0 && j <= nums[i] && i+j < dp.length)
 {
 if (dp[i+j] != null)//agar dp[i+j] null hai tb +1 ni kar skte
 {
 int steps = (int)dp[i+j] +1;
 //agr dp[i] null hai tb compare ni kar skte, seedha steps daal denge
 if (dp[i] != null) dp[i] = Math.min(steps, dp[i]);
 else dp[i] = steps;
 }
 j++;
 }
 }
 return dp[0] - 1;
 }
}

```

## 46. Permutations ↗

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

### Example 1:

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

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

## Example 2:

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

## Example 3:

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

## Constraints:

- $1 \leq \text{nums.length} \leq 6$
- $-10 \leq \text{nums}[i] \leq 10$
- All the integers of nums are **unique**.

```
class Solution {
 public List<List<Integer>> permute(int[] nums) {
 List<List<Integer>> list = new ArrayList<>();
 // Arrays.sort(nums); // not necessary
 backtrack(list, new ArrayList<>(), nums);
 return list;
 }

 private void backtrack(List<List<Integer>> list, List<Integer> tempList, int [] nums){
 if(tempList.size() == nums.length){
 list.add(new ArrayList<>(tempList));
 } else{
 for(int i = 0; i < nums.length; i++){
 if(tempList.contains(nums[i])) continue; // element already exists, skip
 tempList.add(nums[i]);
 backtrack(list, tempList, nums);
 tempList.remove(tempList.size() - 1);
 }
 }
 }
}
```

```

class Solution {
 public List<List<Integer>> permute(int[] nums)//Recursion and backtracking
 {
 List<List<Integer>> list = new ArrayList<>();
 int[] fill = new int[nums.length];
 Arrays.fill(fill, Integer.MIN_VALUE);

 permuteHelp(nums, 0, fill, list);
 return list;
 }
 // [1,2,3]
 public void permuteHelp(int[] nums, int s, int[] fill, List<List<Integer>> list)
 {

 if (s > nums.length-1)
 {
 ArrayList<Integer> al = new ArrayList<>();
 for (int i = 0; i < fill.length; i++)
 {
 //System.out.print(fill[i]);
 al.add(fill[i]);
 }
 list.add(al);
 return;
 }
 //if (list.size() == nums.length*2) return;
 for(int i = 0; i < nums.length; i++)//isme recursion hoga...
 {
 if (fill[i] == Integer.MIN_VALUE)//pahle se fill ni h to fill k
 aro
 {
 fill[i]= nums[s];
 permuteHelp(nums, s+1, fill, list);//fill karke bej do aage
 fill[i] = Integer.MIN_VALUE;
 }
 }
 }
}

```

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

### Example 1:

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

**Output:**

```
[[1,1,2],
 [1,2,1],
 [2,1,1]]
```

### Example 2:

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

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

### Constraints:

- `1 <= nums.length <= 8`
  - `-10 <= nums[i] <= 10`
-

```

class Solution {
 public List<List<Integer>> permuteUnique(int[] nums) {
 Arrays.sort(nums);
 List<List<Integer>> res = new ArrayList<>();

 finfPermutations(new boolean[nums.length], nums, new ArrayList<>(),
 res);
 return res;
 }
 private void finfPermutations(boolean[] visited, int[] nums, List<Integer> curr, List<List<Integer>> res) {

 if (curr.size() == nums.length) { //we found 1st permutation

 if (!res.contains(curr))
 res.add(new ArrayList<Integer>(curr));
 return;
 }

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

 if (visited[i] || (i > 0 && nums[i] == nums[i-1] && visited[i])) continue;

 visited[i] = true;
 curr.add(nums[i]);
 finfPermutations(visited, nums, curr, res);
 curr.remove(curr.size()-1);
 visited[i] = false;

 }

 }
}

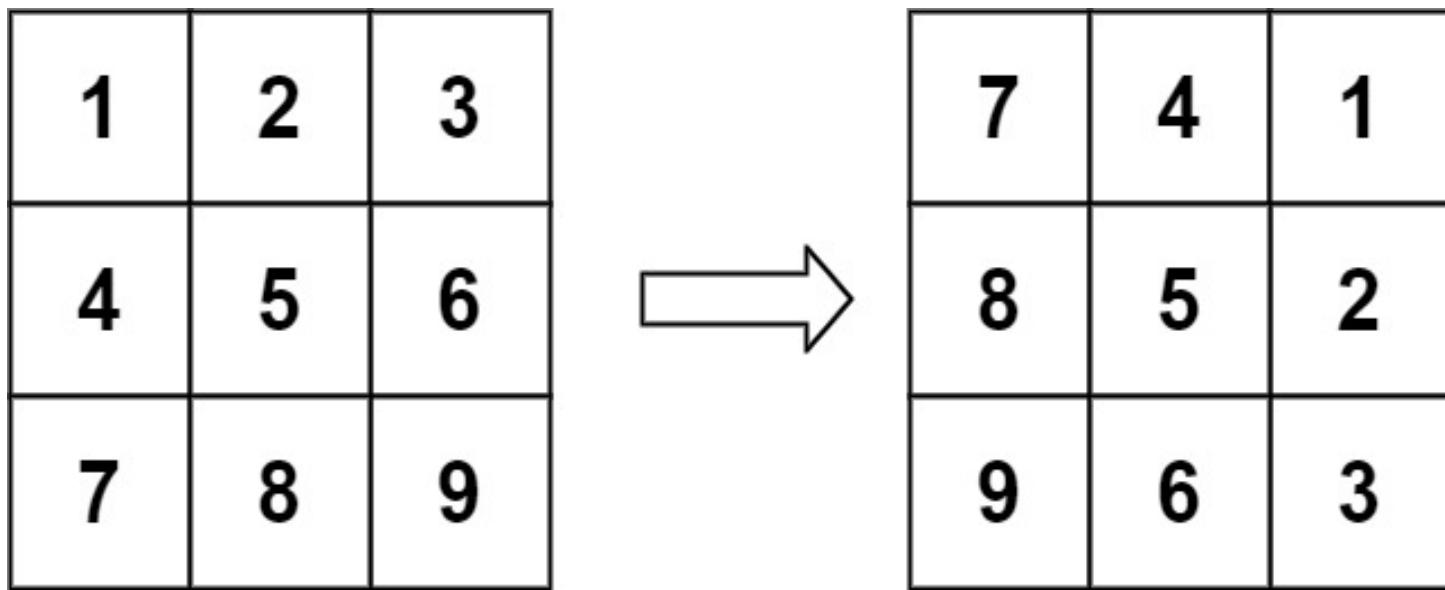
```

## 48. Rotate Image ↗

You are given an  $n \times n$  2D matrix representing an image, rotate the image by **90** degrees (clockwise).

You have to rotate the image **in-place** ([https://en.wikipedia.org/wiki/In-place\\_algorithm](https://en.wikipedia.org/wiki/In-place_algorithm)), which means you have to modify the input 2D matrix directly. **DO NOT** allocate another 2D matrix and do the rotation.

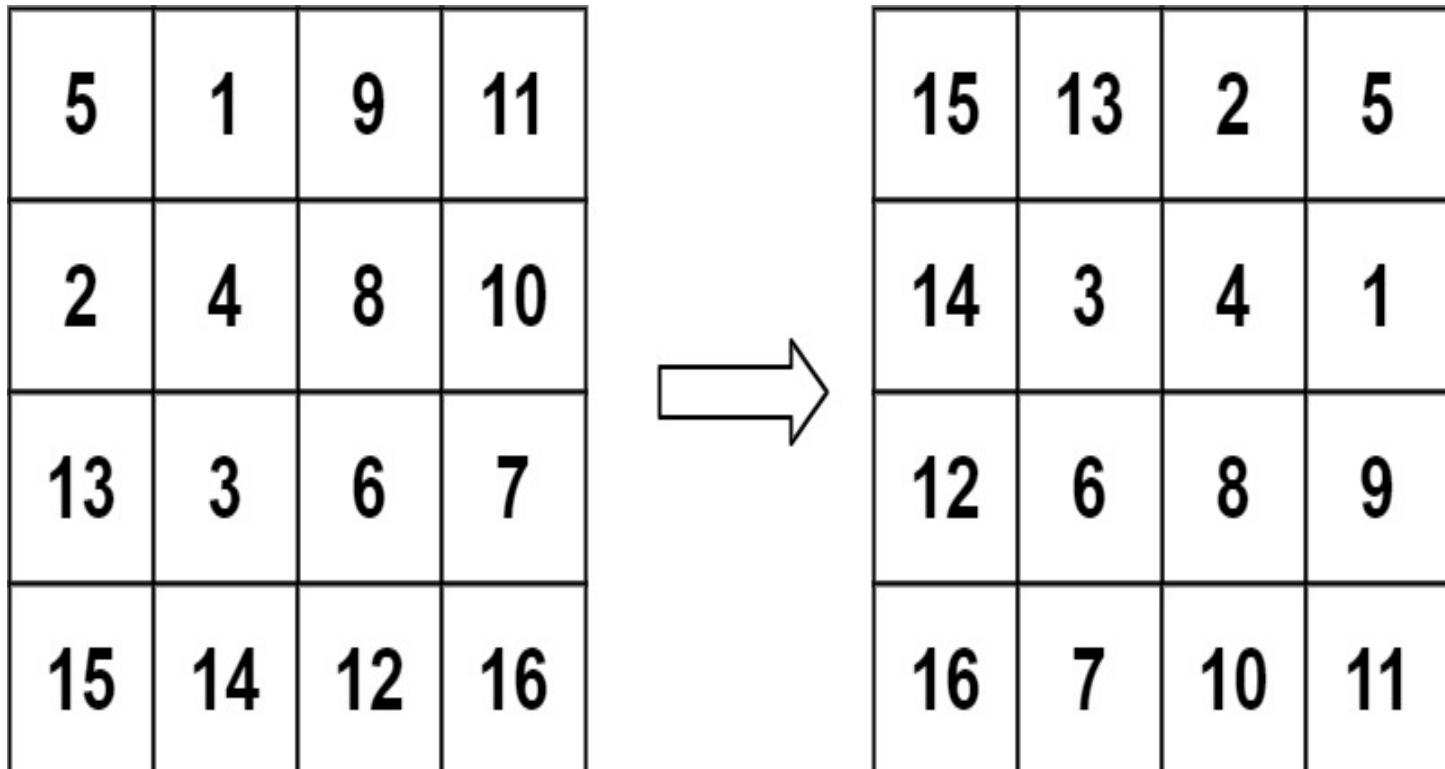
### Example 1:



**Input:** matrix = [[1,2,3],[4,5,6],[7,8,9]]

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

### Example 2:



**Input:** matrix = [[5,1,9,11],[2,4,8,10],[13,3,6,7],[15,14,12,16]]

**Output:** [[15,13,2,5],[14,3,4,1],[12,6,8,9],[16,7,10,11]]

### Example 3:

**Input:** matrix = [[1]]

**Output:** [[1]]

#### **Example 4:**

**Input:** matrix = [[1,2],[3,4]]

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

#### **Constraints:**

- `matrix.length == n`
  - `matrix[i].length == n`
  - `1 <= n <= 20`
  - `-1000 <= matrix[i][j] <= 1000`
- 

#### **Transpose and Interchange**

```
class Solution {
 public void rotate(int[][] matrix)
 {
 int temp = 0;
 //transpose matrix
 for (int i = 0; i < matrix.length; i++)
 {
 for (int j = i; j < matrix.length; j++)
 {
 temp = matrix[i][j];
 matrix[i][j] = matrix[j][i];
 matrix[j][i] = temp;
 }
 }
 int temp1 = 0;
 //interchange column for each row
 for (int i = 0; i < matrix.length; i++) //row
 {
 int li = 0, ri = matrix.length -1;

 while (li < ri)
 {
 temp = matrix[i][li];
 matrix[i][li] = matrix[i][ri];
 matrix[i][ri] = temp;
 li++;
 ri--;
 }
 }
 }
}
```

```

class Solution {
 public void rotate(int[][] matrix) {

 int size = matrix.length;
 int k = size - 1;
 double t = Math.floor(size/2);

 for (int i = 0; i < t; i++)
 {
 for (int j = i; j < k-i; j++)
 {
 int temp = matrix[i][j];
 matrix[i][j] = matrix[k-j][i];
 matrix[k-j][i] = matrix[k-i][k-j];
 matrix[k-i][k-j] = matrix[j][k-i];
 matrix[j][k-i] = temp;
 }
 }
 display(matrix);
 }

 private void display(int[][] matrix) {

 int size = matrix.length;

 for (int i = 0; i < size; i++)
 {
 for (int j = 0; j < size; j++)
 {
 System.out.print(matrix[i][j] + " ");
 }
 System.out.println();
 }
 }
}

```

## 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:

- $1 \leq \text{strs.length} \leq 10^4$
- $0 \leq \text{strs[i].length} \leq 100$
- $\text{strs[i]}$  consists of lowercase English letters.

```

class Solution {
 public List<List<String>> groupAnagrams(String[] strs) {

 HashMap<String, List<String>> map = new HashMap<String, List<String>>(); //hashmap key: string and value : list
 List<List<String>> ans = new ArrayList<List<String>>(); //list of list

 int size = strs.length;
 String currString;

 for (int i = 0; i < size; i++)
 {
 currString = strs[i];
 char[] chars = currString.toCharArray();
 Arrays.sort(chars);
 String sortedString = new String(chars);

 if (!map.containsKey(sortedString)) //if key is not present then add key
 {
 ArrayList<String> al = new ArrayList<String>(); //inserting key
 map.put(sortedString, al);
 }
 map.get(sortedString).add(currString); //add value for key "map.get(sortedString)" will give list present in 'value' then add currString in that list
 }

 for(List l : map.values()) //iterating over values
 {
 ans.add(l);
 }

 return ans;
 }
}

```

## 53. Maximum Subarray ↗



Given an integer array `nums`, find the contiguous subarray (containing at least one number) which has the largest sum and return *its sum*.

A **subarray** is a **contiguous** part of an array.

### Example 1:

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

**Output:** 6

**Explanation:** `[4,-1,2,1]` has the largest sum = 6.

### Example 2:

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

**Output:** 1

### Example 3:

**Input:** `nums = [5,4,-1,7,8]`

**Output:** 23

### Constraints:

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

**Follow up:** If you have figured out the  $O(n)$  solution, try coding another solution using the **divide and conquer** approach, which is more subtle.

```
class Solution {
 public int maxSubArray(int[] nums) {

 int maxSum = 0;
 int finalSum = Integer.MIN_VALUE;

 for (int i = 0; i < nums.length; i++)
 {
 if (maxSum + nums[i] > nums[i])
 {
 maxSum = maxSum + nums[i];
 finalSum = Math.max(finalSum, maxSum);
 }
 if (maxSum + nums[i] <= nums[i])
 {
 maxSum = nums[i];
 finalSum = Math.max(finalSum, maxSum);
 }

 }
 return finalSum;
 }
}
```

Practice

```

class Solution {
 public int maxSubArray(int[] nums) {

 if (nums == null || nums.length == 0) return 0;

 int sum = nums[0], maxSum = Integer.MIN_VALUE;

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

 if (sum > maxSum) maxSum = sum;

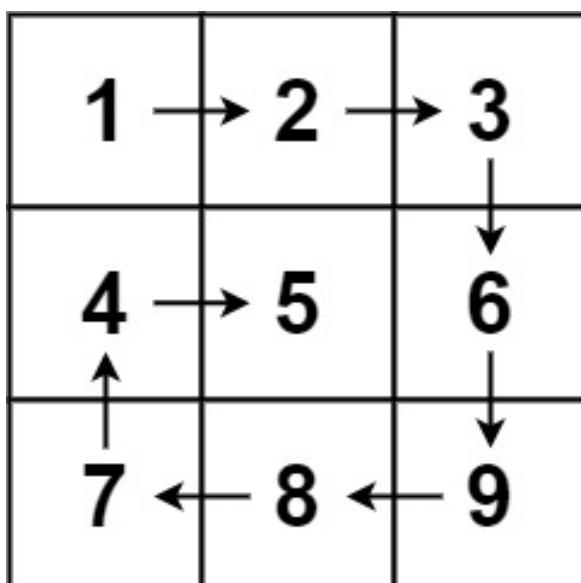
 if (sum + nums[i] >= nums[i]){ // purani train me shamil
 sum += nums[i];
 } else{
 sum = nums[i]; //apni train
 }
 }
 return (sum > maxSum)?sum:maxSum; //incase ek he element h tb sum h
e ans hoga
 }
}

```

## 54. Spiral Matrix ↗

Given an  $m \times n$  matrix , return *all elements of the matrix in spiral order.*

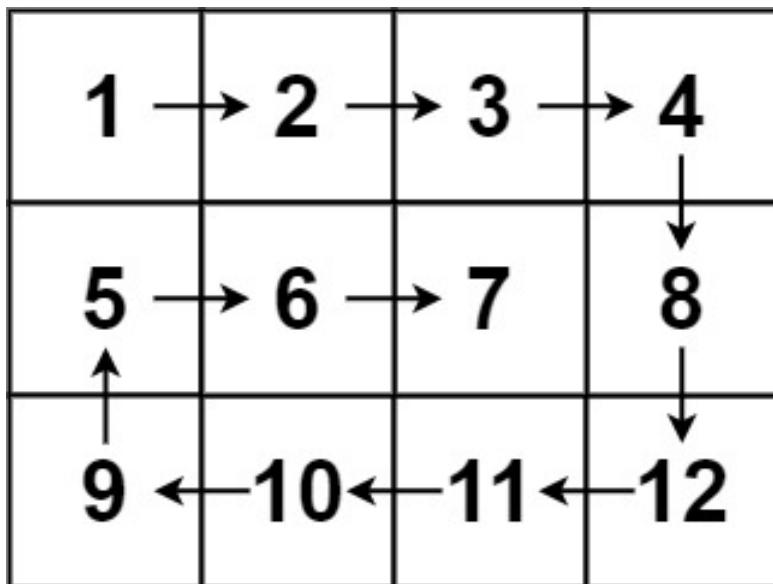
**Example 1:**



**Input:** matrix = [[1,2,3],[4,5,6],[7,8,9]]

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

### Example 2:



**Input:** matrix = [[1,2,3,4],[5,6,7,8],[9,10,11,12]]

**Output:** [1,2,3,4,8,12,11,10,9,5,6,7]

### Constraints:

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

```
class Solution {
 public List<Integer> spiralOrder(int[][] matrix) {

 List<Integer> ans = new ArrayList<Integer>();
 int row = matrix.length;
 int col = matrix[0].length;

 int top = 0, down = row-1;
 int left = 0, right = col-1;
 int count = row*col; //we have to maintain the count coz it is rectangular matrix will give wrong o/p without count
 // System.out.print("count "+ count);

 while (top <= down && left <= right)
 {
 for (int i = left; i <= right && count > 0; i++)
 {
 ans.add(matrix[top][i]);
 count--;
 }
 top++;
 for (int j = top; j <= down && count > 0; j++)
 {
 ans.add(matrix[j][right]);
 count--;
 }
 right--;
 for (int k = right; k >= left && count > 0; k--)//
 {
 ans.add(matrix[down][k]);
 count--;
 }
 down--;
 for (int l = down; l >= top && count > 0; l--)
 {
 ans.add(matrix[l][left]);
 count--;
 }
 left++;

 }
 return ans;
 }
}
```

# 55. Jump Game ↗



You are given an integer array `nums`. You are initially positioned at the array's **first index**, and each element in the array represents your maximum jump length at that position.

Return `true` if you can reach the last index, or `false` otherwise.

## Example 1:

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

**Output:** `true`

**Explanation:** Jump 1 step from index 0 to 1, then 3 steps to the last index.

## Example 2:

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

**Output:** `false`

**Explanation:** You will always arrive at index 3 no matter what. Its maximum ju

## Constraints:

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

```

class Solution {
 public boolean canJump(int[] nums) {

 int lastGoodIndex = nums[nums.length-1];
 for (int i = nums.length-1; i >=0; i--){
 if (i + nums[i] >= lastGoodIndex) { //last good index--> jaha se pakka ja skte h...to agr lastgood index pr aa gae to pahuch jaeg
 //agr hum last good indrx k aage ja skte h to last good index pr bhi ja he skte h
 lastGoodIndex = i;
 }
 }
 return (lastGoodIndex == 0);

 }
}

```

This approach is taking time

```

class Solution {
 public boolean canJump(int[] nums)
 {
 boolean[] dp = new boolean[nums.length];

 dp[nums.length-1] = true;
 for (int i = nums.length-2; i >= 0; i--)//main dp array
 {
 for (int j = 1; j <= nums[i]; j++) //kitne steps ja skte h
 {
 if (i+j < dp.length && dp[i+j] == true)
 {
 dp[i] = true;
 break;
 }
 }
 }
 return dp[0];
 }
}

```

## 56. Merge Intervals ↗



Given an array of intervals where  $\text{intervals}[i] = [\text{start}_i, \text{end}_i]$ , merge all overlapping intervals, and return *an array of the non-overlapping intervals that cover all the intervals in the input*.

### Example 1:

**Input:** `intervals = [[1,3],[2,6],[8,10],[15,18]]`

**Output:** `[[1,6],[8,10],[15,18]]`

**Explanation:** Since intervals [1,3] and [2,6] overlaps, merge them into [1,6].

### Example 2:

**Input:** `intervals = [[1,4],[4,5]]`

**Output:** `[[1,5]]`

**Explanation:** Intervals [1,4] and [4,5] are considered overlapping.

### Constraints:

- $1 \leq \text{intervals.length} \leq 10^4$
- $\text{intervals}[i].length == 2$
- $0 \leq \text{start}_i \leq \text{end}_i \leq 10^4$

```
//PEP style more easy to visualise
class Solution {
 public int[][] merge(int[][] intervals) {

 Arrays.sort(intervals, (a,b) -> a[0] - b[0]);
 List<int[]> ans = new ArrayList<>();

 for (int[] interval : intervals) {

 int[] currInt = interval;
 if (ans.size() == 0) { //1st interval
 ans.add(currInt);
 }
 else {
 int[] prev = ans.get(ans.size()-1);
 if (prev[1] >= currInt[0]) { //merge is there
 prev[1] = Math.max(prev[1], currInt[1]);
 }
 else {
 ans.add(currInt);
 }
 }
 }
 return ans.toArray(new int[ans.size()][]);
 }
}
```

```
class Solution {
 public int[][] merge(int[][] intervals) {

 List<int[]> list = new ArrayList<>();

 Arrays.sort(intervals, (a,b) -> a[0] - b[0]); //based on first index
 // a and b are diff 1D array
 // System.out.println(Arrays.deepToString(intervals));

 int[] curr = intervals[0];
 list.add(curr); //added 1st interval

 for (int[] interval : intervals) {
 int curr_begin = curr[0];
 int curr_end = curr[1];
 int next_begin = interval[0];
 int next_end = interval[1];

 if (curr_end >= next_begin) {
 curr_end = Math.max(curr_end, next_end);
 curr[1] = curr_end;
 }
 else {
 curr = interval;
 list.add(curr);
 }
 }

 return list.toArray(new int[list.size()][]);
 }
}
```

```

class Solution {
 public int[][] merge(int[][] intervals)
 {
 Pair[] p = new Pair[intervals.length]; //pair array
 int c = 0;
 for (int[] a : intervals) //Pair ka array banaya
 {
 p[c] = new Pair(a[0],a[1]);
 c++;
 }
 Arrays.sort(p); //Ascending order me sort

 Stack<Pair> st = new Stack<>();
 st.push(p[0]); //1st input kia

 for (int i = 1; i < p.length; i++)
 {
 Pair ps = st.peek(); //stack se
 Pair pa = p[i]; //array se

 if (pa.start <= ps.end) //merge hai ps-> [1,3], pa->[2,6]
 {
 if(pa.end > ps.end) ps.end = pa.end;
 st.pop(); //old value hatao
 st.push(ps); //new dalo merger wali
 }
 else
 {
 st.push(pa);
 }
 }
 int[][] ans = new int[st.size()][2];
 c = st.size()-1;
 while(!st.isEmpty())
 {
 Pair pp = st.pop();
 // System.out.println("start " + pp.start + " end " + pp.end);
 ans[c][0] = pp.start;
 ans[c][1] = pp.end;
 c--;
 }
 return ans;
 }

 class Pair implements Comparable<Pair>
 {

```

```

int start;
int end;

Pair(int start, int end)
{
 this.start = start;
 this.end = end;
}
//imp sorting in Ascending order
//this chota -ve
public int compareTo(Pair p)
{
 if (p.start > this.start) return -1;
 if (p.start < this.start) return +1;
 else return 0;
}
}

```

## 57. Insert Interval ↗

You are given an array of non-overlapping intervals `intervals` where `intervals[i] = [starti, endi]` represent the start and the end of the  $i^{\text{th}}$  interval and `intervals` is sorted in ascending order by `starti`. You are also given an interval `newInterval = [start, end]` that represents the start and end of another interval.

Insert `newInterval` into `intervals` such that `intervals` is still sorted in ascending order by `starti` and `intervals` still does not have any overlapping intervals (merge overlapping intervals if necessary).

Return `intervals` after the insertion.

### Example 1:

**Input:** `intervals = [[1,3],[6,9]]`, `newInterval = [2,5]`  
**Output:** `[[1,5],[6,9]]`

### Example 2:

**Input:** intervals = [[1,2],[3,5],[6,7],[8,10],[12,16]], newInterval = [4,8]

**Output:** [[1,2],[3,10],[12,16]]

**Explanation:** Because the new interval [4,8] overlaps with [3,5],[6,7],[8,10].

### Example 3:

**Input:** intervals = [], newInterval = [5,7]

**Output:** [[5,7]]

### Example 4:

**Input:** intervals = [[1,5]], newInterval = [2,3]

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

### Example 5:

**Input:** intervals = [[1,5]], newInterval = [2,7]

**Output:** [[1,7]]

### Constraints:

- $0 \leq \text{intervals.length} \leq 10^4$
- $\text{intervals[i].length} == 2$
- $0 \leq \text{start}_i \leq \text{end}_i \leq 10^5$
- $\text{intervals}$  is sorted by  $\text{start}_i$  in **ascending** order.
- $\text{newInterval.length} == 2$
- $0 \leq \text{start} \leq \text{end} \leq 10^5$

```

//try without pq
class Solution {
 public int[][] insert(int[][] intervals, int[] newInterval) {

 ArrayList<Interval> ans = new ArrayList<>();

 PriorityQueue<Interval> pq = new PriorityQueue<>((a,b) -> a.start - b.start);
 pq.add(new Interval(newInterval[0], newInterval[1]));

 for (int[] inte : intervals) {
 Interval temp = new Interval(inte[0], inte[1]);
 pq.add(temp);
 }

 Interval firstInterval = pq.remove();
 // if(pq.isEmpty()) ans.add(firstInterval);

 while (!pq.isEmpty()) {

 Interval fromPQ = pq.remove();

 if(fromPQ.start <= firstInterval.end) { //merge

 if(fromPQ.end <= firstInterval.end) { //end km hai to pura
he gayab
 fromPQ = firstInterval;

 }
 else if (fromPQ.end > firstInterval.end) { //end will change
now
 firstInterval.end = fromPQ.end;
 }
 }

 else if (fromPQ.start > firstInterval.end) { //no merge
 ans.add(firstInterval); //confirmed ki ye o ek dum alag hai
 firstInterval = fromPQ;

 }
 //pq khali hai matlab ye he last tha ab while k baher chala jaega
 // islia add kar do
 // if(pq.isEmpty()) ans.add(firstInterval);
 }

 if(pq.isEmpty()) ans.add(firstInterval); //upper k do if k jagah las
 }
}

```

t me ek if and all cases handled

```
////////*****////
 int finalSize = ans.size();

 int[][] ansArray= new int[finalSize][2];
 int row = 0;
 for (Interval i : ans) {
 int start = i.start;
 int end = i.end;
 ansArray[row][0] = start;
 ansArray[row][1] = end;
 row++;
 }
 return ansArray;

}

class Interval {

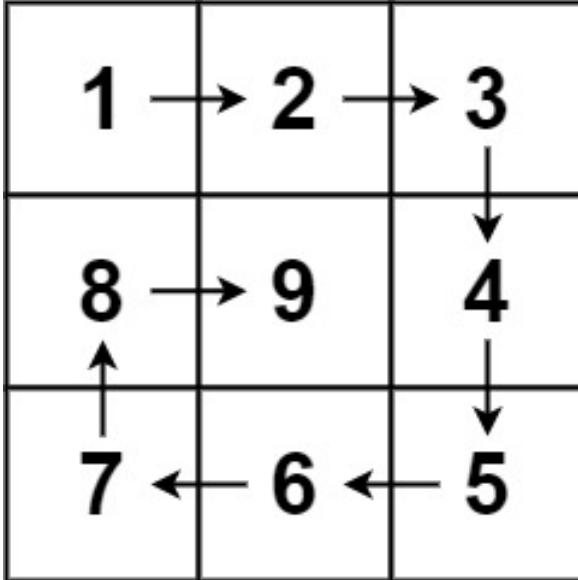
 int start;
 int end;

 Interval(int start, int end) {
 this.start = start;
 this.end = end;
 }
}
```

## 59. Spiral Matrix II ↗

Given a positive integer  $n$ , generate an  $n \times n$  matrix filled with elements from 1 to  $n^2$  in spiral order.

**Example 1:**



**Input:** n = 3

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

### Example 2:

**Input:** n = 1

**Output:** [[1]]

### Constraints:

- $1 \leq n \leq 20$

```

class Solution {
 public int[][] generateMatrix(int n)
 {
 int result[][] = new int [n][n];

 int rowStart = 0, rowEnd = n-1;
 int colStart = 0, colEnd = n-1;
 int c = 1;

 while (rowStart <= rowEnd && colStart <= colEnd)
 {
 for (int i = colStart; i <= colEnd; i++)
 {
 result[rowStart][i] = c++;
 }rowStart++;
 for (int j = rowStart; j <= rowEnd; j++)
 {
 result[j][colEnd] = c++;
 }colEnd--;
 if (colEnd >= colStart)
 {
 for (int i = colEnd; i >= colStart; i--)
 {
 result[rowEnd][i] = c++;
 }rowEnd--;
 }
 if (rowEnd >= rowStart)
 {
 for (int j = rowEnd; j >= rowStart; j--)
 {
 result[j][colStart] = c++;
 }colStart++;
 }
 }
 return result;
 }
}

```

## 62. Unique Paths ↗



A robot is located at the top-left corner of a  $m \times n$  grid (marked 'Start' in the diagram below).

The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid (marked 'Finish' in the diagram below).

How many possible unique paths are there?

### Example 1:



**Input:**  $m = 3$ ,  $n = 7$

**Output:** 28

### Example 2:

**Input:**  $m = 3$ ,  $n = 2$

**Output:** 3

#### Explanation:

From the top-left corner, there are a total of 3 ways to reach the bottom-right corner.

1. Right -> Down -> Down
2. Down -> Down -> Right
3. Down -> Right -> Down

### Example 3:

**Input:**  $m = 7$ ,  $n = 3$

**Output:** 28

### Example 4:

**Input:**  $m = 3$ ,  $n = 3$

**Output:** 6

### Constraints:

- $1 \leq m, n \leq 100$
- It's guaranteed that the answer will be less than or equal to  $2 * 10^9$ .

```

class Solution {
 public int uniquePaths(int m, int n) {

 int dp[][] = new int[m][n];

 dp[m-1][n-1] = 0;
 //last row.. direct right
 for (int i = 0; i < n; i++)
 {
 dp[m-1][i] = 1;
 }
 //last col..direct down
 for (int i = 0; i < m; i++)
 {
 dp[i][n-1] = 1;
 }

 for (int i = m-2; i >= 0; i--)
 {
 for (int j = n-2; j >= 0; j--)
 {
 dp[i][j] = dp[i][j+1] + dp[i+1][j];
 }
 }

 return dp[0][0];
 }
}

```

## 63. Unique Paths II ↗

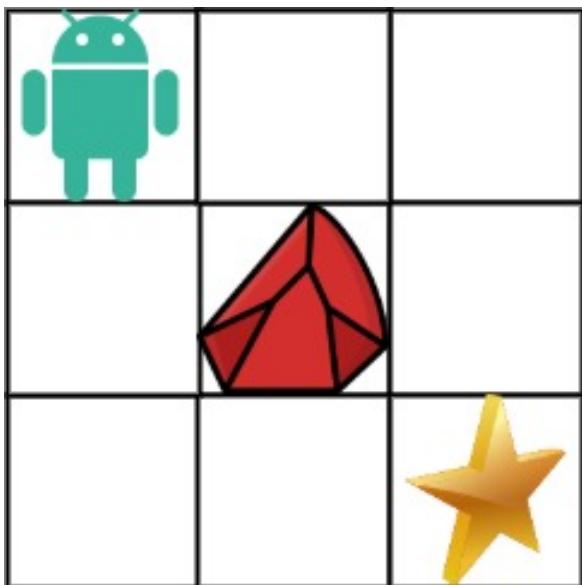
A robot is located at the top-left corner of a  $m \times n$  grid (marked 'Start' in the diagram below).

The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid (marked 'Finish' in the diagram below).

Now consider if some obstacles are added to the grids. How many unique paths would there be?

An obstacle and space is marked as `1` and `0` respectively in the grid.

### Example 1:



**Input:** obstacleGrid = [[0,0,0],[0,1,0],[0,0,0]]

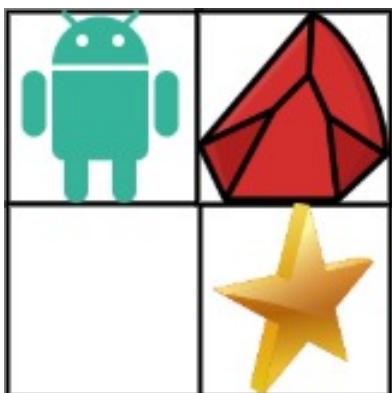
**Output:** 2

**Explanation:** There is one obstacle in the middle of the 3x3 grid above.

There are two ways to reach the bottom-right corner:

1. Right -> Right -> Down -> Down
2. Down -> Down -> Right -> Right

### Example 2:



**Input:** obstacleGrid = [[0,1],[0,0]]

**Output:** 1

### Constraints:

- $m == \text{obstacleGrid.length}$
- $n == \text{obstacleGrid}[i].length$
- $1 \leq m, n \leq 100$
- $\text{obstacleGrid}[i][j]$  is 0 or 1.

```

class Solution {
 public int uniquePathsWithObstacles(int[][][] obstacleGrid) {

 int row = obstacleGrid.length;
 int col = obstacleGrid[0].length;

 int[][] dp = new int[row][col];
 //corner case: agr last cell me he rock hai tb koi rasta ni h
 if (obstacleGrid[row-1][col-1] == 1)
 {
 return 0;
 }

 //last cell
 dp[row-1][col-1] = 1;

 //last row: move only right
 for (int i = col-2; i >= 0; i--)
 {
 if (obstacleGrid[row-1][i] != 1 && obstacleGrid[row-1][i+1] != 1 && dp[row-1][i+1] != 0)
 {
 dp[row-1][i] = 1;
 }
 }

 //last col: move only down
 for (int i = row-2; i >= 0; i--)
 {
 if (obstacleGrid[i][col-1] != 1 && obstacleGrid[i+1][col-1] != 1 && dp[i+1][col-1] != 0)
 {
 dp[i][col-1] = 1;
 }
 }

 //rest matrix
 for(int i = row-2; i >= 0; i--)
 {
 for (int j = col-2; j >= 0; j--)
 {
 if (obstacleGrid[i][j] != 1)
 {
 dp[i][j] = dp[i][j+1] + dp[i+1][j];
 }
 else
 {

```

```

 dp[i][j] = 0;
 }
}

return dp[0][0];
}
}

```

## 64. Minimum Path Sum ↗

Given a  $m \times n$  grid filled with non-negative numbers, find a path from top left to bottom right, which minimizes the sum of all numbers along its path.

**Note:** You can only move either down or right at any point in time.

### Example 1:

|   |   |   |
|---|---|---|
| 1 | 3 | 1 |
| 1 | 5 | 1 |
| 4 | 2 | 1 |

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

**Output:** 7

**Explanation:** Because the path 1 → 3 → 1 → 1 → 1 minimizes the sum.

### Example 2:

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

**Output:** 12

## Constraints:

- $m == \text{grid.length}$
- $n == \text{grid}[i].length$
- $1 \leq m, n \leq 200$
- $0 \leq \text{grid}[i][j] \leq 100$

```
class Solution {
 public int minPathSum(int[][] grid)
 {
 int row = grid.length-1;
 int col = grid[0].length-1;

 int dp[][] = new int[row+1][col+1];
 dp[row][col] = grid[row][col];

 //last row : move right
 for (int i = col-1; i >= 0; i--)
 {
 dp[row][i] = grid[row][i] + dp[row][i+1];
 }
 //last col : move left
 for (int i = row-1; i >= 0; i--)
 {
 dp[i][col] = grid[i][col] + dp[i+1][col];
 }

 for (int i = row -1; i >= 0; i--)
 {
 for (int j = col -1; j >= 0; j--)
 {
 dp[i][j] = grid[i][j] + Math.min(dp[i][j+1], dp[i+1][j]);
 }
 }
 return dp[0][0];
 }
}
```



You are given a **large integer** represented as an integer array `digits`, where each `digits[i]` is the  $i^{\text{th}}$  digit of the integer. The digits are ordered from most significant to least significant in left-to-right order. The large integer does not contain any leading 0's.

Increment the large integer by one and return *the resulting array of digits*.

### Example 1:

**Input:** `digits = [1,2,3]`

**Output:** `[1,2,4]`

**Explanation:** The array represents the integer 123.

Incrementing by one gives  $123 + 1 = 124$ .

Thus, the result should be `[1,2,4]`.

### Example 2:

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

**Output:** `[4,3,2,2]`

**Explanation:** The array represents the integer 4321.

Incrementing by one gives  $4321 + 1 = 4322$ .

Thus, the result should be `[4,3,2,2]`.

### Example 3:

**Input:** `digits = [0]`

**Output:** `[1]`

**Explanation:** The array represents the integer 0.

Incrementing by one gives  $0 + 1 = 1$ .

Thus, the result should be `[1]`.

### Example 4:

**Input:** `digits = [9]`

**Output:** `[1,0]`

**Explanation:** The array represents the integer 9.

Incrementing by one gives  $9 + 1 = 10$ .

Thus, the result should be `[1,0]`.

### Constraints:

- $1 \leq \text{digits.length} \leq 100$

- $0 \leq \text{digits}[i] \leq 9$
  - `digits` does not contain any leading `0`'s.
- 

## Interview Bit

```
public static ArrayList<Integer> plusOne(ArrayList<Integer> A)
{
 int n;
 boolean flag = false;
 ArrayList<Integer> al = new ArrayList<Integer>();

 for (int i = A.size() - 1; i >= 0; i--)
 {
 // flag will indicate if addition is done already
 if (A.get(i) == 9 && !flag)
 {
 A.set(i, 0);
 flag = false;
 }
 else if (A.get(i) < 9 && !flag)
 {
 A.set(i, A.get(i) + 1);
 flag = true;
 }
 }
 // if addition is not done this means num was 999 so we need to add
1 in
 // starting
 if (!flag)
 {
 al.add(1);
 for (int i = 0; i < A.size(); i++)
 {
 al.add(A.get(i));
 }
 return al;
 }
 else
 {
 // if final number has 0 before it
 int i = 0;
 while (A.size() > 0 && A.get(i) == 0)
 {
 A.remove(i);
 }
 }
 return A;
}
```

```

class Solution {
 public int[] plusOne(int[] digits) {

 int l = digits.length;
 for (int i = l-1; i >=0; i--)
 {
 if(digits [i] < 9)
 {
 digits[i] = digits[i]+1;
 return digits;
 }
 else
 {
 digits[i] = 0;
 }
 }

 int[] newNum = new int[l+1];
 newNum[0] = 1;

 return newNum;
 }
}

```

## 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 <= n <= 45

```
class Solution {
 public int climbStairs(int n) {
 int[] memo = new int[n+1];

 return climbHelp(n , memo);

 }

 int climbHelp(int n , int[] memo) {

 if (memo[n] != 0) return memo[n];
 if (n == 0 || n ==1) return 1;

 int count = climbHelp(n-1, memo)+ climbHelp(n-2, memo); //can take
2 stairs
 memo[n] = count;
 return count;
 }
}
```

```

class Solution {
 public int climbStairs(int n) {

 int[] dp = new int[n+1];

 dp[0] = 1; // number of ways to climb 0 steps
 dp[1] = 1; // number of ways to climb 1 steps

 for (int i = 2; i <= n; i++)
 {
 dp[i] = dp[i-1]+dp[i-2];
 }
 return dp[dp.length-1];
 }
}

```

## 73. Set Matrix Zeroes ↗

Given an  $m \times n$  integer matrix  $\text{matrix}$ , if an element is  $0$ , set its entire row and column to  $0$ 's, and return *the matrix*.

You must do it in place ([https://en.wikipedia.org/wiki/In-place\\_algorithm](https://en.wikipedia.org/wiki/In-place_algorithm)).

### Example 1:

|   |   |   |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

|   |   |   |
|---|---|---|
| 1 | 0 | 1 |
| 0 | 0 | 0 |
| 1 | 0 | 1 |

**Input:**  $\text{matrix} = [[1,1,1],[1,0,1],[1,1,1]]$

**Output:**  $[[1,0,1],[0,0,0],[1,0,1]]$

### Example 2:

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 2 | 0 |
| 3 | 4 | 5 | 2 |
| 1 | 3 | 1 | 5 |



|   |   |   |   |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 4 | 5 | 0 |
| 0 | 3 | 1 | 0 |

**Input:** matrix = [[0,1,2,0],[3,4,5,2],[1,3,1,5]]

**Output:** [[0,0,0,0],[0,4,5,0],[0,3,1,0]]

### Constraints:

- $m == \text{matrix.length}$
- $n == \text{matrix[0].length}$
- $1 \leq m, n \leq 200$
- $-2^{31} \leq \text{matrix}[i][j] \leq 2^{31} - 1$

### Follow up:

- A straightforward solution using  $O(mn)$  space is probably a bad idea.
- A simple improvement uses  $O(m + n)$  space, but still not the best solution.
- Could you devise a constant space solution?

```

class Solution {
 //Approach: matrix ka 1st row and 1st col ko use kar rai...ye store kar
 ne k lia ki is row ya is col me sb ko 0 karna h

 public void setZeroes(int[][] matrix)
 {
 boolean firstRow = false;
 boolean firstCol = false;
 //check if 1st row is having 0, baad me use karege--> corner case
 for (int j = 0; j < matrix[0].length; j++) // col will change row w
ill be 0
 {
 if (matrix[0][j] == 0) firstRow = true;
 }
 //check if 1st col is having 0, baad me use karege --> corner case
 for (int i = 0; i < matrix.length; i++)//row will change col will b
e 0
 {
 if (matrix[i][0] == 0) firstCol = true;
 }

 //make first row and fir col as identifier of 0 rows
 for (int i = 1; i < matrix.length; i++)
 {
 for (int j = 1; j < matrix[0].length; j++)
 {
 if (matrix[i][j] == 0)
 {
 matrix[i][0] = 0; //first col
 matrix[0][j] = 0; //first row
 }
 }
 }
 //first row and col k acc 0 kar rai
 for (int i = 1; i < matrix.length; i++)
 {
 for (int j = 1; j < matrix[0].length; j++)
 {
 if (matrix[i][0] == 0) matrix[i][j] = 0;
 if (matrix[0][j] == 0) matrix[i][j] = 0;
 }
 }

 if (firstRow)//make first row 0
 {
 for (int j = 0; j < matrix[0].length; j++)

```

```

 matrix[0][j] = 0;
 }
}

if (firstCol)//make first col 0
{
 for (int i = 0; i < matrix.length; i++)
 {
 matrix[i][0] = 0;
 }
}
}

```

## 74. Search a 2D Matrix ↗

Write an efficient algorithm that searches for a value in an  $m \times n$  matrix. This matrix has the following properties:

- Integers in each row are sorted from left to right.
- The first integer of each row is greater than the last integer of the previous row.

### 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:

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

```

class Solution {
 public boolean searchMatrix(int[][] matrix, int target)
 {
 int row = matrix.length;
 int col = matrix[0].length;

 int rs = 0, cs = col-1;

 while (rs < row && cs >= 0)
 {
 if (matrix[rs][cs] == target) return true;

 else if (matrix[rs][cs] > target) cs--;

 else if (matrix[rs][cs] < target) rs++;
 }

 return false;
 }
}

```

## 75. Sort Colors ↗

Given an array `nums` with `n` objects colored red, white, or blue, sort them **in-place** ([https://en.wikipedia.org/wiki/In-place\\_algorithm](https://en.wikipedia.org/wiki/In-place_algorithm)) so that objects of the same color are adjacent, with the colors in the order red, white, and blue.

We will use the integers `0`, `1`, and `2` to represent the color red, white, and blue, respectively.

You must solve this problem without using the library's sort function.

### Example 1:

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

### Example 2:

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

### **Example 3:**

**Input:** nums = [0]

**Output:** [0]

### **Example 4:**

**Input:** nums = [1]

**Output:** [1]

### **Constraints:**

- n == nums.length
- 1 <= n <= 300
- nums[i] is 0, 1, or 2.

**Follow up:** Could you come up with a one-pass algorithm using only constant extra space?

---

```

class Solution {
 public void sortColors(int[] nums)
 {
 int i = 0;
 int start = 0, end = nums.length-1;
 int temp;

 while (i <= end)
 {
 if (nums[i] == 0)
 {
 temp = nums[start];
 nums[start] = nums[i];
 nums[i] = temp;
 start++;
 i++;
 }
 else if (nums[i] == 2)
 {
 temp = nums[end];
 nums[end] = nums[i];
 nums[i] = temp;
 end--;
 }
 else if (nums[i] == 1)
 {
 i++;
 }
 }
 }
}

```

## 77. Combinations ↗

Given two integers  $n$  and  $k$ , return *all possible combinations of  $k$  numbers out of the range  $[1, n]$* .

You may return the answer in **any order**.

**Example 1:**

**Input:** n = 4, k = 2

**Output:**

```
[
 [2,4],
 [3,4],
 [2,3],
 [1,2],
 [1,3],
 [1,4],
]
```

### Example 2:

**Input:** n = 1, k = 1

**Output:** [[1]]

### Constraints:

- $1 \leq n \leq 20$
  - $1 \leq k \leq n$
-

```

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

 List<List<Integer>> res = new ArrayList<>();
 findComb(1, n, k, new ArrayList<Integer>(), res);
 return res;
 }

 void findComb(int start, int n, int k, ArrayList<Integer> curr, List<List<Integer>> res) {

 if (curr.size() == k) {
 res.add(new ArrayList<Integer>(curr));
 return;
 }
 for (int i = start; i <= n; i++) {

 curr.add(i);
 findComb(i+1, n, k, curr, res);
 curr.remove(curr.size()-1);
 }
 }
}

```

## 78. Subsets ↗

Given an integer array `nums` of **unique** elements, return *all possible subsets (the power set)*.

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

### Example 1:

|                                                                  |
|------------------------------------------------------------------|
| <b>Input:</b> nums = [1,2,3]                                     |
| <b>Output:</b> [[], [1], [2], [1,2], [3], [1,3], [2,3], [1,2,3]] |

### Example 2:

|                          |
|--------------------------|
| <b>Input:</b> nums = [0] |
| <b>Output:</b> [[], [0]] |

## Constraints:

- $1 \leq \text{nums.length} \leq 10$
- $-10 \leq \text{nums}[i] \leq 10$
- All the numbers of `nums` are **unique**.

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

 List<List<Integer>> subsets = new ArrayList<>();
 generateSubsets(0, nums, new ArrayList<>(), subsets);
 return subsets;
 }

 private void generateSubsets(int start,int[] nums, List<Integer> currLi
st,
subsets){

 subsets.add(new ArrayList<>(currList)); //add copy of current list

 for (int i = start; i < nums.length; i++) {
 currList.add(nums[i]); //added in Current List
 generateSubsets(i+1, nums, currList, subsets);
 currList.remove(currList.size()-1); //remove last element--> ba
cktracking vapas aate time pure list dubara empty ho jaeg
 }
 }
}
```

```

class Solution {
 public List<List<Integer>> subsets(int[] nums)
 {
 List<Integer> l = new ArrayList<>();
 List<List<Integer>> resList = new ArrayList<>();

 subsets(nums, 0, l, resList); //call by ref use kar rai
 return resList;
 }
 public void subsets(int[] nums, int s, List<Integer> l, List<List<Integer>> resList)
 {
 if (s == nums.length)
 {
 resList.add(new ArrayList<Integer>(l)); //mil gar ek list, add
 karo
 return; //add karo return ho jao
 }
 l.add(nums[s]);
 subsets(nums, s+1, l, resList); //send s+1 not ++s or s++
 l.remove(l.size()-1);
 subsets(nums, s+1, l, resList);
 }
}

```

## 79. Word Search ↗

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

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

**Example 1:**

|   |   |   |   |
|---|---|---|---|
| A | B | C | E |
| S | F | C | S |
| A | D | E | E |

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

**Output:** true

### Example 2:

|   |   |   |   |
|---|---|---|---|
| A | B | C | E |
| S | F | C | S |
| A | D | E | E |

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

**Output:** true

### Example 3:

|   |   |   |   |
|---|---|---|---|
| A | B | C | E |
| S | F | C | S |
| A | D | E | E |

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

**Output:** false

### Constraints:

- $m == \text{board.length}$
- $n = \text{board}[i].length$
- $1 \leq m, n \leq 6$
- $1 \leq \text{word.length} \leq 15$
- board and word consists of only lowercase and uppercase English letters.

**Follow up:** Could you use search pruning to make your solution faster with a larger board ?

```

class Solution {
 public boolean exist(char[][] board, String word)
 {
 boolean [][] visited = new boolean [board.length][board[0].length];
 char firstChar = word.charAt(0);
 for (int i = 0; i < board.length; i++)
 {
 for (int j = 0; j < board[0].length; j++)
 {
 //jb pahla character board pr mil jae tb DFS karo board me
 if (board[i][j] == firstChar)
 {
 if (helper(board, word, visited, i, j, 0)) return true;
 }
 }
 }
 return false;
 }

 private boolean helper(char[][] board, String word, boolean[][] visited, int i, int j, int count)
 {
 //agar abhi ka count (word ka character) same ni h to return kar ja o
 if (i < 0 || i >= board.length || j < 0 || j >= board[0].length || visited[i][j] || word.charAt(count) != board[i][j]) return false;

 if (word.length()-1 == count) return true; //means word mil gaya..count equal ho gya word k length se

 visited[i][j] = true;
 if (helper(board, word, visited, i+1, j, count+1) || helper(board, word, visited, i-1, j, count+1)
 || helper(board, word, visited, i, j+1, count+1) || helper(board, word, visited, i, j-1, count+1))
 return true;

 //backtracking 2 lines
 visited[i][j] = false;
 count--;

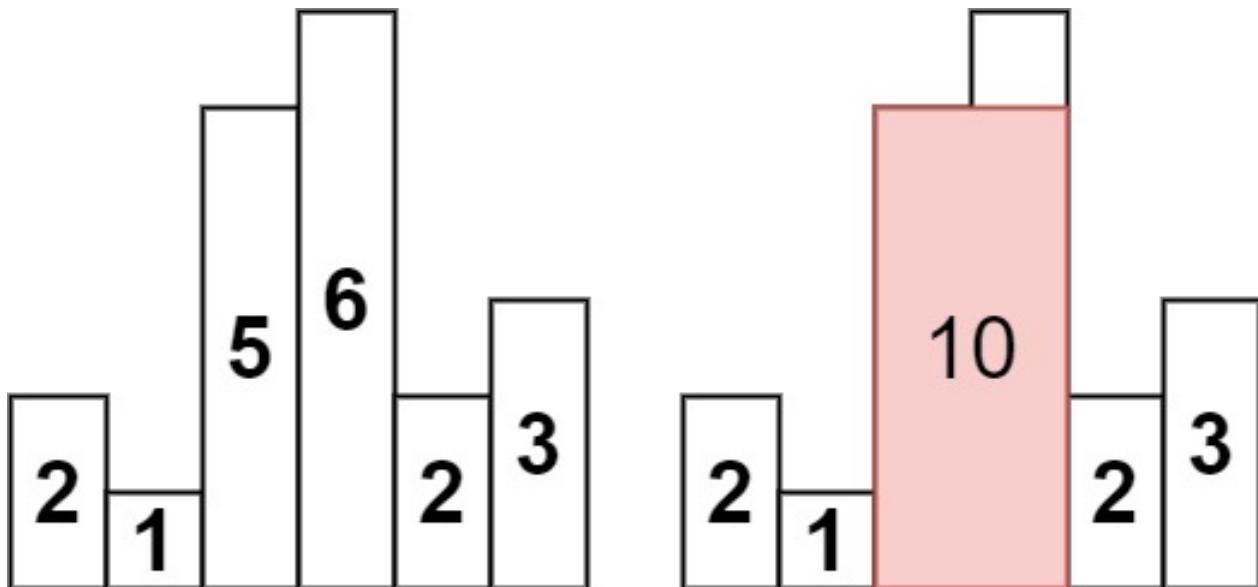
 return false;
 }
}

```

## 84. Largest Rectangle in Histogram ↗

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

### Example 1:

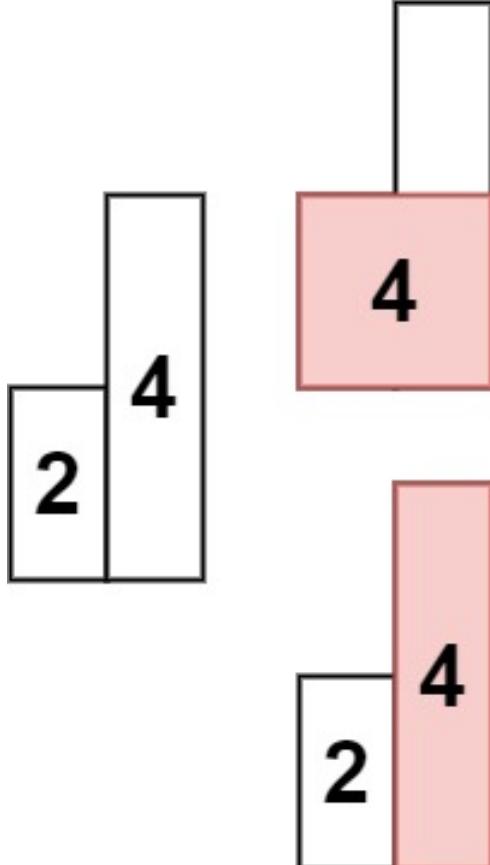


**Input:** heights = [2,1,5,6,2,3]

**Output:** 10

**Explanation:** The above is a histogram where width of each bar is 1. The largest rectangle is shown in the red area, which has an area = 10 units.

### Example 2:



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

**Output:** 4

**Constraints:**

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

```

class Solution {
 public int largestRectangleArea(int[] heights)
 {
 int max = Integer.MIN_VALUE;
 int[] nextSmallRight = smallestToRight(heights);
 int[] nextSmallLeft = smallestToLeft(heights);

 for (int i = 0; i < heights.length; i++)
 {
 int area = heights[i] * (nextSmallRight[i] - nextSmallLeft[i] - 1);
 max = Math.max(max, area);
 }
 return max;
 }

 public int[] smallestToRight(int[] heights)
 {
 Stack<Integer> st = new Stack<>();
 int[] sR = new int[heights.length];

 for (int i = heights.length-1; i >= 0; i--)//right se start kar rai
 {
 if (st.isEmpty())
 {
 sR[i] = heights.length;//array k baher minimum h(right side)
 st.push(i); //index daal rai
 }
 else
 {
 while(!st.isEmpty() && heights[st.peek()] >= heights[i])
 {
 st.pop();
 }
 if (st.isEmpty())
 {
 sR[i] = heights.length;//array k baher minimum h(right side)
 st.push(i);
 }
 else
 {
 sR[i] = st.peek(); //index array me b
 st.push(i);
 }
 }
 }
 }
}

```

```

 }
 return sR;
}
public int[] smallestToLeft(int[] heights)
{
 Stack<Integer> st = new Stack<>();
 int[] sL = new int[heights.length];

 for (int i = 0; i < heights.length; i++)
 {
 if (st.isEmpty())
 {
 sL[i] = -1;//array k baher minimum h(left side)
 st.push(i); //index daal rai
 }
 else
 {
 while(!st.isEmpty() && heights[st.peek()] >= heights[i])
 {
 st.pop();
 }
 if (st.isEmpty())
 {
 sL[i] = -1;//array k baher minimum h(left side)
 st.push(i);
 }
 else
 {
 sL[i] = st.peek(); //index array me b
 st.push(i);
 }
 }
 }
 return sL;
}
}

```

## 85. Maximal Rectangle ↗

Given a `rows x cols` binary matrix filled with 0's and 1's, find the largest rectangle containing only 1's and return its area.

### **Example 1:**

|   |   |   |   |   |
|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 |

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

**Output:** 6

**Explanation:** The maximal rectangle is shown in the above picture.

### **Example 2:**

**Input:** matrix = []

**Output:** 0

### **Example 3:**

**Input:** matrix = [[ "0"]]

**Output:** 0

### **Example 4:**

**Input:** matrix = [[ "1"]]

**Output:** 1

### **Example 5:**

**Input:** matrix = [[ "0", "0" ]]

**Output:** 0

**Constraints:**

- rows == matrix.length
  - cols == matrix[i].length
  - 0 <= row, cols <= 200
  - matrix[i][j] is '0' or '1' .
-

```
class Solution {
 public int maximalRectangle(char[][] matrix) {
 if (matrix == null || matrix.length == 0) return 0;

 int[] currRow = new int[matrix[0].length];

 int maxArea = Integer.MIN_VALUE;

 for (int i = 0; i < matrix.length; i++) {
 for (int j = 0; j < matrix[0].length; j++) {

 if (matrix[i][j] == '0') currRow[j] = 0;
 else {
 currRow[j] = 1 + currRow[j];
 }
 }

 int area = largestRectangleArea(currRow);
 maxArea = Math.max(maxArea, area);
 }
 return maxArea;
 }

 public int largestRectangleArea(int[] heights) {

 int[] minToLeft = minimumToLeft(heights);
 int[] minToRight = minimumToRight(heights);

 int area = 0;
 int totalArea = 0;

 for (int i = 0; i < heights.length; i++) {
 int ml = minToLeft[i];
 int rl = minToRight[i];

 area = heights[i] * (rl - ml-1);

 totalArea = Math.max(totalArea, area);
 area = 0;
 }

 return totalArea;
 }

 private int[] minimumToLeft(int[] heights) {
 int[] ret = new int[heights.length];
```

```

Stack<Integer> st = new Stack<>();

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

 while (!st.isEmpty() && heights[i] <= heights[st.peek()]) {
 st.pop();
 }
 if (!st.isEmpty()) {
 ret[i] = st.peek();
 st.push(i);
 }
 else {
 ret[i] = -1;
 st.push(i);
 }

}
return ret;
}

private int[] minimumToRight(int[] heights) {
 int[] ret = new int[heights.length];
 Stack<Integer> st = new Stack<>();

 for (int i = heights.length-1; i >= 0; i--) {

 while (!st.isEmpty() && heights[i] <= heights[st.peek()]) {
 st.pop();
 }
 if (!st.isEmpty()) {
 ret[i] = st.peek();
 st.push(i);
 }
 else {
 ret[i] = heights.length;
 st.push(i);
 }

 }
 return ret;
}
}

```

## 88. Merge Sorted Array ↗



You are given two integer arrays `nums1` and `nums2`, sorted in **non-decreasing order**, and two integers `m` and `n`, representing the number of elements in `nums1` and `nums2` respectively.

**Merge** `nums1` and `nums2` into a single array sorted in **non-decreasing order**.

The final sorted array should not be returned by the function, but instead be *stored inside the array* `nums1`. To accommodate this, `nums1` has a length of `m + n`, where the first `m` elements denote the elements that should be merged, and the last `n` elements are set to `0` and should be ignored. `nums2` has a length of `n`.

### Example 1:

**Input:** `nums1 = [1,2,3,0,0,0]`, `m = 3`, `nums2 = [2,5,6]`, `n = 3`

**Output:** `[1,2,2,3,5,6]`

**Explanation:** The arrays we are merging are `[1,2,3]` and `[2,5,6]`.

The result of the merge is `[1,2,2,3,5,6]` with the underlined elements coming

### Example 2:

**Input:** `nums1 = [1]`, `m = 1`, `nums2 = []`, `n = 0`

**Output:** `[1]`

**Explanation:** The arrays we are merging are `[1]` and `[]`.

The result of the merge is `[1]`.

### Example 3:

**Input:** `nums1 = [0]`, `m = 0`, `nums2 = [1]`, `n = 1`

**Output:** `[1]`

**Explanation:** The arrays we are merging are `[]` and `[1]`.

The result of the merge is `[1]`.

Note that because `m = 0`, there are no elements in `nums1`. The `0` is only there

### Constraints:

- `nums1.length == m + n`
- `nums2.length == n`
- `0 <= m, n <= 200`
- `1 <= m + n <= 200`

- $-10^9 \leq \text{nums1}[i], \text{nums2}[j] \leq 10^9$

**Follow up:** Can you come up with an algorithm that runs in  $O(m + n)$  time?

```
class Solution {
 public void merge(int[] nums1, int m, int[] nums2, int n) {

 //corner case
 if (n == 0) return;
 else if (m == 0 && n > 0) {
 for (int i = 0; i < n; i++) {
 nums1[i] = nums2[i];
 }
 }
 ////
 int i = m-1, j = n-1, k = m+n -1;
 while (i >= 0 && j >=0) {
 if (nums1[i] >= nums2[j]) {
 nums1[k] = nums1[i];
 nums1[i] = 0;
 i--;

 } else if (nums2[j] > nums1[i]) {
 nums1[k] = nums2[j];
 j--;

 }
 k--;
 }
 while (i < 0 && j >= 0) {
 nums1[k] = nums2[j];
 j--;
 k--;
 }
 while (j < 0 && i >= 0) {
 nums1[k] = nums1[i];
 i--;
 k--;
 }
 }
}
```

```
class Solution {
 public void merge(int[] nums1, int m, int[] nums2, int n)
 {

 int res[] = new int[nums1.length];
 int i = 0, j = 0, c = 0;
 while (i < m && j < n)
 {
 if (nums1[i] < nums2[j])
 {
 res[c] = nums1[i];
 i++;
 c++;
 }
 else
 {
 res[c] = nums2[j];
 j++;
 c++;
 }
 // res[i] = Math.max(nums1[i], nums2[j]);
 }
 while (i < m)
 {
 res[c] = nums1[i];
 i++;
 c++;
 }
 while (j < n)
 {
 res[c] = nums2[j];
 j++;
 c++;
 }
 for (i = 0; i < nums1.length; i++)
 {
 nums1[i] = res[i];
 }

 }
}
```

# 90. Subsets II ↗

Given an integer array `nums` that may contain duplicates, return *all possible subsets (the power set)*.

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

## Example 1:

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

## Example 2:

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

## Constraints:

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

```

class Solution {
 public List<List<Integer>> subsetsWithDup(int[] nums) {
 Arrays.sort(nums);
 List<List<Integer>> res = new ArrayList<>();
 generateSubsets(0, nums, new ArrayList<>(), res);
 return res;
 }
 void generateSubsets(int start, int[] nums, ArrayList<Integer> curr, List<List<Integer>> res) {
 res.add(new ArrayList<>(curr));

 for (int i = start; i < nums.length; i++) {
 if (i > start && nums[i] == nums[i-1]) continue;

 curr.add(nums[i]);
 generateSubsets(i+1, nums, curr, res);
 curr.remove(curr.size()-1);
 }
 }
}

```

## 91. Decode Ways ↗

A message containing letters from A-Z can be **encoded** into numbers using the following mapping:

|     |    |      |
|-----|----|------|
| 'A' | -> | "1"  |
| 'B' | -> | "2"  |
| ... |    |      |
| 'Z' | -> | "26" |

To **decode** an encoded message, all the digits must be grouped then mapped back into letters using the reverse of the mapping above (there may be multiple ways). For example, "11106" can be mapped into:

- "AAJF" with the grouping (1 1 10 6)
- "KJF" with the grouping (11 10 6)

Note that the grouping (1 11 06) is invalid because "06" cannot be mapped into 'F' since "6" is different from "06".

Given a string `s` containing only digits, return *the number of ways to decode it*.

The answer is guaranteed to fit in a **32-bit** integer.

### Example 1:

**Input:** `s = "12"`

**Output:** 2

**Explanation:** "12" could be decoded as "AB" (1 2) or "L" (12).

### Example 2:

**Input:** `s = "226"`

**Output:** 3

**Explanation:** "226" could be decoded as "BZ" (2 26), "VF" (22 6), or "BBF" (2

### Example 3:

**Input:** `s = "0"`

**Output:** 0

**Explanation:** There is no character that is mapped to a number starting with 0. The only valid mappings with 0 are 'J' -> "10" and 'T' -> "20", neither of which map to 0. Hence, there are no valid ways to decode this since all digits need to be mapped.

### Example 4:

**Input:** `s = "06"`

**Output:** 0

**Explanation:** "06" cannot be mapped to "F" because of the leading zero ("6" is

### Constraints:

- $1 \leq s.length \leq 100$
- `s` contains only digits and may contain leading zero(s).

This is different recursion than subset... sum me continuous nahi hota... aha continuous way me hum select kar rahai hai... islia for ki jariyat nahi

```
class Solution {

 public int numDecodings(String s) {

 HashMap<Integer, Integer> memo = new HashMap<>();

 int finalCount = decode(0, s, memo);
 return finalCount;
 }

 int decode(int start, String s, HashMap<Integer, Integer> memo)
 {
 if (start >= s.length()) {
 return 1;
 }
 if (s.charAt(start) == '0') return 0;

 if (memo.containsKey(start)) return memo.get(start);

 String oneChar = s.substring(start, start+1);

 int ways = decode(start+1, s, memo);

 if (start+2 <= s.length()) {

 String twoChar = s.substring(start, start+2);

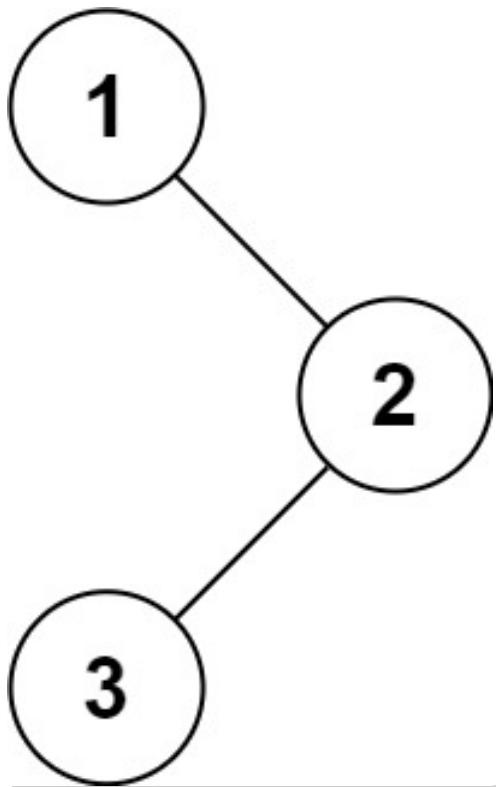
 if (Integer.parseInt(twoChar) < 27) {
 ways += decode(start+2,s, memo);
 }
 }
 memo.put(start, ways);
 return ways;
 }
}
```

## 94. Binary Tree Inorder Traversal ↗



Given the `root` of a binary tree, return *the inorder traversal of its nodes' values*.

### **Example 1:**



**Input:** root = [1,null,2,3]

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

### **Example 2:**

**Input:** root = []

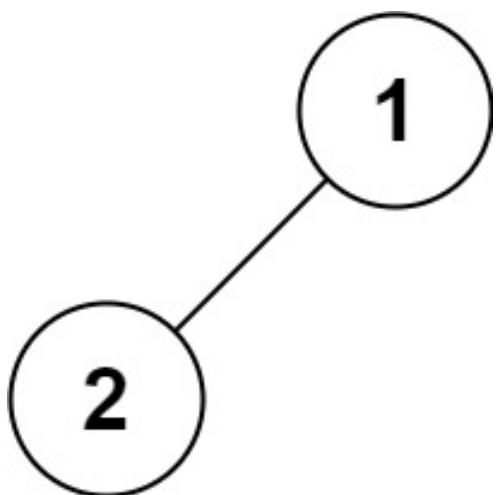
**Output:** []

### **Example 3:**

**Input:** root = [1]

**Output:** [1]

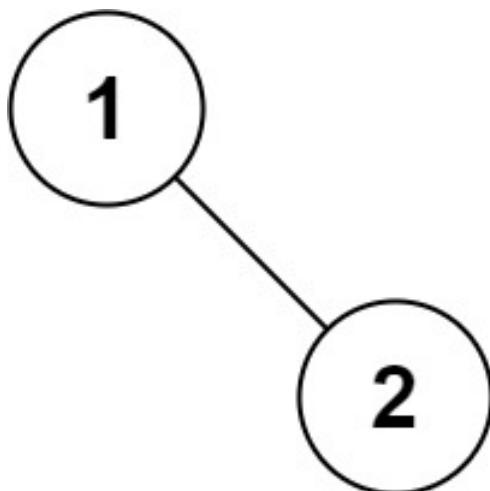
### **Example 4:**



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

**Output:** [2,1]

**Example 5:**



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

**Output:** [1,2]

**Constraints:**

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

**Follow up:** Recursive solution is trivial, could you do it iteratively?

```
class Solution {
 public List<Integer> inorderTraversal(TreeNode root)
 {
 List<Integer> list= new ArrayList<Integer>();
 if(root != null)
 {
 list.addAll(inorderTraversal(root.left));
 list.add(root.val);
 list.addAll(inorderTraversal(root.right));
 }

 return list;
 }
}
```

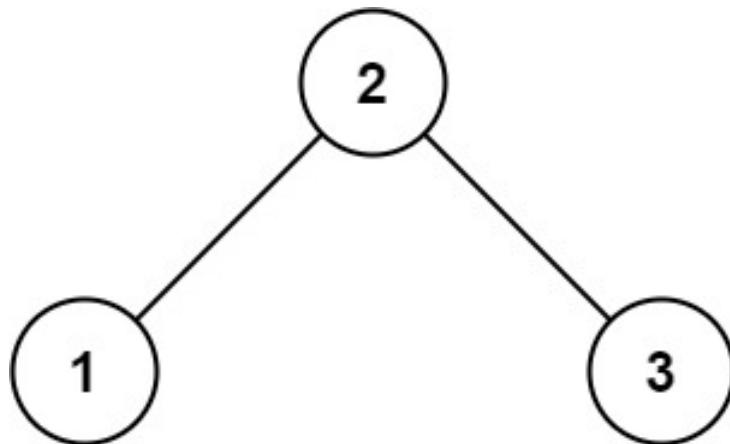
## 98. Validate Binary Search Tree ↗

Given the root of a binary tree, determine if it is a valid binary search tree (BST).

A **valid BST** is defined as follows:

- The left subtree of a node contains only nodes with keys **less than** the node's key.
- The right subtree of a node contains only nodes with keys **greater than** the node's key.
- Both the left and right subtrees must also be binary search trees.

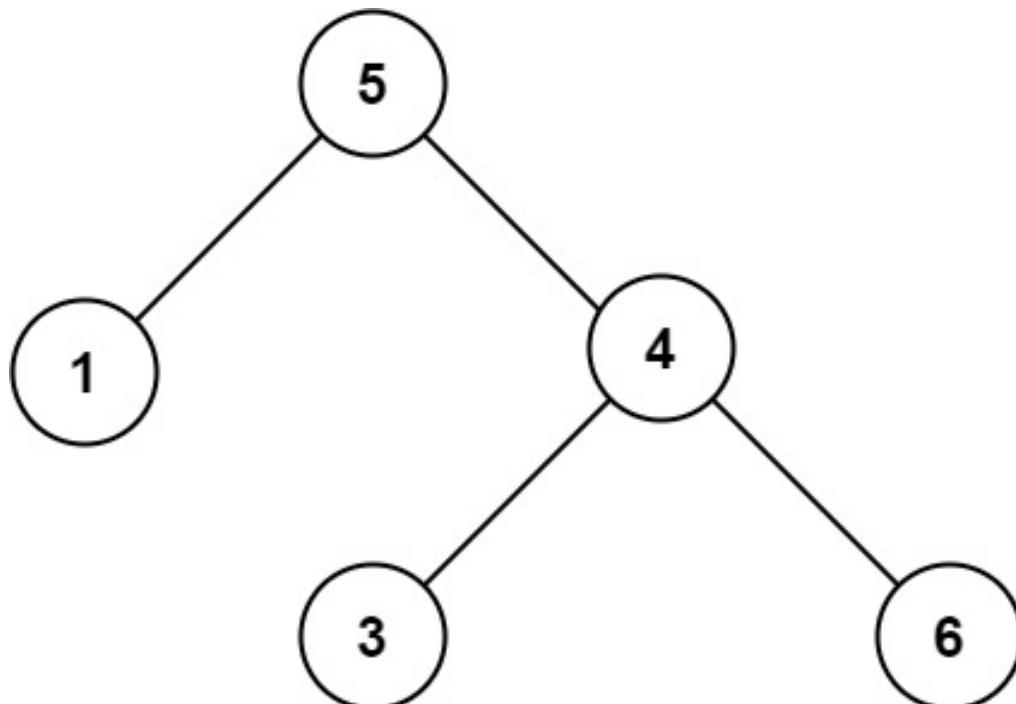
**Example 1:**



**Input:** root = [2,1,3]

**Output:** true

**Example 2:**



**Input:** root = [5,1,4,null,null,3,6]

**Output:** false

**Explanation:** The root node's value is 5 but its right child's value is 4.

### Constraints:

- The number of nodes in the tree is in the range  $[1, 10^4]$ .
- $-2^{31} \leq \text{Node.val} \leq 2^{31} - 1$

```
class Solution {
 public boolean isValidBST(TreeNode root) {

 if (root == null) return false;

 return validate (root, Long.MIN_VALUE, Long.MAX_VALUE);
 }

 public boolean validate(TreeNode root, long leftRange, long rightRange)
{
 if (root == null) return true;

 if (root.val <= leftRange || root.val >= rightRange) return false;

 boolean left = validate(root.left, leftRange, root.val);

 boolean right = validate(root.right, root.val, rightRange);

 if (left == false || right == false) return false;

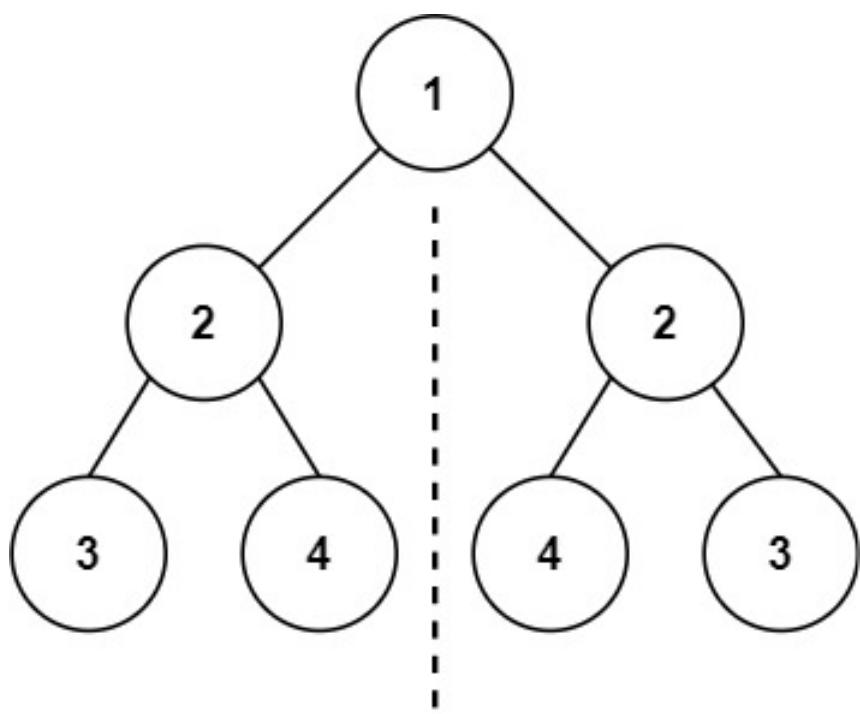
 return true;
 }
}
```

## 101. Symmetric Tree ↗



Given the `root` of a binary tree, *check whether it is a mirror of itself* (i.e., symmetric around its center).

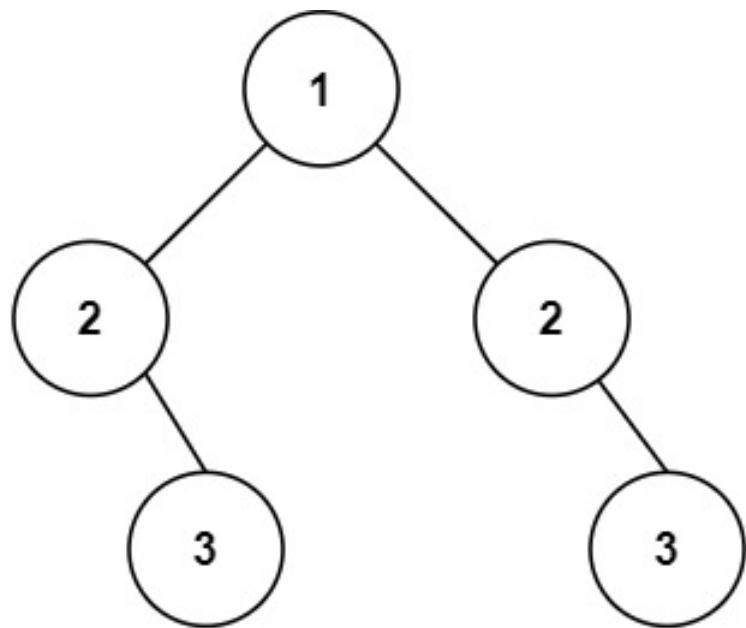
### Example 1:



**Input:** root = [1,2,2,3,4,4,3]

**Output:** true

### Example 2:



**Input:** root = [1,2,2,null,3,null,3]

**Output:** false

### Constraints:

- The number of nodes in the tree is in the range [1, 1000].
- $-100 \leq \text{Node.val} \leq 100$

**Follow up:** Could you solve it both recursively and iteratively?

---

```
/**
 * 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 boolean isSymmetric(TreeNode root)
 {
 if (root != null)
 {
 return isSymmetrichelp(root.left, root.right);
 }
 return true;

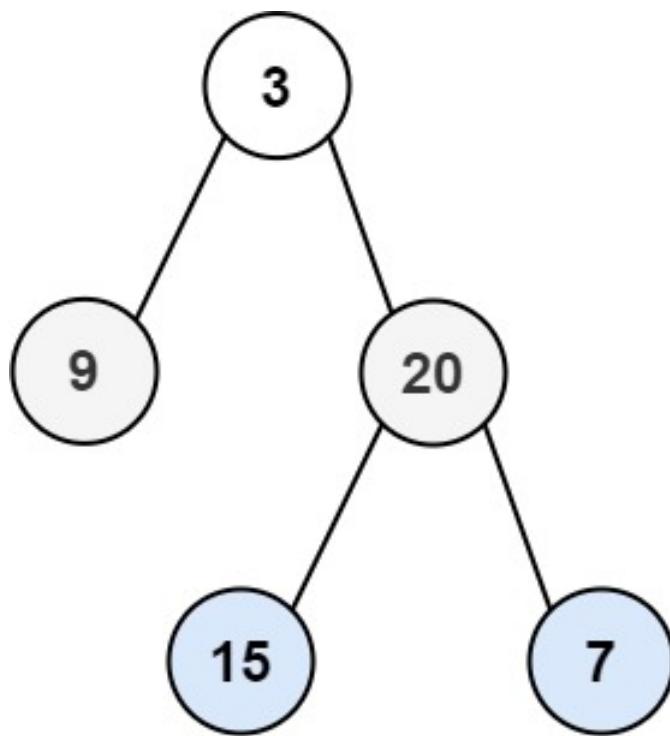
 }
 public boolean isSymmetrichelp(TreeNode left, TreeNode right)
 {
 if (left == null && right == null) return true;
 if ((left == null && right != null) || (left != null && right == null))return false;
 if (left.val == right.val)
 {
 return (isSymmetrichelp(left.left, right.right)) && (isSymmetrichelp(left.right, right.left));
 }

 return false;
 }
}
```

# 102. Binary Tree Level Order Traversal ↗

Given the `root` of a binary tree, return *the level order traversal of its nodes' values*. (i.e., from left to right, level by level).

## Example 1:



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

**Output:** [[3],[9,20],[15,7]]

## Example 2:

**Input:** root = [1]

**Output:** [[1]]

## Example 3:

**Input:** root = []

**Output:** []

## Constraints:

- The number of nodes in the tree is in the range  $[0, 2000]$ .
- $-1000 \leq \text{Node.val} \leq 1000$

```

class Solution {
 public List<List<Integer>> levelOrder(TreeNode root) {

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

 Queue<TreeNode> q = new LinkedList<TreeNode>();
 ArrayList<Integer> al = new ArrayList<Integer>();

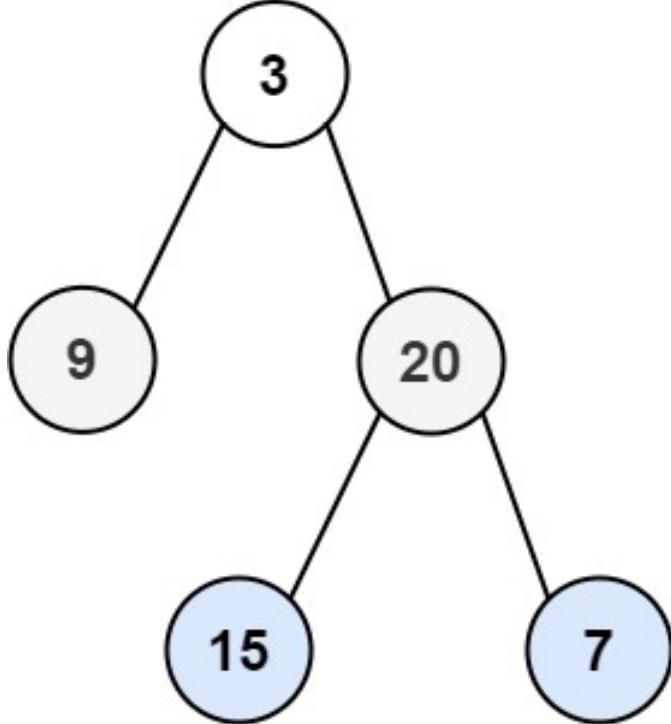
 if (root != null)
 {
 q.add(root);
 while (!q.isEmpty())
 {
 //System.out.print(q);
 int size = q.size();
 //TreeNode n = null;
 for (int i = 0; i < size; i++)
 {
 TreeNode n = q.remove();
 al.add (n.val);
 if (n.left != null) q.add(n.left);
 if (n.right != null) q.add(n.right);
 }
 list.add (al);
 al = new ArrayList<Integer>();
 }
 }
 return list;
 }
}

```

## 103. Binary Tree Zigzag Level Order Traversal ↗

Given the `root` of a binary tree, return *the zigzag level order traversal of its nodes' values*. (i.e., from left to right, then right to left for the next level and alternate between).

**Example 1:**



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

**Output:** [[[3],[20,9],[15,7]]]

### Example 2:

**Input:** root = [1]

**Output:** [[1]]

### Example 3:

**Input:** root = []

**Output:** []

### Constraints:

- The number of nodes in the tree is in the range [0, 2000].
- $-100 \leq \text{Node.val} \leq 100$

```
class Solution {
 public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
 List<List<Integer>> ans = new ArrayList<>();

 if (root == null) return ans;

 List<Integer> midList = new ArrayList<>();
 Deque<TreeNode> dq = new LinkedList<TreeNode>();

 boolean putRight = true;

 dq.addFirst(root);

 while (!dq.isEmpty())
 {
 int size = dq.size();
 midList = new ArrayList<>();
 for (int i = 0; i < size; i++)
 {
 TreeNode node = null;
 if (!putRight)
 {
 node = dq.removeLast();
 if (node.right != null) dq.addFirst(node.right);
 if (node.left != null) dq.addFirst(node.left);
 }
 if (putRight)
 {
 node = dq.removeFirst();
 if (node.left != null) dq.addLast(node.left);
 if (node.right != null) dq.addLast(node.right);
 }
 midList.add(node.val);
 }

 ans.add(midList);
 putRight = !putRight;
 }
 return ans;
 }
}
```

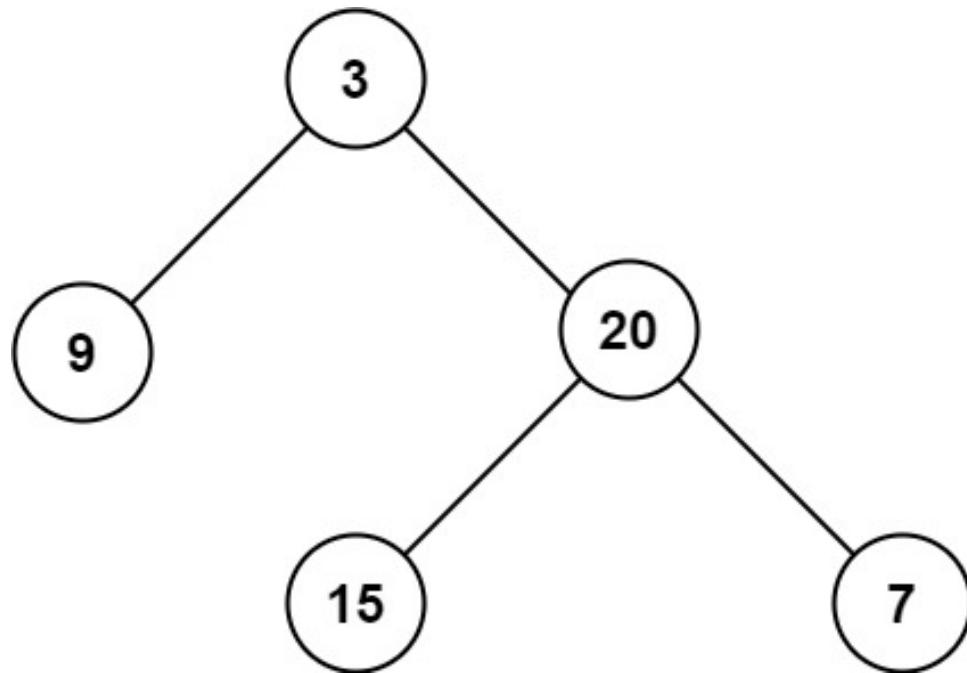
# 104. Maximum Depth of Binary Tree ↗



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

## Example 3:

**Input:** root = []

**Output:** 0

## Example 4:

**Input:** root = [0]

**Output:** 1

## Constraints:

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

```
class Solution {
 public int maxDepth(TreeNode root) {
 //int ans = findmaxDepth(root, 0);
 if (root != null)
 {
 return (1 + Math.max(maxDepth(root.left), maxDepth(root.right)));
 //return ans;
 }
 return 0;
 }
 public static int findmaxDepth(TreeNode node, int depth)
 {
 if (node != null)
 {
 depth = depth + 1;

 depth = Math.max(findmaxDepth(node.left, depth), findmaxDept
h(node.right, depth));
 }

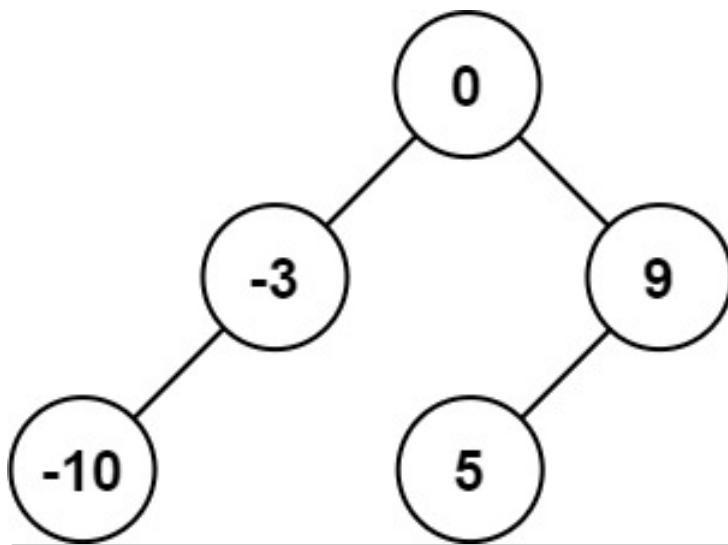
 return depth;
 }
}
```

## 108. Convert Sorted Array to Binary Search Tree

Given an integer array `nums` where the elements are sorted in **ascending order**, convert *it to a height-balanced binary search tree*.

A **height-balanced** binary tree is a binary tree in which the depth of the two subtrees of every node never differs by more than one.

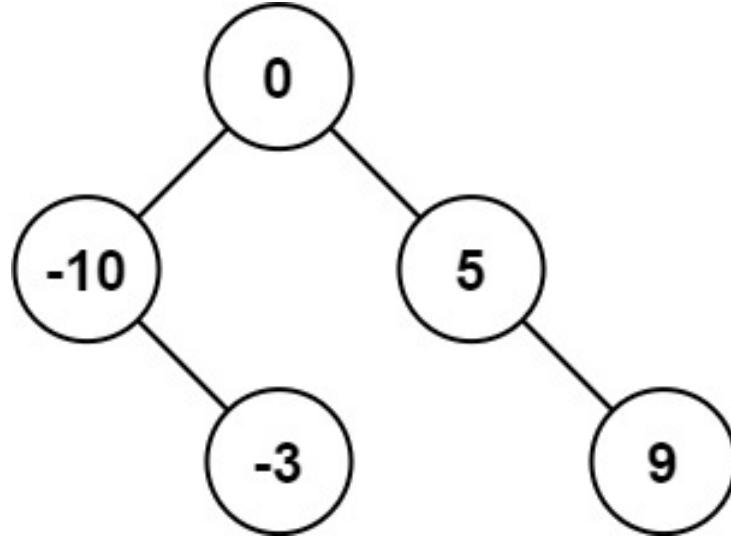
### Example 1:



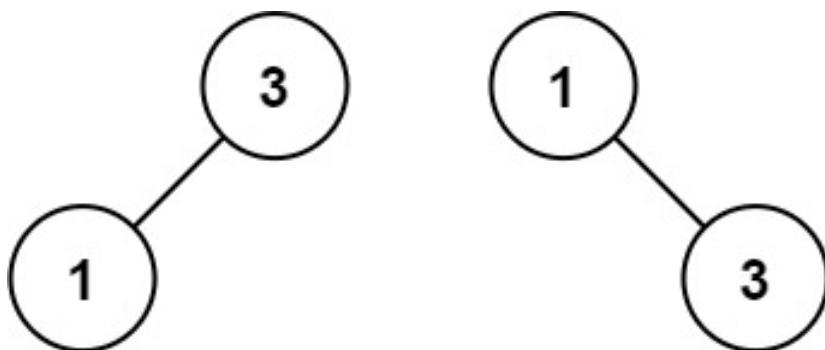
**Input:** nums = [-10, -3, 0, 5, 9]

**Output:** [0, -3, 9, -10, null, 5]

**Explanation:** [0, -10, 5, null, -3, null, 9] is also accepted:



### Example 2:



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

**Output:** [3,1]

**Explanation:** [1,3] and [3,1] are both a height-balanced BSTs.

### Constraints:

- $1 \leq \text{nums.length} \leq 10^4$
- $-10^4 \leq \text{nums}[i] \leq 10^4$
- `nums` is sorted in a **strictly increasing** order.

```
class Solution {
 public TreeNode sortedArrayToBST(int[] nums)
 {
 return construct(nums, 0, nums.length-1);

 }

 public TreeNode construct(int[] nums, int lo, int hi)
 {
 if (lo > hi) return null;

 int mid = lo + (hi - lo)/2;

 TreeNode node = new TreeNode (nums[mid]);

 node.left = construct(nums, lo, mid-1);
 node.right = construct(nums, mid +1, hi);

 return node;
 }
}
```

## 116. Populating Next Right Pointers in Each Node ↗

You are given a **perfect binary tree** where all leaves are on the same level, and every parent has two children. The binary tree has the following definition:

```
struct Node {
 int val;
 Node *left;
 Node *right;
 Node *next;
}
```

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to `NULL`.

Initially, all next pointers are set to `NULL`.

### Example 1:

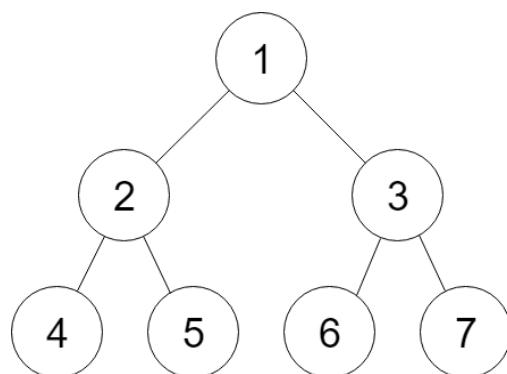


Figure A

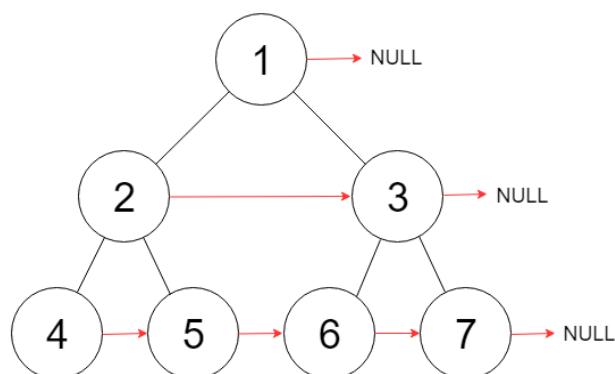


Figure B

**Input:** `root = [1,2,3,4,5,6,7]`

**Output:** `[1,#,2,3,#,4,5,6,7,#]`

**Explanation:** Given the above perfect binary tree (Figure A), your function sh

### Example 2:

**Input:** `root = []`

**Output:** `[]`

### Constraints:

- The number of nodes in the tree is in the range  $[0, 2^{12} - 1]$ .
- $-1000 \leq \text{Node.val} \leq 1000$

### Follow-up:

- You may only use constant extra space.
- The recursive approach is fine. You may assume implicit stack space does not count as extra space for this problem.

```

class Solution {
 public Node connect(Node root) {
 if (root == null) return null;
 Queue<Node> q = new LinkedList<Node>();

 q.add(root); //add root
 // root.next = null;
 while (!q.isEmpty())
 {
 int size = q.size();
 Node n = null;
 for (int i = 0; i < size; i++)
 {
 n = q.remove();
 n.next = q.peek(); //main idea think

 if (n.left != null) q.add(n.left);
 if (n.right != null) q.add(n.right);
 }
 n.next = null;
 }
 return root;
 }
}

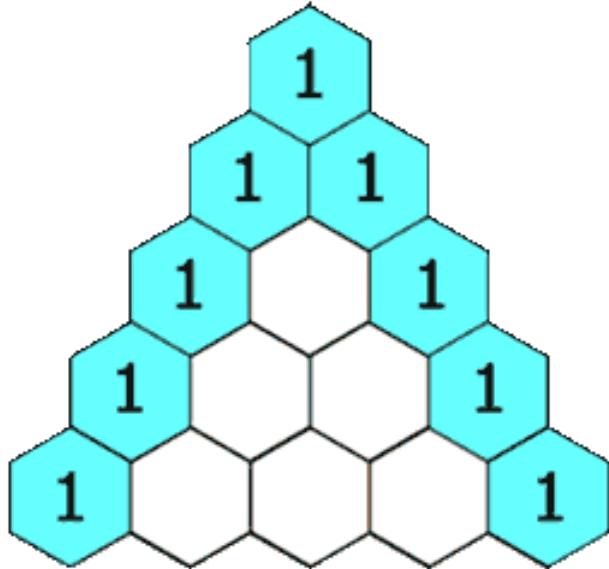
```

## 118. Pascal's Triangle ↗



Given an integer `numRows`, return the first `numRows` of **Pascal's triangle**.

In **Pascal's triangle**, each number is the sum of the two numbers directly above it as shown:



### Example 1:

**Input:** numRows = 5

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

### Example 2:

**Input:** numRows = 1

**Output:** [[1]]

### Constraints:

- $1 \leq \text{numRows} \leq 30$

```

class Solution {
 public List<List<Integer>> generate(int numRows)
 {
 List<List<Integer>> res = new ArrayList<List<Integer>>();
 List<Integer> prev = new ArrayList<Integer>();

 if (numRows == 0) return res;

 prev.add(1);
 res.add(prev);
 for (int i = 1; i < numRows; i++)
 {
 ArrayList<Integer> temp = new ArrayList<Integer>();
 for (int j = 0; j <= i; j++)
 {
 if (j != 0 && j!= i)
 {
 temp.add((prev.get(j-1) + prev.get(j)));
 }
 else
 {
 temp.add(1);
 }
 }
 res.add(temp);
 prev = temp;
 }

 return res;
 }
}

```

## 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:**

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

```

class Solution {
 public int maxProfit(int[] prices) {

 int[] max = new int[prices.length];
 int[] min = new int[prices.length];
 int minP = prices[0];
 int maxP = prices[prices.length-1];

 int result = 0;

 for (int i = 0; i < prices.length; i++)
 {
 min[i] = Math.min(minP, prices[i]);
 minP = min[i];
 // System.out.print(min[i]);
 }
 // System.out.println();
 for (int i = prices.length-1; i >= 0; i--)
 {
 max[i] = Math.max(maxP, prices[i]);
 maxP = max[i];
 // System.out.print(max[i]);
 }
 for (int i = 0; i < prices.length; i++)
 {
 int diff = max[i] - min[i];
 result = Math.max(result, diff);
 }

 return result;
 }
}

```

## 122. Best Time to Buy and Sell Stock II ↗ ▾

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

On each day, you may decide to buy and/or sell the stock. You can only hold **at most one** share of the stock at any time. However, you can buy it then immediately sell it on the **same day**.

Find and return *the maximum profit you can achieve*.

### **Example 1:**

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

**Output:** 7

**Explanation:** Buy on day 2 (price = 1) and sell on day 3 (price = 5), profit = 5 - 1 = 4  
Then buy on day 4 (price = 3) and sell on day 5 (price = 6), profit = 6 - 3 = 3  
Total profit is 4 + 3 = 7.

### **Example 2:**

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

**Output:** 4

**Explanation:** Buy on day 1 (price = 1) and sell on day 5 (price = 5), profit = 5 - 1 = 4  
Total profit is 4.

### **Example 3:**

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

**Output:** 0

**Explanation:** There is no way to make a positive profit, so we never buy the stock.

### **Constraints:**

- $1 \leq \text{prices.length} \leq 3 * 10^4$
- $0 \leq \text{prices}[i] \leq 10^4$

```
class Solution {
 public int maxProfit(int[] prices) {
 int profit = 0;

 for (int i = 1; i < prices.length; i++)
 {
 if(prices [i] > prices [i-1])
 {
 profit = profit + (prices [i] - prices [i-1]);
 }
 }
 return profit;
 }
}
```

## 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.

```
class Solution {
 public boolean isPalindrome(String s) {

 s = s.replaceAll("[^a-zA-Z0-9]", "");

 int j = s.length()-1;
 s = s.toLowerCase();
 for(int i = 0; i < s.length()/2; i++)
 {
 if(s.charAt(i) != s.charAt(j))
 {
 return false;
 }
 j--;
 }

 return true;
 }
}
```

## Second Approach

```

class Solution {
 public boolean isPalindrome(String s) {

 s = s.replaceAll("[^a-zA-Z0-9]", "");

 int j = s.length()-1;
 s = s.toLowerCase();
 for(int i = 0; i < s.length()/2; i++)
 {
 if(s.charAt(i) != s.charAt(j))
 {
 return false;
 }
 j--;
 }

 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]. There

### 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)
 {
 Set<Integer> s = new HashSet<>();
 for (int i: nums) s.add(i);

 int maxCount = 0;
 for (int i = 0; i < nums.length; i++)
 {
 int count = 0;
 //means is number se chota numerber ab nahi h set me to ye star
ting point
 if (!s.contains(nums[i] -1))
 {
 int num = nums[i];
 while(s.contains(num)) //her baar hum num+1 karte ja rai de
khna h kb tk consequutive hogा
 {
 count++;
 num += 1; //num = num+1
 }
 maxCount = Math.max(maxCount, count);
 }
 }
 return maxCount;
 }
}
```

```

class Solution {
 public int longestConsecutive(int[] nums)
 {
 //corner case
 if (nums == null || nums.length == 0) return 0;
 else if (nums.length == 1) return 1;

 Arrays.sort(nums);

 int c = 1; //ek element to pakka hai he islia 1 se start
 int max = Integer.MIN_VALUE;
 for (int i = 0; i < nums.length-1; i++)
 {
 if (nums[i] +1 == nums[i+1])c++;
 else if (nums[i] == nums[i+1]) continue; // [2,0,1,1] ignore duplicate
 else c = 1;

 max = Math.max(max, c);
 }
 return (max==Integer.MIN_VALUE)?1:max;
 }
}

```

## 130. Surrounded Regions ↗

Given an  $m \times n$  matrix board containing 'X' and 'O', capture all regions that are 4-directionally surrounded by 'X'.

A region is **captured** by flipping all 'O' s into 'X' s in that surrounded region.

### Example 1:

|   |   |   |   |
|---|---|---|---|
| X | X | X | X |
| X | O | O | X |
| X | X | O | X |
| X | O | X | X |

|   |   |   |   |
|---|---|---|---|
| X | X | X | X |
| X | X | X | X |
| X | X | X | X |
| X | O | X | X |

**Input:** board = [["X", "X", "X", "X"], ["X", "O", "O", "X"], ["X", "X", "O", "X"], ["X", "O", "X", "X"]]

**Output:** [[ "X", "X", "X", "X"], [ "X", "X", "X", "X"], [ "X", "X", "X", "X"], [ "X", "O", "X", "X"]]

**Explanation:** Surrounded regions should not be on the border, which means that

### Example 2:

**Input:** board = [[ "X" ]]

**Output:** [[ "X" ]]

### Constraints:

- $m == \text{board.length}$
- $n == \text{board}[i].length$
- $1 \leq m, n \leq 200$
- $\text{board}[i][j]$  is 'X' or 'O' .

```

class Solution {
 public void solve(char[][] board)
 {
 //sides me traverse karege or dekhege ki kitan fail skta h..
 //failne wale ko 'A' mark kar deneg
 //baad me fir traverse kar k 'A' ko '0' karege or existing '0' ko
 'X'

 boolean[][] visited = new boolean[board.length][board[0].length];

 //upper ki row(row = 0, col will change)
 for (int i = 0; i < board.length; i++)
 {
 solveHelp(board, 0, i, visited);
 }
 //left side col(col = 0, row will change)
 for (int i = 0; i < board.length; i++)
 {
 solveHelp(board, i, 0, visited);
 }
 //right side col(col = board[0].length-1, row will change)
 for (int i = 0; i < board.length; i++)
 {
 solveHelp(board, i, board[0].length-1, visited);
 }
 //last row(row = board[0].length-1, col will change)
 for (int i = 0; i < board[0].length-1; i++)
 {
 solveHelp(board, board.length-1, i, visited);
 }
 //'A' ko '0' karege or existing '0' ko 'X'
 for (int i = 0; i < board.length; i++)
 {
 for (int j = 0; j < board[0].length; j++)
 {
 if (board[i][j] == 'A') board[i][j] = '0';
 else if (board[i][j] == '0') board[i][j] = 'X';
 }
 }
 }

 public void solveHelp(char[][] board, int a, int b, boolean[][] visite
d)
{
 //board k baher, visited, ya 'X' to return
 if (a < 0 || a >= board.length || b < 0 || b >= board[0].length ||

```

```

 visited[a][b] == true || board[a][b] == 'X')
 {
 return;
 }
 board[a][b] = 'A';
 visited[a][b] = true;
 solveHelp(board, a, b+1, visited); //right
 solveHelp(board, a, b-1, visited); //left
 solveHelp(board, a-1, b, visited); //up
 solveHelp(board, a+1, b, visited); //down
}
}

```

## 131. Palindrome Partitioning ↗

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

A **palindrome** string is a string that reads the same backward as forward.

### Example 1:

```

Input: s = "aab"
Output: [["a", "a", "b"], ["aa", "b"]]

```

### Example 2:

```

Input: s = "a"
Output: [["a"]]

```

### Constraints:

- $1 \leq s.length \leq 16$
- $s$  contains only lowercase English letters.

```

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

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

 pallindrome(s, 0, new ArrayList<String>(), res);
 return res;
 }

 void pallindrome(String s, int start, ArrayList<String> currList, List<List<String>> res) {

 if (start == s.length()) {
 res.add(new ArrayList<String>(currList));
 }

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

 String prefix = s.substring(0, i+1);
 String restString = s.substring(i+1);

 if (isPallindrome(prefix)) {
 currList.add(prefix);
 pallindrome(restString, 0, currList, res); //start is 0 coz
//everytime its a new string
 currList.remove(currList.size()-1); //backtrack
 }
 }
 }

 boolean isPallindrome(String s) {
 int start = 0, end = s.length()-1;

 while (start < end) {
 if (s.charAt(start) != s.charAt(end)) return false;

 start++;
 end--;
 }
 return true;
 }
}

```

# 133. Clone Graph ↗

Given a reference of a node in a **connected**

([https://en.wikipedia.org/wiki/Connectivity\\_\(graph\\_theory\)#Connected\\_graph](https://en.wikipedia.org/wiki/Connectivity_(graph_theory)#Connected_graph)) undirected graph.

Return a **deep copy** ([https://en.wikipedia.org/wiki/Object\\_copying#Deep\\_copy](https://en.wikipedia.org/wiki/Object_copying#Deep_copy)) (clone) of the graph.

Each node in the graph contains a value ( `int` ) and a list ( `List[Node]` ) of its neighbors.

```
class Node {
 public int val;
 public List<Node> neighbors;
}
```

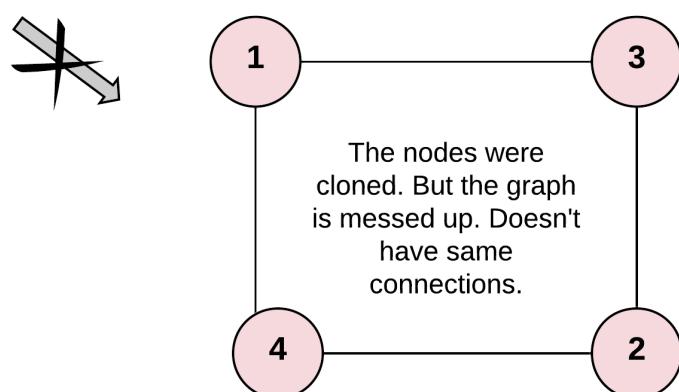
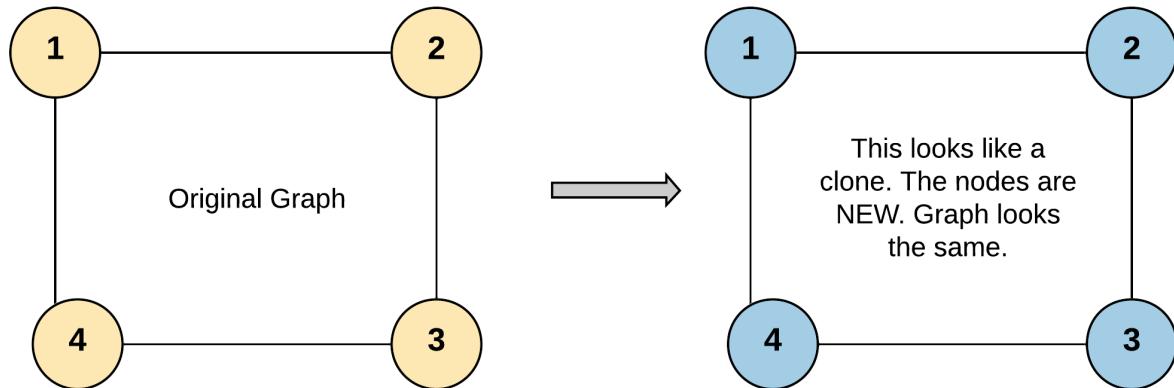
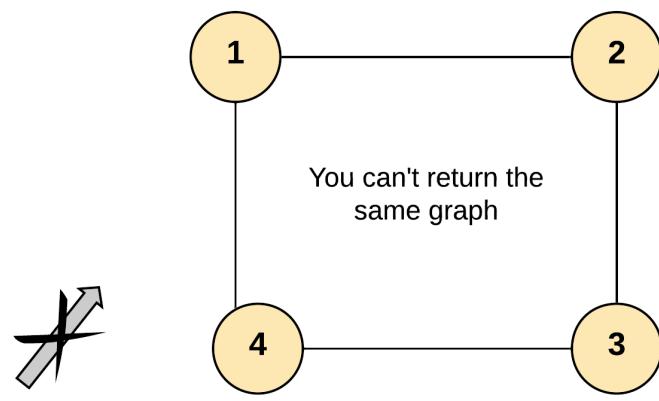
## Test case format:

For simplicity, each node's value is the same as the node's index (1-indexed). For example, the first node with `val == 1`, the second node with `val == 2`, and so on. The graph is represented in the test case using an adjacency list.

An **adjacency list** is a collection of unordered **lists** used to represent a finite graph. Each list describes the set of neighbors of a node in the graph.

The given node will always be the first node with `val = 1`. You must return the **copy of the given node** as a reference to the cloned graph.

## Example 1:



**Input:** adjList = [[2,4],[1,3],[2,4],[1,3]]

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

**Explanation:** There are 4 nodes in the graph.

1st node (val = 1)'s neighbors are 2nd node (val = 2) and 4th node (val = 4).

2nd node (val = 2)'s neighbors are 1st node (val = 1) and 3rd node (val = 3).

3rd node (val = 3)'s neighbors are 2nd node (val = 2) and 4th node (val = 4).

4th node (val = 4)'s neighbors are 1st node (val = 1) and 3rd node (val = 3).

**Example 2:**



**Input:** adjList = [[]]

**Output:** []

**Explanation:** Note that the input contains one empty list. The graph consists

### Example 3:

**Input:** adjList = []

**Output:** []

**Explanation:** This is an empty graph, it does not have any nodes.

### Example 4:



**Input:** adjList = [[2],[1]]

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

### Constraints:

- The number of nodes in the graph is in the range [0, 100].
- $1 \leq \text{Node.val} \leq 100$
- `Node.val` is unique for each node.
- There are no repeated edges and no self-loops in the graph.
- The Graph is connected and all nodes can be visited starting from the given node.

//mapping bahut confusion

```

/*
// Definition for a Node.
class Node {
 public int val;
 public List<Node> neighbors;
 public Node() {
 val = 0;
 neighbors = new ArrayList<Node>();
 }
 public Node(int _val) {
 val = _val;
 neighbors = new ArrayList<Node>();
 }
 public Node(int _val, ArrayList<Node> _neighbors) {
 val = _val;
 neighbors = _neighbors;
 }
}
*/
//visited ki jarurat nahi map to hai he
class Solution {
 public Node cloneGraph(Node node) {

 if (node == null) return null;

 Queue<Node> queue = new LinkedList<>();
 HashMap<Integer, Node> map = new HashMap<>();
 queue.add(node);

 map.put(node.val, new Node(node.val));

 while (!queue.isEmpty()) {

 Node curr = queue.remove(); //parent node

 for (Node neighbour : curr.neighbors) { //exploring child/neigh
 neighbour
 if (!map.containsKey(neighbour.val)) {
 map.put(neighbour.val, new Node(neighbour.val)); //firs
 t time ye dikha
 queue.add(neighbour);
 }
 map.get(curr.val).neighbors.add(map.get(neighbour.val)); ////
curr node k neighbour me bhi add karte chalo
 }
 }
}

```

```
 }

}

return map.get(node.val);

}

}
```

## 136. Single Number ↗

Given a **non-empty** array of integers `nums`, every element appears *twice* except for one. Find that single one.

You must implement a solution with a linear runtime complexity and use only constant extra space.

### Example 1:

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

### Example 2:

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

### Example 3:

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

### Constraints:

- $1 \leq \text{nums.length} \leq 3 * 10^4$
- $-3 * 10^4 \leq \text{nums}[i] \leq 3 * 10^4$
- Each element in the array appears twice except for one element which appears only once.

## \* XOR used

```
class Solution {
 public int singleNumber(int[] nums) {

 int a = 0;
 for(int i : nums)
 a = a^i;

 return a;

 }
}
```

## 138. Copy List with Random Pointer ↗

A linked list of length `n` is given such that each node contains an additional random pointer, which could point to any node in the list, or `null`.

Construct a **deep copy** ([https://en.wikipedia.org/wiki/Object\\_copying#Deep\\_copy](https://en.wikipedia.org/wiki/Object_copying#Deep_copy)) of the list. The deep copy should consist of exactly `n` **brand new** nodes, where each new node has its value set to the value of its corresponding original node. Both the `next` and `random` pointer of the new nodes should point to new nodes in the copied list such that the pointers in the original list and copied list represent the same list state. **None of the pointers in the new list should point to nodes in the original list.**

For example, if there are two nodes `X` and `Y` in the original list, where `X.random --> Y`, then for the corresponding two nodes `x` and `y` in the copied list, `x.random --> y`.

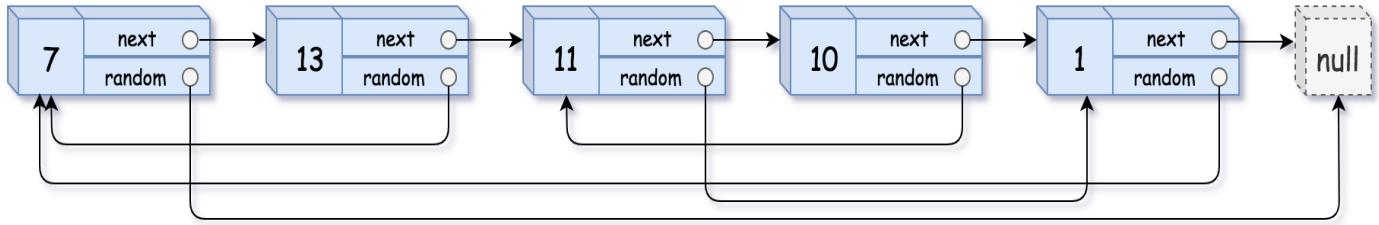
Return *the head of the copied linked list*.

The linked list is represented in the input/output as a list of `n` nodes. Each node is represented as a pair of `[val, random_index]` where:

- `val` : an integer representing `Node.val`
- `random_index` : the index of the node (range from `0` to `n-1`) that the `random` pointer points to, or `null` if it does not point to any node.

Your code will **only** be given the `head` of the original linked list.

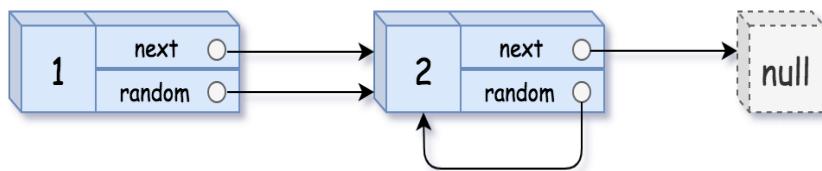
**Example 1:**



**Input:** head = [[7,null],[13,0],[11,4],[10,2],[1,0]]

**Output:** [[7,null],[13,0],[11,4],[10,2],[1,0]]

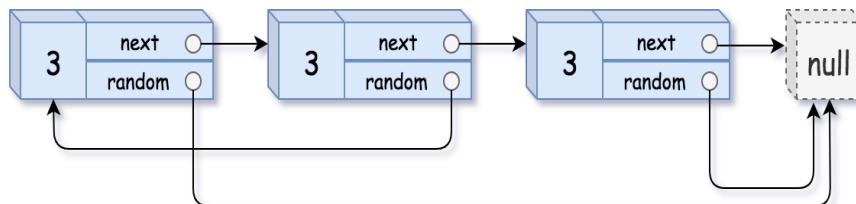
### Example 2:



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

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

### Example 3:



**Input:** head = [[3,null],[3,0],[3,null]]

**Output:** [[3,null],[3,0],[3,null]]

### Example 4:

**Input:** head = []

**Output:** []

**Explanation:** The given linked list is empty (null pointer), so return null.

### Constraints:

- $0 \leq n \leq 1000$
- $-10000 \leq \text{Node.val} \leq 10000$

- `Node.random` is `null` or is pointing to some node in the linked list.

```
class Solution {
 public Node copyRandomList(Node head) {

 HashMap<Node,Node> hm = new HashMap<Node,Node>();
 Node currNode = head;

 while (currNode != null)//creating a copy in Hashmap
 {
 hm.put(currNode, new Node(currNode.val));
 currNode = currNode.next;
 }
 currNode = head;

 while (currNode != null)
 {
 hm.get(currNode).next = hm.get(currNode.next);
 hm.get(currNode).random = hm.get(currNode.random);
 currNode = currNode.next;
 }
 Node cloneHead = hm.get(head);
 return cloneHead;

 }
}
```

## 139. Word Break ↗

Given a string `s` and a dictionary of strings `wordDict`, return `true` if `s` can be segmented into a space-separated sequence of one or more dictionary words.

**Note** that the same word in the dictionary may be reused multiple times in the segmentation.

**Example 1:**

**Input:** s = "leetcode", wordDict = ["leet", "code"]

**Output:** true

**Explanation:** Return true because "leetcode" can be segmented as "leet code".

### Example 2:

**Input:** s = "applepenapple", wordDict = ["apple", "pen"]

**Output:** true

**Explanation:** Return true because "applepenapple" can be segmented as "apple p

Note that you are allowed to reuse a dictionary word.

### Example 3:

**Input:** s = "catsandog", wordDict = ["cats", "dog", "sand", "and", "cat"]

**Output:** false

### Constraints:

- $1 \leq s.length \leq 300$
- $1 \leq \text{wordDict.length} \leq 1000$
- $1 \leq \text{wordDict}[i].length \leq 20$
- s and wordDict[i] consist of only lowercase English letters.
- All the strings of wordDict are **unique**.

```
class Solution { public boolean wordBreak(String s, List wordDict) { Set set = new HashSet<>(); for (String se : wordDict) set.add(se); System.out.print(set); return generateSubsets(s, set, new HashMap<String, Boolean>()); }
```

```

private boolean generateSubsets(String s, Set<String> set, HashMap<String, Boolean> memo) {

 if (memo.containsKey(s)) return memo.get(s);

 if (set.contains(s)) return true;

 for (int i = 1; i < s.length(); i++) {//we got our first word

 if (set.contains(s.substring(0, i))){

 if (generateSubsets(s.substring(i, s.length()), set, memo)) {

 memo.put(s, true);
 return true;
 }
 }
 }
 memo.put(s, false);
 return false;
}

}

```

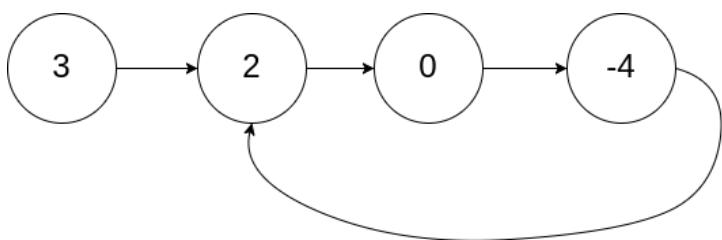
## 141. Linked List Cycle ↗

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:**

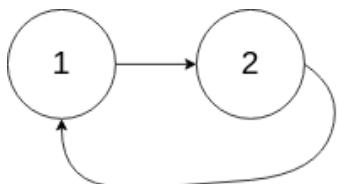


**Input:** head = [3,2,0,-4], pos = 1

**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 slownode = head;
 ListNode fastnode = head;

 while (fastnode != null && fastnode.next != null)
 {
 fastnode = fastnode.next.next; //moving by 2
 slownode = slownode.next; //moving by 1

 if(slownode == fastnode)
 {
 return true;
 }
 }

 return false;
}
```

}

## 142. Linked List Cycle II ↗

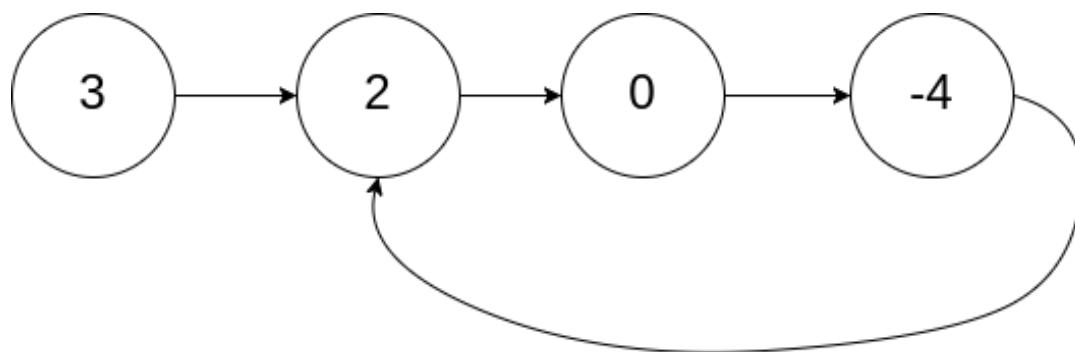


Given the `head` of a linked list, return *the node where the cycle begins. If there is no cycle, return null*.

There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the `next` pointer. Internally, `pos` is used to denote the index of the node that tail's `next` pointer is connected to (**0-indexed**). It is `-1` if there is no cycle. **Note that pos is not passed as a parameter.**

**Do not modify** the linked list.

### Example 1:

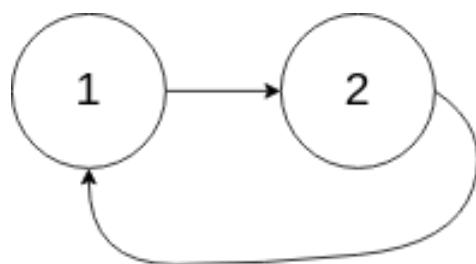


**Input:** head = [3,2,0,-4], pos = 1

**Output:** tail connects to node index 1

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

### Example 2:



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

**Output:** tail connects to node index 0

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

### Example 3:



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

**Output:** no cycle

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

### Constraints:

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

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

```
public class Solution {
 public ListNode detectCycle(ListNode head)
 {
 ListNode slownode = head;
 ListNode fastnode = head;

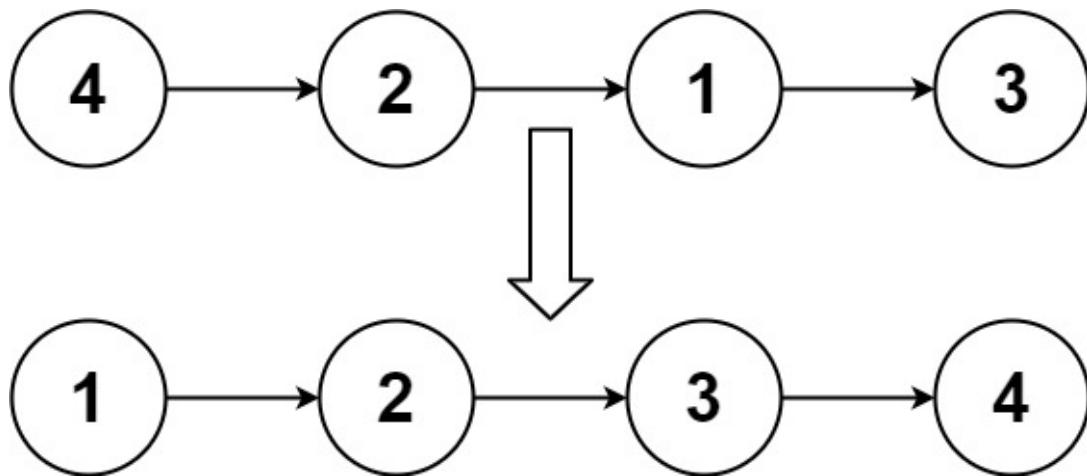
 while (fastnode != null && fastnode.next != null)
 {
 fastnode = fastnode.next.next; //moving by 2
 slownode = slownode.next; //moving by 1

 if(slownode == fastnode)//cycle detected
 {
 ListNode slownode2 = head;
 while (slownode != slownode2)
 {
 slownode2 = slownode2.next;
 slownode = slownode.next;
 }
 return slownode;
 }
 }
 return null;
 }
}
```

# 148. Sort List ↗

Given the head of a linked list, return *the list after sorting it in ascending order*.

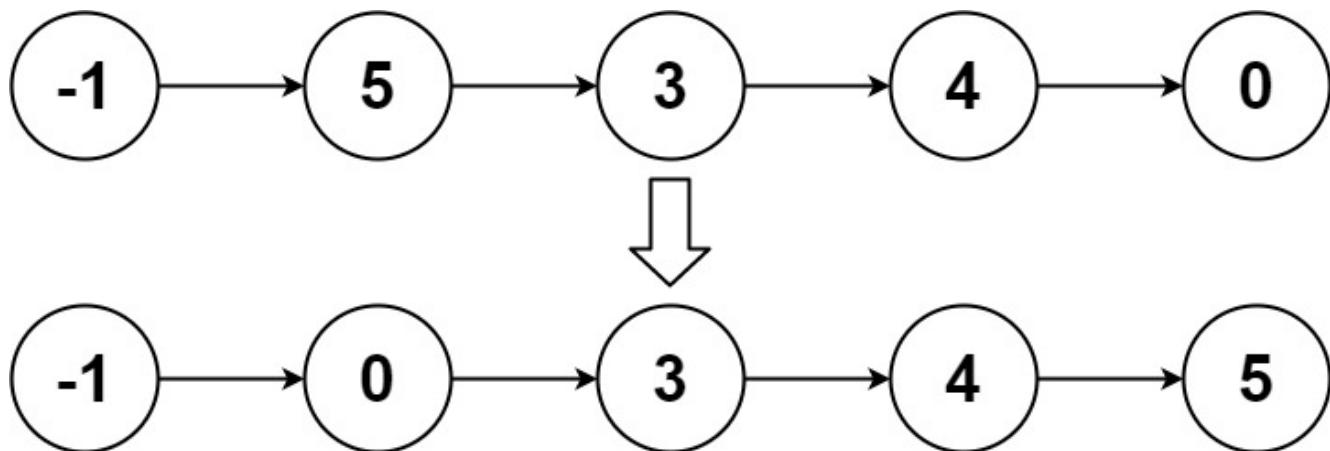
## Example 1:



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

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

## Example 2:



**Input:** head = [-1,5,3,4,0]

**Output:** [-1,0,3,4,5]

## Example 3:

**Input:** head = []

**Output:** []

## Constraints:

- The number of nodes in the list is in the range  $[0, 5 * 10^4]$  .
- $-10^5 \leq \text{Node.val} \leq 10^5$

**Follow up:** Can you sort the linked list in  $O(n \log n)$  time and  $O(1)$  memory (i.e. constant space)?

---

```

/**
 * 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 sortList(ListNode head) {

 if (head == null || head.next == null) return head; //single node or no node

 ListNode slow = head; //after while ends slow will be start of 1st half
 ListNode fast = head; //after while ends fast will be end of 2nd half
 if
 ListNode temp = head; //after while ends temp will be end of 1t half
 f

 while (fast != null && fast.next != null) {
 temp = slow;
 slow = slow.next;
 fast = fast.next.next;
 }
 temp.next = null;//break list
 ListNode left_side = sortList(head); //sort left side
 ListNode right_side = sortList(slow); //sort right side

 ListNode sorted_list = sortTwoList(left_side, right_side); //merge two sorted list
 return sorted_list;

 }

 private ListNode sortTwoList(ListNode n1, ListNode n2) {

 ListNode sortedHead = new ListNode(-1);
 ListNode currNode = sortedHead;

 while (n1 != null && n2 != null) {

```

```

 if (n1.val < n2.val){
 currNode.next = n1;
 n1 = n1.next;
 }
 else {
 currNode.next = n2;
 n2 = n2.next;
 }
 currNode = currNode.next;
 }
 if (n1 != null) {
 currNode.next = n1;
 n1 = n1.next;
 }
 if (n2 != null) {
 currNode.next = n2;
 n2 = n2.next;
 }
 return sortedHead.next;
}
}

```

## 150. Evaluate Reverse Polish Notation ↗

Evaluate the value of an arithmetic expression in Reverse Polish Notation ([http://en.wikipedia.org/wiki/Reverse\\_Polish\\_notation](http://en.wikipedia.org/wiki/Reverse_Polish_notation)).

Valid operators are `+`, `-`, `*`, and `/`. Each operand may be an integer or another expression.

**Note** that division between two integers should truncate toward zero.

It is guaranteed that the given RPN expression is always valid. That means the expression would always evaluate to a result, and there will not be any division by zero operation.

### 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].

```
class Solution {
 public int evalRPN(String[] tokens) {

 int l = tokens.length;

 Stack<Integer> st = new Stack<Integer>();
 int ans;
 int number2;
 int number1;

 for (int i = 0; i < l; i++)
 {
 switch (tokens[i])
 {
 case "+":
 number2= st.pop();
 number1= st.pop();
 ans = number1 + number2;
 st.push(ans);
 break;

 case "-":
 number2= st.pop();
 number1= st.pop();
 ans = number1 - number2;
 st.push(ans);
 break;

 case "*":
 number2= st.pop();
 number1= st.pop();
 ans = number1 * number2;
 st.push(ans);
 break;

 case "/":
 number2= st.pop();
 number1= st.pop();
 ans = number1 / number2;
 st.push(ans);
 break;

 default:
 st.push(Integer.parseInt(tokens[i]));
 break;
 }
 }
 return ans;
 }
}
```

```
 }
 }

 return st.pop();
}

}
```

## 151. Reverse Words in a String ↗



Given an input string `s`, reverse the order of the **words**.

A **word** is defined as a sequence of non-space characters. The **words** in `s` will be separated by at least one space.

Return *a string of the words in reverse order concatenated by a single space*.

**Note** that `s` may contain leading or trailing spaces or multiple spaces between two words. The returned string should only have a single space separating the words. Do not include any extra spaces.

### Example 1:

**Input:** `s = "the sky is blue"`

**Output:** `"blue is sky the"`

### Example 2:

**Input:** `s = " hello world "`

**Output:** `"world hello"`

**Explanation:** Your reversed string should not contain leading or trailing spaces.

### Example 3:

**Input:** `s = "a good example"`

**Output:** `"example good a"`

**Explanation:** You need to reduce multiple spaces between two words to a single space.

#### Example 4:

**Input:** s = " Bob Loves Alice "

**Output:** "Alice Loves Bob"

#### Example 5:

**Input:** s = "Alice does not even like bob"

**Output:** "bob like even not does Alice"

#### Constraints:

- $1 \leq s.length \leq 10^4$
- s contains English letters (upper-case and lower-case), digits, and spaces ' '.
- There is **at least one** word in s .

**Follow-up:** If the string data type is mutable in your language, can you solve it **in-place** with  $O(1)$  extra space?

---

```
class Solution {

 public String reverseWords(String s) {

 s = s.trim();
 String[] sa = s.split("\s+");
 // System.out.println(sa.toString());
 String result = "";

 for(int i = sa.length-1; i >= 0; i--)
 {
 result = result + sa[i];

 if(i!=0)
 result = result + " ";

 }
 //System.out.println(result);
 return result;
 }
}
```

## ***Second Approach***

```

class Solution {
 public String reverseWords(String s) {

 String res = "";
 StringBuilder sb = new StringBuilder (s);
 int i = 0;
 while (sb.length() > 0) // example
 {
 res = res.trim();
 if (sb.indexOf(" ") == 0)
 {
 sb = sb.delete(0, 1);
 continue;
 }
 if (sb.indexOf(" ") == -1) //agoodexample
 {
 res = sb.substring(0) + " " +res;
 sb.delete(0, sb.length());
 }
 else
 {
 //good ex
 res = sb.substring(0, sb.indexOf(" ")) + " " +res;
 res = res + " ";
 sb = sb.delete(0, sb.indexOf(" "));
 }
 }

 return res.trim();
 }
}

```

## 152. Maximum Product Subarray ↗

Given an integer array `nums`, find a contiguous non-empty subarray within the array that has the largest product, and return *the product*.

It is **guaranteed** that the answer will fit in a **32-bit** integer.

A **subarray** is a contiguous subsequence of the array.

### Example 1:

**Input:** nums = [2,3,-2,4]

**Output:** 6

**Explanation:** [2,3] has the largest product 6.

### Example 2:

**Input:** nums = [-2,0,-1]

**Output:** 0

**Explanation:** The result cannot be 2, because [-2,-1] is not a subarray.

### Constraints:

- $1 \leq \text{nums.length} \leq 2 * 10^4$
  - $-10 \leq \text{nums}[i] \leq 10$
  - The product of any prefix or suffix of `nums` is **guaranteed** to fit in a **32-bit** integer.
-

```
class Solution {
 public int maxProduct(int[] nums) {

 int l = nums.length;
 int[] org = nums.clone();
 int[] rev = new int[l];
 int r = 0, max = Integer.MIN_VALUE;

 for (int i = l-1; i >= 0; i--)//reversing
 {
 rev[r++] = nums[i];
 }

 int temp = nums[0];
 int tempRev = rev[0];
 max = Math.max(temp, tempRev);
 //seedha multiply karo 0 chorke
 //ulta multiply karo 0 chor k
 for (int i = 1; i < l; i++)
 {
 if (temp != 0)
 {
 org[i] = temp * nums[i];
 temp = org[i];
 }
 else
 {
 org[i] = nums[i];
 temp = nums[i];
 }
 if (tempRev != 0)
 {
 rev[i] = tempRev * rev[i];
 tempRev = rev[i];
 }
 else
 {
 rev[i] = rev[i];
 tempRev = rev[i];
 }
 max = Math.max(max, (Math.max(org[i],rev[i])));
 }
 return max;
 }
}
```

# 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.

## 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  $3 * 10^4$  calls will be made to `push`, `pop`, `top`, and `getMin`.

```
class MinStack {

 int min = Integer.MAX_VALUE;
 Stack<Node> stack = new Stack<>();

 public void push(int val) {
 if (val < min) min = val;

 Node newNode = new Node();
 newNode.val = val; //orginal val
 newNode.min = min; //abhi tk ki min value
 stack.push(newNode);
 }

 public void pop() {

 Node n = stack.pop();
 if (n.val == min && !stack.isEmpty()) min = stack.peek().min;
 if (n.val == min && stack.isEmpty()) min = Integer.MAX_VALUE;
 }

 public int top() {
 return stack.peek().val;
 }

 public int getMin() {
 return min;
 }

 private class Node {
 int val;
 int min;
 Node next;
 private Node() {

 }
 private Node(int val, int min, Node next) {
 this.val = val;
 this.min = min;
 }
 }
}

/**
 * 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();
*/
```

## 159. Longest Substring with At Most Two Distinct Characters ↗

Given a string  $s$ , return *the length of the longest substring that contains at most two distinct characters*.

### Example 1:

**Input:**  $s = \text{"eceba"}$

**Output:** 3

**Explanation:** The substring is "ece" which its length is 3.

### Example 2:

**Input:**  $s = \text{"ccaabbb"}$

**Output:** 5

**Explanation:** The substring is "aabbb" which its length is 5.

### Constraints:

- $1 \leq s.length \leq 10^5$
- $s$  consists of English letters.

```

//same as https://leetcode.com/problems/longest-substring-with-at-most-k-d
stinct-characters/
class Solution {
 public int lengthOfLongestSubstringTwoDistinct(String s) {

 if (s.length() <= 2) return s.length();

 int count = 0;
 int maxCount = Integer.MIN_VALUE;
 HashMap<Character, Integer> map = new HashMap<>();

 int start = 0;
 int end = 0;

 while (end < s.length()) {

 char currChar = s.charAt(end);
 map.put(currChar, map.getOrDefault(currChar, 0) +1);
 count++;

 if (map.size() < 2) { //unique characters < k
 end++;
 }
 else if (map.size() == 2) { //found k unique character ab calcu
lation karo

 maxCount = Math.max(maxCount, count);
 end++;
 }
 else if (map.size() > 2){ //more unique character remove

 while (map.size() > 2) { //move start jb tk window normal n
ahi hoti

 char startChar = s.charAt(start);
 int freq = map.get(startChar);
 if (freq == 1){
 map.remove(startChar);
 count--;
 }
 else {
 map.put(startChar, map.get(startChar) -1);
 count--;
 }
 start++;
 }
 }
 }
 }
}

```

```

 }
 end++; //put to upper ho he raha h, bs end badhana kafe h
 }
 maxCount = Math.max(maxCount, count); //jb nahi add kia tha tb
 ek tc fail ho raha tha...qki pure string ki length he unique count k barabe
 r t
}
return maxCount;
}

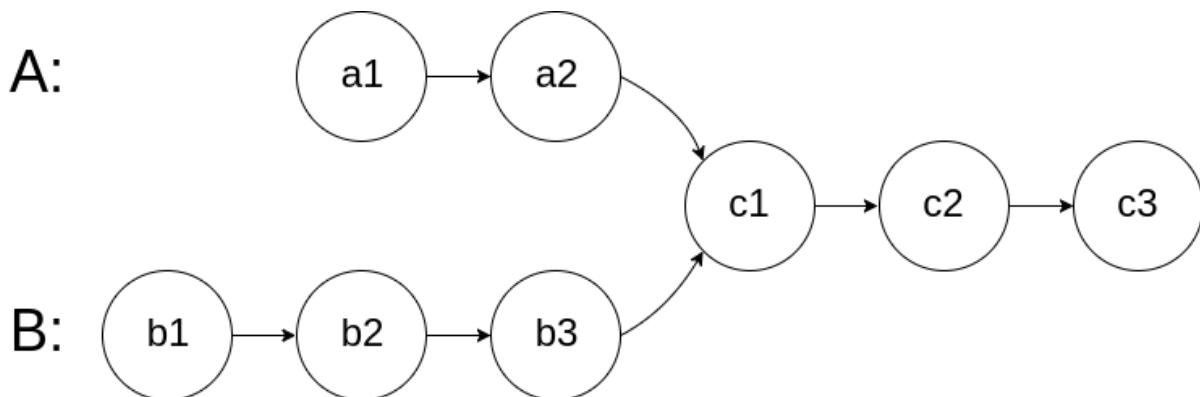
}

```

## 160. Intersection of Two Linked Lists ↗ ▾

Given the heads of two singly linked-lists `headA` and `headB`, return *the node at which the two lists intersect*. If the two linked lists have no intersection at all, return `null`.

For example, the following two linked lists begin to intersect at node `c1`:



The test cases are generated such that there are no cycles anywhere in the entire linked structure.

**Note** that the linked lists must **retain their original structure** after the function returns.

### Custom Judge:

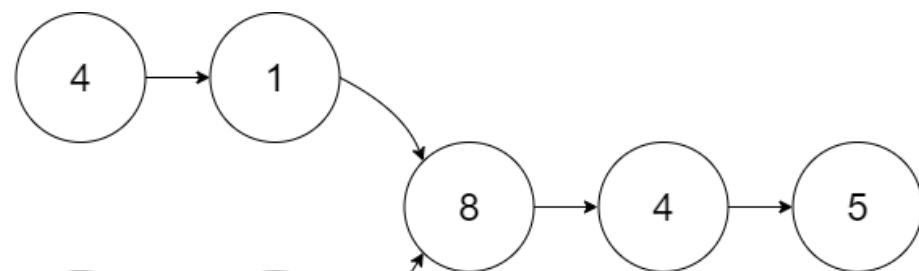
The inputs to the **judge** are given as follows (your program is **not** given these inputs):

- `intersectVal` - The value of the node where the intersection occurs. This is `0` if there is no intersected node.
- `listA` - The first linked list.
- `listB` - The second linked list.
- `skipA` - The number of nodes to skip ahead in `listA` (starting from the head) to get to the intersected node.
- `skipB` - The number of nodes to skip ahead in `listB` (starting from the head) to get to the intersected node.

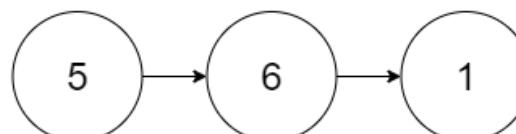
The judge will then create the linked structure based on these inputs and pass the two heads, `headA` and `headB` to your program. If you correctly return the intersected node, then your solution will be **accepted**.

### Example 1:

A:



B:



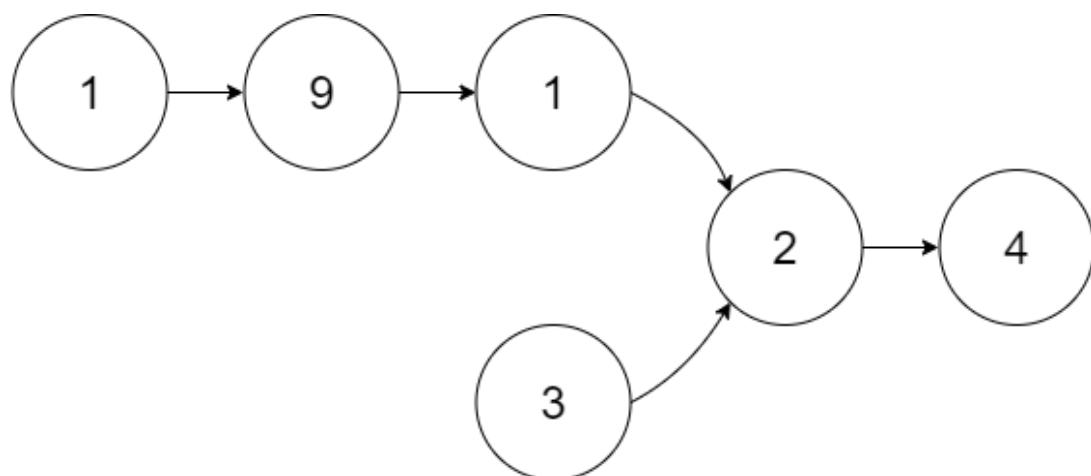
**Input:** `intersectVal = 8`, `listA = [4,1,8,4,5]`, `listB = [5,6,1,8,4,5]`, `skipA = 0`

**Output:** Intersected at '8'

**Explanation:** The intersected node's value is 8 (note that this must not be 0). From the head of A, it reads as [4,1,8,4,5]. From the head of B, it reads as [5,6,1,8,4,5].

### Example 2:

A:



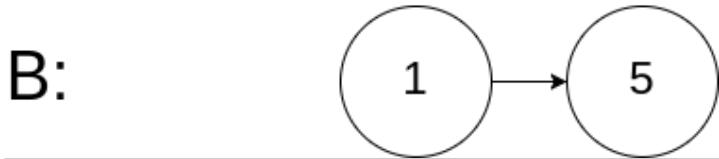
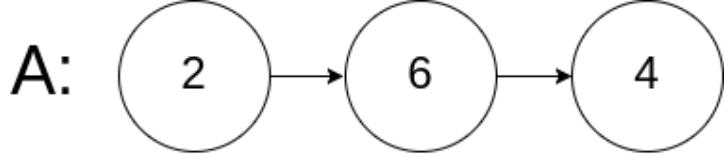
B:

**Input:** `intersectVal = 2`, `listA = [1,9,1,2,4]`, `listB = [3,2,4]`, `skipA = 3`, `skipB = 1`

**Output:** Intersected at '2'

**Explanation:** The intersected node's value is 2 (note that this must not be 0). From the head of A, it reads as [1,9,1,2,4]. From the head of B, it reads as [3,2,4].

### Example 3:



**Input:** intersectVal = 0, listA = [2,6,4], listB = [1,5], skipA = 3, skipB = 2

**Output:** No intersection

**Explanation:** From the head of A, it reads as [2,6,4]. From the head of B, it

**Explanation:** The two lists do not intersect, so return null.

### Constraints:

- The number of nodes of listA is in the m .
- The number of nodes of listB is in the n .
- $0 \leq m, n \leq 3 * 10^4$
- $1 \leq \text{Node.val} \leq 10^5$
- $0 \leq \text{skipA} \leq m$
- $0 \leq \text{skipB} \leq n$
- intersectVal is 0 if listA and listB do not intersect.
- intersectVal == listA[skipA] == listB[skipB] if listA and listB intersect.

---

**Follow up:** Could you write a solution that runs in O(n) time and use only O(1) memory?

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 * int val;
 * ListNode next;
 * ListNode(int x) {
 * val = x;
 * next = null;
 * }
 * }
 */
public class Solution {
 public ListNode getIntersectionNode(ListNode headA, ListNode headB)
 {
 if (headA == null || headB == null) return null;

 int lA = getLength(headA);
 int lB = getLength(headB);

 ListNode currNodeA = headA;
 ListNode currNodeB = headB;

 while (lA > lB)
 {
 currNodeA = currNodeA.next;
 lA--;
 }
 while (lA < lB)
 {
 currNodeB = currNodeB.next;
 lB--;
 }
 while (currNodeA != null && currNodeB != null)
 {
 if (currNodeA == currNodeB) return currNodeB;

 currNodeA = currNodeA.next;
 currNodeB = currNodeB.next;
 }

 return null;
 }

 public int getLength(ListNode node)
 {
 if (node == null) return 0;
 }
```

```
int length = 0;

while (node != null)
{
 ++length;
 node = node.next;
}
return length;
}
```

## 162. Find Peak Element ↗



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

Given an integer array `nums`, find a peak element, and return its index. If the array contains multiple peaks, return the index to **any of the peaks**.

You may imagine that `nums[-1] = nums[n] = -∞`.

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

### Example 1:

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

**Output:** 2

**Explanation:** 3 is a peak element and your function should return the index nu

### Example 2:

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

**Output:** 5

**Explanation:** Your function can return either index number 1 where the peak el

### Constraints:

- $1 \leq \text{nums.length} \leq 1000$
- $-2^{31} \leq \text{nums}[i] \leq 2^{31} - 1$
- $\text{nums}[i] \neq \text{nums}[i + 1]$  for all valid  $i$ .

```

class Solution {
 public int findPeakElement(int[] nums) {

 if (nums.length == 1) return 0;
 int po = 0;
 int low = 0, high = nums.length - 1;
 while (low <= high) {

 int mid = low+(high - low)/2;

 //mid 0 hai to aage nahi check karega..agr hai tb mid-1 out of
 bound nahi dega
 //mi last hai to aage nahi check karega..agr last nahi h tb out
 of bound nahi aaega
 if ((mid == 0 || nums[mid] > nums[mid-1]) &&
 (mid == nums.length-1 || nums[mid] > nums[mid+1]))
 {
 return mid;
 }
 if (nums[mid] > nums[mid+1]){ //possible peak eki mid+1 se ba
 da h
 po = mid;
 high = mid-1;
 }
 else{
 low = mid +1;
 }

 }
 return po;
 }
}

```

## 169. Majority Element ↗

Given an array `nums` of size `n`, return *the majority element*.

The majority element is the element that appears more than  $\lfloor n / 2 \rfloor$  times. You may assume that the majority element always exists in the array.

### Example 1:

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

**Output:** 3

### Example 2:

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

**Output:** 2

### Constraints:

- $n == \text{nums.length}$
- $1 \leq n \leq 5 * 10^4$
- $-2^{31} \leq \text{nums}[i] \leq 2^{31} - 1$

**Follow-up:** Could you solve the problem in linear time and in  $O(1)$  space?

---

```
//https://www.youtube.com/watch?v=X0G5jEcvroo
class Solution {
 public int majorityElement(int[] nums) {

 int majority = nums[0];
 int count = 1;

 for (int i = 1; i < nums.length; i++) {
 if (nums[i] == majority) {
 count++;
 } else {
 count--;
 if (count == 0) {
 majority = nums[i];
 count++;
 }
 }
 }
 return majority;
 }
}
```

Taking care of **The majority element is the element that appears more than [ n/2 ] times** as well..

```
class Solution {
 public int majorityElement(int[] nums) {

 Arrays.sort(nums);
 // int e = nums.length/2;
 return nums[nums.length/2];
 }
}
```

## ***Second Approach***

- used **Collections.max(map.values())** to get maximum value among all the value in map

```
class Solution {
 public int majorityElement(int[] nums) {

 HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();

 for (int i = 0; i < nums.length; i++)
 {
 if (map.containsKey(nums[i]))
 {
 map.put(nums[i], map.get(nums[i])+1);
 }
 else {
 map.put(nums[i], 1);
 }
 }

 int maxValueInMap=(Collections.max(map.values()));

 Set<Integer> keys= map.keySet();
 for (int k : keys)
 {
 if (map.get(k) == maxValueInMap)
 return k;
 }
 return 0;
 }
}
```

Practice

```

class Solution {
 public int majorityElement(int[] nums) {

 HashMap<Integer, Integer> map = new HashMap<>();
 int maxTimes = Integer.MIN_VALUE, max = nums[0];

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

 if (map.containsKey(nums[i])){
 map.put(nums[i], map.get(nums[i])+1);

 }
 else {
 map.put(nums[i], 1);
 }
 if (maxTimes < map.get(nums[i])){
 maxTimes = map.get(nums[i]);
 max = nums[i];
 }
 }

 return max;
 }
}

```

## 172. Factorial Trailing Zeroes ↗

Given an integer  $n$ , return *the number of trailing zeroes in  $n!$* .

Note that  $n! = n * (n - 1) * (n - 2) * \dots * 3 * 2 * 1$ .

### Example 1:

**Input:**  $n = 3$

**Output:** 0

**Explanation:**  $3! = 6$ , no trailing zero.

### Example 2:

**Input:** n = 5

**Output:** 1

**Explanation:**  $5! = 120$ , one trailing zero.

### Example 3:

**Input:** n = 0

**Output:** 0

### Constraints:

- $0 \leq n \leq 10^4$

**Follow up:** Could you write a solution that works in logarithmic time complexity?

```
class Solution
{
 //Basically we are counting number of 5 in the number
 public int trailingZeroes(int n)
 {
 int numZeroes = 0;
 for (int i = 5; i <= n; i *= 5)
 {
 numZeroes += Math.floor(n / i);
 }
 return numZeroes;
 }
}
```

## 179. Largest Number ↗



Given a list of non-negative integers `nums`, arrange them such that they form the largest number.

**Note:** The result may be very large, so you need to return a string instead of an integer.

### **Example 1:**

**Input:** nums = [10, 2]

**Output:** "210"

### **Example 2:**

**Input:** nums = [3, 30, 34, 5, 9]

**Output:** "9534330"

### **Example 3:**

**Input:** nums = [1]

**Output:** "1"

### **Example 4:**

**Input:** nums = [10]

**Output:** "10"

### **Constraints:**

- $1 \leq \text{nums.length} \leq 100$
- $0 \leq \text{nums}[i] \leq 10^9$

```
class Solution {
 public String largestNumber(int[] nums) {

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

 for (int i = 0; i < nums.length; i++)
 {
 al.add(String.valueOf(nums[i]));
 }
 // System.out.print(al);
 Collections.sort(al, (a,b) -> {
 String s1 = a + b;
 String s2 = b + a;
 return s1.compareTo(s2);
 });
 // System.out.print(al);
 StringBuffer sb = new StringBuffer();
 for (int i = al.size() -1; i >= 0; i--)
 {
 sb.append(al.get(i));
 }
 String ans = sb.toString();
 return (ans.charAt(0) == '0')? "0" : ans; //for 000 case
 }

}
```

```

class Solution {
 public String largestNumber(int[] nums) {

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

 for (int i = 0; i < nums.length; i++)
 {
 al.add(String.valueOf(nums[i]));
 }
 System.out.print(al);
 Collections.sort(al, new Comparator()
 {
 public int compare(Object s1, Object s2)
 {
 String ss1 = (String)s1 + (String)s2;
 String ss2 = (String)s2 + (String)s1;
 return ss1.compareTo(ss2);
 }
 });
 System.out.print(al);
 StringBuffer sb = new StringBuffer();
 for (int i = al.size() - 1; i >= 0; i--)
 {
 sb.append(al.get(i));
 }
 String ans = sb.toString();
 return (ans.charAt(0) == '0')? "0" : ans; //for 000 cases

 //return "ans";
 }

}

```

## 189. Rotate Array ↗

Given an array, rotate the array to the right by  $k$  steps, where  $k$  is non-negative.

**Example 1:**

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

**Output:** [5,6,7,1,2,3,4]

**Explanation:**

rotate 1 steps to the right: [7,1,2,3,4,5,6]

rotate 2 steps to the right: [6,7,1,2,3,4,5]

rotate 3 steps to the right: [5,6,7,1,2,3,4]

## Example 2:

**Input:** nums = [-1,-100,3,99], k = 2

**Output:** [3,99,-1,-100]

**Explanation:**

rotate 1 steps to the right: [99,-1,-100,3]

rotate 2 steps to the right: [3,99,-1,-100]

## Constraints:

- $1 \leq \text{nums.length} \leq 10^5$
- $-2^{31} \leq \text{nums}[i] \leq 2^{31} - 1$
- $0 \leq k \leq 10^5$

## Follow up:

- Try to come up with as many solutions as you can. There are at least **three** different ways to solve this problem.
- Could you do it in-place with  $O(1)$  extra space?

```

//for negative k: k = k+nums.length
class Solution {
 public void rotate(int[] nums, int k)
 {
 if (k > nums.length)
 {
 k = k%nums.length;
 }
 reverse(nums, 0, nums.length - k -1); //reverse 1st half
 reverse(nums, nums.length -k, nums.length-1); //reverse 2nd half
 reverse(nums, 0, nums.length-1); //reverse full
 }

 public void reverse(int[] nums, int start, int end)
 {
 while (start < end)
 {
 int temp = nums[start];
 nums[start] = nums[end];
 nums[end] = temp;
 start++;
 end--;
 }
 }
}

```

## 191. Number of 1 Bits ↗

Write a function that takes an unsigned integer and returns the number of '1' bits it has (also known as the Hamming weight ([http://en.wikipedia.org/wiki/Hamming\\_weight](http://en.wikipedia.org/wiki/Hamming_weight))).

### Note:

- Note that in some languages, such as Java, there is no unsigned integer type. In this case, the input will be given as a signed integer type. It should not affect your implementation, as the integer's internal binary representation is the same, whether it is signed or unsigned.
- In Java, the compiler represents the signed integers using 2's complement notation ([https://en.wikipedia.org/wiki/Two%27s\\_complement](https://en.wikipedia.org/wiki/Two%27s_complement)). Therefore, in **Example 3**, the input represents the signed integer. -3 .

### Example 1:

**Input:** n = 00000000000000000000000000001011

**Output:** 3

**Explanation:** The input binary string 00000000000000000000000000001011 has a total of 3 set bits.

### Example 2:

**Input:** n = 00000000000000000000000000001000

**Output:** 1

**Explanation:** The input binary string 00000000000000000000000000001000 has a total of 1 set bit.

### Example 3:

**Input:** n = 11111111111111111111111111111101

**Output:** 31

**Explanation:** The input binary string 111111111111111111111111111101 has a total of 31 set bits.

### Constraints:

- The input must be a **binary string** of length 32.

**Follow up:** If this function is called many times, how would you optimize it?

```
public class Solution { // you need to treat n as an unsigned value public int hammingWeight(int n) { int count = 0;
```

```
while (n!=0)///if we will write n > 0 it will fail for some cases(may be Integer Overflow) {
```

```
count++; n = n & (n-1);///this makes the last 1 in number as 0 } return count;
```

```
//Second Approach
//return Integer.bitCount(n);
}
```

```
}
```

# 198. House Robber ↗

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security systems connected and **it will automatically contact the police if two adjacent houses were broken into on the same night.**

Given an integer array `nums` representing the amount of money of each house, return *the maximum amount of money you can rob tonight **without alerting the police**.*

## Example 1:

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

**Output:** 4

**Explanation:** Rob house 1 (`money = 1`) and then rob house 3 (`money = 3`).

Total amount you can rob =  $1 + 3 = 4$ .

## Example 2:

**Input:** `nums = [2,7,9,3,1]`

**Output:** 12

**Explanation:** Rob house 1 (`money = 2`), rob house 3 (`money = 9`) and rob house 5

Total amount you can rob =  $2 + 9 + 1 = 12$ .

## Constraints:

- $1 \leq \text{nums.length} \leq 100$
- $0 \leq \text{nums}[i] \leq 400$

```
nums: 2 | 1 | 1 | 2
 | | |
eHouse: 0 | 2 | 2 | 3 max od eHouse and oHouse
oHouse: 2 | 1 | 3 | 4 purana eHouse ka value plus current(ye pakka ho ga
ya ek gap to hai he, adjacent ni h)
```

```
class Solution {
 public int rob(int[] nums)
 {
 int eHouse = 0;//include ni karna
 int oHouse = nums[0];//include kis

 int newEHouse = 0, newOHouse = 0;
 for (int i = 1; i < nums.length; i++)
 {
 newEHouse = Math.max(eHouse,oHouse);
 newOHouse = eHouse + nums[i];

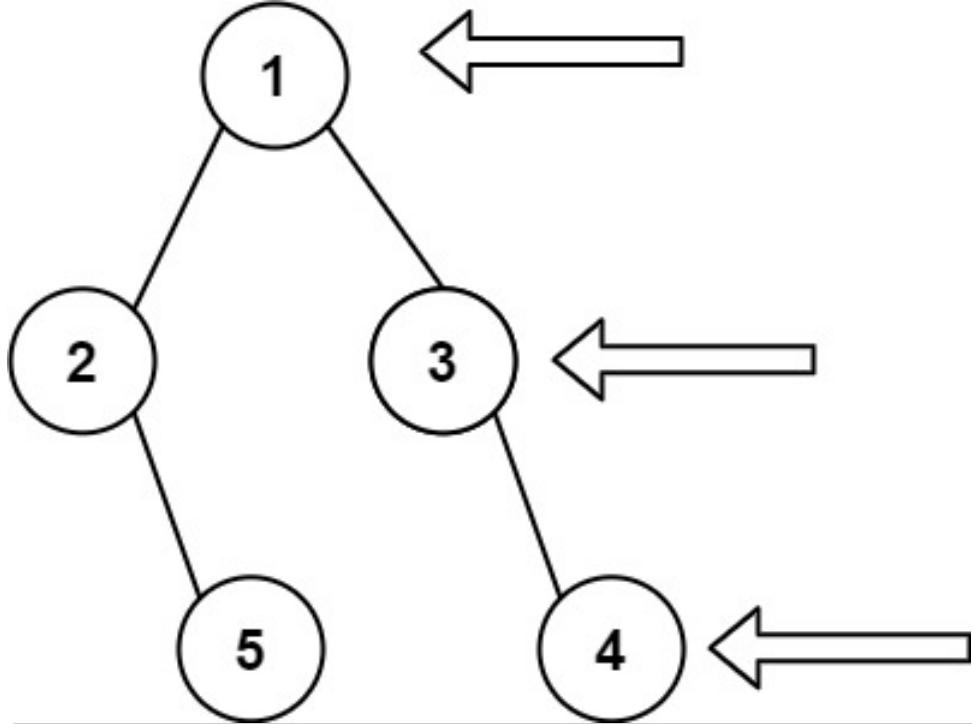
 oHouse = newOHouse;
 eHouse = newEHouse;
 }

 return Math.max(eHouse, oHouse);
 }
}
```

## 199. Binary Tree Right Side View ↗

Given the `root` of a binary tree, imagine yourself standing on the **right side** of it, return *the values of the nodes you can see ordered from top to bottom.*

**Example 1:**



**Input:** root = [1,2,3,null,5,null,4]

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

### Example 2:

**Input:** root = [1,null,3]

**Output:** [1,3]

### Example 3:

**Input:** root = []

**Output:** []

### Constraints:

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

```

class Solution {
 public List<Integer> rightSideView(TreeNode root) {
 List<Integer> visibleValue = new ArrayList<Integer>();
 if (root == null) return visibleValue;
 Queue<TreeNode> queue = new LinkedList<TreeNode>();

 queue.add(root);

 while (!queue.isEmpty())
 {
 int size = queue.size();
 TreeNode n;
 for (int i = 0; i < size; i++)
 {
 n = queue.remove();
 if (i == size-1) visibleValue.add(n.val); //this will be last node on right side, so store it

 if (n.left != null) queue.add(n.left);
 if (n.right != null) queue.add(n.right);
 }
 }
 return visibleValue;
 }
}

```

## 200. Number of Islands ↗

Given an  $m \times n$  2D binary grid  $\text{grid}$  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:

- $m == \text{grid.length}$
- $n == \text{grid[i].length}$
- $1 \leq m, n \leq 300$
- $\text{grid[i][j]}$  is '0' or '1' .

```

class Solution {
 public int numIslands(char[][] grid)
 {
 int c = 0;
 boolean visited[][] = new boolean[grid.length][grid[0].length];
 for (int i = 0; i < grid.length; i++)
 {
 for (int j = 0; j < grid[0].length; j++)
 {
 //land & unvisited
 if (grid[i][j] == '1' && visited[i][j] == false)
 {
 countIsland(grid, i, j, visited);
 c++;
 }
 }
 }
 return c;
 }
 public void countIsland(char[][] grid, int a, int b, boolean[][] visite
d)
{
 //board k baher, visited, water to baher
 if (a < 0 || b < 0 || a >= grid.length || b >= grid[0].length ||
 visited[a][b] == true || grid[a][b] == '0')
 {
 return;
 }

 visited[a][b] = true;
 countIsland(grid, a+1, b, visited); //down
 countIsland(grid, a-1, b, visited); //up
 countIsland(grid, a, b+1, visited); //right
 countIsland(grid, a, b-1, visited); //left
}
}

```

## 202. Happy Number ↗

Write an algorithm to determine if a number  $n$  is happy.

A **happy number** is a number defined by the following process:

- Starting with any positive integer, replace the number by the sum of the squares of its digits.
- Repeat the process until the number equals 1 (where it will stay), or it **loops endlessly in a cycle** which does not include 1.
- Those numbers for which this process **ends in 1** are happy.

Return `true` if  $n$  is a happy number, and `false` if not.

### Example 1:

**Input:**  $n = 19$

**Output:** true

**Explanation:**

$$1^2 + 9^2 = 82$$

$$8^2 + 2^2 = 68$$

$$6^2 + 8^2 = 100$$

$$1^2 + 0^2 + 0^2 = 1$$

### Example 2:

**Input:**  $n = 2$

**Output:** false

### Constraints:

- $1 \leq n \leq 2^{31} - 1$

```

class Solution {
 public boolean isHappy(int n) {

 while(true)
 {
 int sq = 0;
 while(n>0)
 {
 int r = n%10;
 sq = sq + (r*r);
 n = n/10;

 }
 if(sq == 1)
 return true;
 else if (sq == 4)
 return false;
 else
 n = sq;
 }
 }
}

```

## 204. Count Primes ↗

Given an integer  $n$ , return *the number of prime numbers that are strictly less than  $n$* .



### Example 1:

**Input:**  $n = 10$

**Output:** 4

**Explanation:** There are 4 prime numbers less than 10, they are 2, 3, 5, 7.

### Example 2:

**Input:**  $n = 0$

**Output:** 0

### Example 3:

**Input:** n = 1

**Output:** 0

### Constraints:

- $0 \leq n \leq 5 * 10^6$

```
class Solution {
 public int countPrimes(int n) {

 if(n < 3) return 0;

 int count = 0;
 boolean isPrime[] = new boolean[n];
 Arrays.fill(isPrime, true);

 isPrime[0] = false;
 isPrime[1] = false;

 for (int i = 2; i*i < n; i++)
 {
 if (isPrime[i])
 {
 for (int j = 2*i; j < n; j= j+i)
 {
 if(isPrime[j]) isPrime[j] = false;
 }
 }
 }

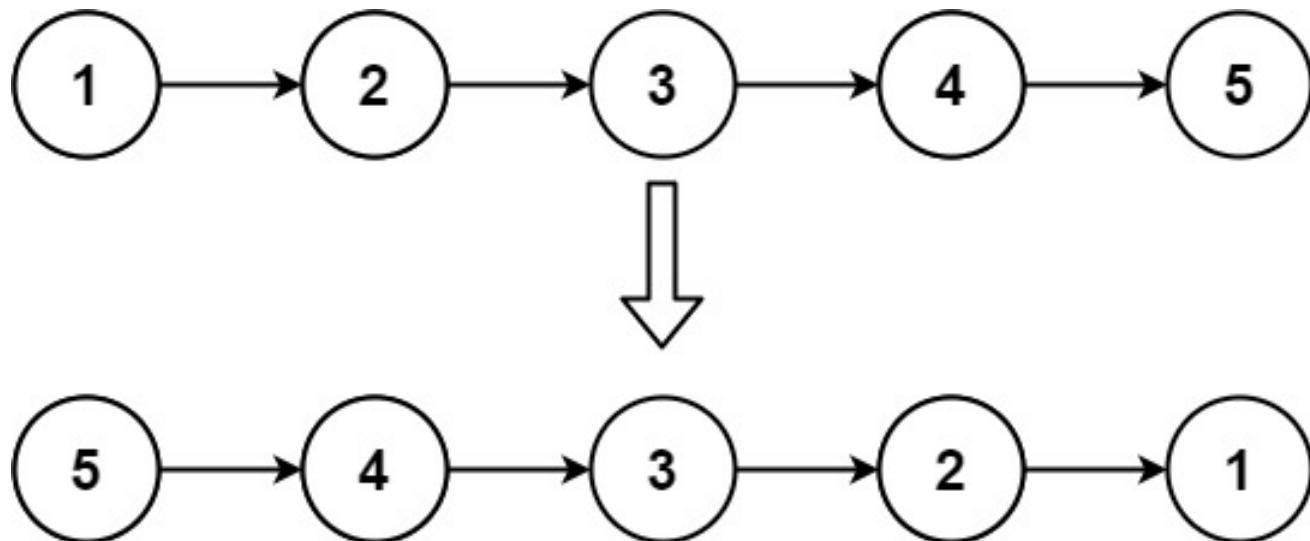
 for (int i = 0; i < n; i++)
 {
 if (isPrime[i]) count++;
 }

 return count;
 }
}
```

# 206. Reverse Linked List ↗

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

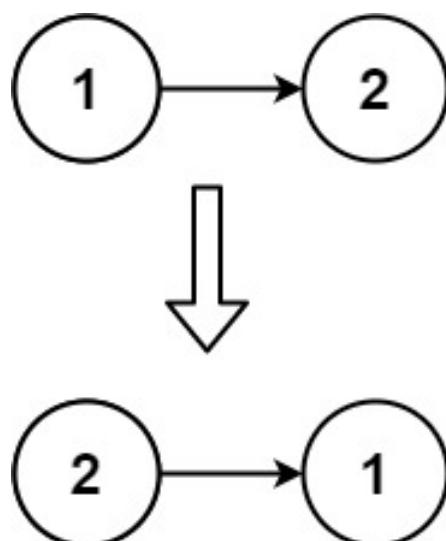
## Example 1:



**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 in 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?

```
class Solution {
 public ListNode reverseList(ListNode head)
 {
 ListNode prevNode = null;
 ListNode currNode = head;
 ListNode nextNode = head;

 while (currNode != null)
 {
 nextNode = currNode.next;
 currNode.next = prevNode;
 prevNode = currNode;
 currNode = nextNode;
 }
 head = prevNode;

 return head;
 }
}
```

## 209. Minimum Size Subarray Sum

Given an array of positive integers `nums` and a positive integer `target`, return the minimal length of a **contiguous subarray**  $[nums_1, nums_{1+1}, \dots, nums_{r-1}, nums_r]$  of which the sum is greater than or equal to `target`. If there is no such subarray, return `0` instead.

### Example 1:

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

**Output:** 2

**Explanation:** The subarray [4,3] has the minimal length under the problem constraints.

### Example 2:

**Input:** target = 4, nums = [1,4,4]

**Output:** 1

### Example 3:

**Input:** target = 11, nums = [1,1,1,1,1,1,1,1]

**Output:** 0

### Constraints:

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

**Follow up:** If you have figured out the  $O(n)$  solution, try coding another solution of which the time complexity is  $O(n \log(n))$ .

## Sliding Window

```

class Solution {
 public int minSubArrayLen(int s, int[] nums)
 {
 int sum = 0, min = 0, nElements = Integer.MAX_VALUE;
 int j = 0, i = 0;
 while(i < nums.length)
 {
 while (sum < s && j < nums.length)
 {
 sum = sum + nums[j];
 ++j;
 ++min;
 }
 if (sum == s)
 {
 nElements = Math.min(min, nElements);
 sum = sum - nums[i];
 --min;
 }

 if (sum > s)
 {
 sum = sum - nums[i];
 nElements = Math.min(min, nElements);
 --min;
 }
 ++i;
 }
 return (nElements == Integer.MAX_VALUE)?0:nElements;
 }
}

```

## 210. Course Schedule II ↗

There are a total of `numCourses` courses you have to take, labeled from `0` to `numCourses - 1`. You are given an array `prerequisites` where `prerequisites[i] = [ai, bi]` indicates that you **must** take course `bi` first if you want to take course `ai`.

- For example, the pair `[0, 1]`, indicates that to take course `0` you have to first take course `1`.

Return the ordering of courses you should take to finish all courses. If there are many valid answers, return **any** of them. If it is impossible to finish all courses, return **an empty array**.

### Example 1:

**Input:** numCourses = 2, prerequisites = [[1,0]]

**Output:** [0,1]

**Explanation:** There are a total of 2 courses to take. To take course 1 you shc

### Example 2:

**Input:** numCourses = 4, prerequisites = [[1,0],[2,0],[3,1],[3,2]]

**Output:** [0,2,1,3]

**Explanation:** There are a total of 4 courses to take. To take course 3 you shc

So one correct course order is [0,1,2,3]. Another correct ordering is [0,2,1,

### Example 3:

**Input:** numCourses = 1, prerequisites = []

**Output:** [0]

### Constraints:

- $1 \leq \text{numCourses} \leq 2000$
- $0 \leq \text{prerequisites.length} \leq \text{numCourses} * (\text{numCourses} - 1)$
- $\text{prerequisites}[i].length == 2$
- $0 \leq a_i, b_i < \text{numCourses}$
- $a_i \neq b_i$
- All the pairs  $[a_i, b_i]$  are **distinct**.

```

//Topological Sort : BFS
//0 indegree ko queue me dalo
//queue ko process karo
//element nikalo..mataln ab vo element hata dia...uske karan jo indegree thi
//uske neighbour pr vo kam kar do
//agr koi neighbour ab 0 indegree ka ho gaya...to use queue me daal do
//last me index se check karo kya humne sb course complete kia ?
class Solution {
 public int[] findOrder(int numCourses, int[][] prerequisites) {

 int[] ans = new int[numCourses];
 List<List<Integer>> graph = new ArrayList<>();
 int[] indegree = new int[numCourses];
 Queue<Integer> queue = new LinkedList<>();

 for (int i = 0; i < numCourses; i++){
 graph.add(new ArrayList<>(numCourses));
 }

 for (int[] row : prerequisites) {

 int course = row[0]; //pre se course me jana h*****
 int pre = row[1];
 graph.get(pre).add(course);
 indegree[course]++;
 }

 for (int i = 0; i < indegree.length; i++) {
 if (indegree[i] == 0) queue.add(i); //from this node we can start as it independent
 }

 int index = 0;
 while (!queue.isEmpty()) {

 int source = queue.remove(); //isko baher kar rahai islia iske karan jisme indegree thi use -1 kar rahi

 ans[index] = source;
 index++; //this is also counting number of course completed

 for (int neighbour : graph.get(source)) {
 indegree[neighbour]--;
 if (indegree[neighbour] == 0){
 queue.add(neighbour);
 }
 }
 }
 }
}

```

```

 }
 }

 //index was also acting as a counter
 if (index == numCourses) return ans;

 //count match nahi kar rahi matlab hum sb course nahi kar paे****
 if (index != numCourses) return new int[0];//queue me kabhi kch gay
a he nahi

 return new int[1]; //matlab pre me kch nahi tha

}

```

## 212. Word Search II ↗

Given an  $m \times n$  board of characters and a list of strings `words`, return *all words on the board*.

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

**Example 1:**

|          |          |          |          |
|----------|----------|----------|----------|
| <b>o</b> | <b>a</b> | <b>a</b> | <b>n</b> |
| <b>e</b> | <b>t</b> | <b>a</b> | <b>e</b> |
| <b>i</b> | <b>h</b> | <b>k</b> | <b>r</b> |
| <b>i</b> | <b>f</b> | <b>l</b> | <b>v</b> |

**Input:** board = [["o","a","a","n"],["e","t","a","e"],["i","h","k","r"],["i","f","l","v"]]

**Output:** ["eat", "oath"]

### Example 2:

|          |          |
|----------|----------|
| <b>a</b> | <b>b</b> |
| <b>c</b> | <b>d</b> |

**Input:** board = [["a","b"],["c","d"]], words = ["abcb"]

**Output:** []

### Constraints:

- $m == \text{board.length}$
- $n == \text{board}[i].length$
- $1 \leq m, n \leq 12$
- $\text{board}[i][j]$  is a lowercase English letter.
- $1 \leq \text{words.length} \leq 3 * 10^4$
- $1 \leq \text{words}[i].length \leq 10$
- $\text{words}[i]$  consists of lowercase English letters.

- All the strings of words are unique.
- 

1457 ms, Should be done using trie

```

class Solution {
 public List<String> findWords(char[][] board, String[] words) {

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

 boolean [][] visited;
 boolean found;

 for (String word : words)
 {
 char firstChar = word.charAt(0);
 visited = new boolean [board.length][board[0].length];
 found = false;
 for (int i = 0; i < board.length && !found; i++)
 {
 for (int j = 0; j < board[0].length && !found; j++)
 {
 if (board[i][j] == firstChar)
 {
 if (helper(board, word, visited, i, j, 0)) {
 res.add(word);
 found = true;
 }
 }
 }
 }
 }

 return res;
 }

 private boolean helper(char[][] board, String word, boolean[][] visited, int i, int j, int count)
 {
 if (i < 0 || i >= board.length || j < 0 || j >= board[0].length || visited[i][j] ||
 word.charAt(count) != board[i][j]) return false;

 if (word.length()-1 == count) return true; //means word mil gaya

 visited[i][j] = true;
 if (helper(board, word, visited, i+1, j, count+1) ||
 helper(board, word, visited, i, j+1, count+1) ||
 helper(board, word, visited, i, j-1, count+1) ||
 helper(board, word, visited, i-1, j, count+1))
 }
}

```

```

 return true;

 visited[i][j] = false;
 count--;

 return false;
}

}

```

## 213. House Robber II ↗

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed. All houses at this place are **arranged in a circle**. That means the first house is the neighbor of the last one. Meanwhile, adjacent houses have a security system connected, and **it will automatically contact the police if two adjacent houses were broken into on the same night**.

Given an integer array `nums` representing the amount of money of each house, return *the maximum amount of money you can rob tonight **without alerting the police***.

### Example 1:

**Input:** `nums = [2,3,2]`

**Output:** 3

**Explanation:** You cannot rob house 1 (`money = 2`) and then rob house 3 (`money =`

### Example 2:

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

**Output:** 4

**Explanation:** Rob house 1 (`money = 1`) and then rob house 3 (`money = 3`).

Total amount you can rob =  $1 + 3 = 4$ .

### Example 3:

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

**Output:** 3

### **Constraints:**

- $1 \leq \text{nums.length} \leq 100$
  - $0 \leq \text{nums}[i] \leq 1000$
-

```

/*
 1, 3, 1, 3, 100
eHouse 0 1 3 3 9
oHouse 1 0 + 3 = 3 1+1=2 3+3=9 100+3 = 103

*/
/* pahle mai alternate leke kar rahai thi...but dhyaan rakho ye alternate ka ques nahi h...ye INCLUDE EXCLUDE hai...beck k 2 ya 3 house bhi hum chor skte h */
*/

class Solution {
 public int rob(int[] nums) {

 if (nums.length == 0) return 0;
 else if (nums.length == 1) return nums[0];
 else if (nums.length == 2) return Math.max(nums[0], nums[1]);

 int[] firstHouseExcluded = Arrays.copyOfRange(nums, 1, nums.length); //1st excluded
 int[] firstHouseIncluded = Arrays.copyOf(nums, nums.length-1); //last excluded

 int fhEMaxMoney = robHouse(firstHouseExcluded);
 int fhIMaxMoney = robHouse(firstHouseIncluded);

 return Math.max(fhEMaxMoney, fhIMaxMoney);
 }

 public int robHouse(int[] nums)
 {
 int eHouse = 0;//include ni karna
 int oHouse = nums[0];//include kia

 int newEHouse = 0, newOHouse = 0;
 for (int i = 1; i < nums.length; i++)
 {
 newEHouse = Math.max(eHouse,oHouse); //isko include nahi kar rahai islia abhi tk ka max leker chal rahai
 newOHouse = eHouse + nums[i]; //include kar rahai hai to e se leker add karege(ye o hai islia e se le rahai)

 oHouse = newOHouse;
 eHouse = newEHouse;
 }
 }
}

```

```
 return Math.max(eHouse, oHouse);
}
}
```

## 215. Kth Largest Element in an Array ↗

Given an integer array `nums` and an integer `k`, return *the k<sup>th</sup> largest element in the array*.

Note that it is the `kth` largest element in the sorted order, not the `kth` distinct element.

### Example 1:

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

### Example 2:

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

### Constraints:

- $1 \leq k \leq \text{nums.length} \leq 10^4$
- $-10^4 \leq \text{nums}[i] \leq 10^4$

```
class Solution {
 public int findKthLargest(int[] nums, int k) {

 //min heap
 PriorityQueue<Integer> pQueue = new PriorityQueue<Integer>();

 for (int n : nums)
 {
 pQueue.add(n);

 if (pQueue.size() > k)
 {
 pQueue.poll();
 }
 }
 return pQueue.peek();
 }
}
```

## 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) {

 Set<Integer> s = new HashSet<Integer>();
 for(int i = 0; i < nums.length; i++)
 {
 if(s.contains(nums[i]))
 {
 return true;
 }
 else
 s.add(nums[i]);

 }

 return false;
 }
}
```

## 219. Contains Duplicate II ↗

Given an integer array `nums` and an integer `k`, return `true` if there are two **distinct indices** `i` and `j` in the array such that `nums[i] == nums[j]` and `abs(i - j) <= k`.

### Example 1:

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

### Example 2:

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

**Output:** true

### Example 3:

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

**Output:** false

### Constraints:

- $1 \leq \text{nums.length} \leq 10^5$
- $-10^9 \leq \text{nums}[i] \leq 10^9$
- $0 \leq k \leq 10^5$

this should come first in mind

```
//index maintain kar rahai hai element ka
class Solution {
 public boolean containsNearbyDuplicate(int[] nums, int k) {

 HashMap<Integer, Integer> map = new HashMap<>(); //element : last index it occurred

 for (int i = 0; i < nums.length; i++) {
 int currEle = nums[i];
 if (map.containsKey(currEle) && Math.abs(map.get(currEle) - i) <= k) {
 return true;
 }
 map.put(currEle,i); //inserting or updating curr index
 }
 return false;
 }
}
```

Slower approach but similar to contains Duplicate 3

```

class Solution {
 public boolean containsNearbyDuplicate(int[] nums, int k) {

 TreeSet<Integer> window = new TreeSet<>();

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

 int currEle = nums[i];

 if (window.contains(currEle)) return true;

 window.add(currEle);
 if (window.size() > k) {
 window.remove(nums[i-k]);
 }
 }
 return false;
 }
}

```

## 220. Contains Duplicate III ↗

Given an integer array `nums` and two integers `k` and `t`, return `true` if there are **two distinct indices** `i` and `j` in the array such that `abs(nums[i] - nums[j]) <= t` and `abs(i - j) <= k`.

### Example 1:

**Input:** `nums = [1,2,3,1]`, `k = 3`, `t = 0`  
**Output:** `true`

### Example 2:

**Input:** `nums = [1,0,1,1]`, `k = 1`, `t = 2`  
**Output:** `true`

### Example 3:

**Input:** nums = [1,5,9,1,5,9], k = 2, t = 3

**Output:** false

**Constraints:**

- $0 \leq \text{nums.length} \leq 2 * 10^4$
  - $-2^{31} \leq \text{nums}[i] \leq 2^{31} - 1$
  - $0 \leq k \leq 10^4$
  - $0 \leq t \leq 2^{31} - 1$
-

```

//Using TreeSet as it maintains order
//Sliding in nums and maintaining set size as k since (i-j) <=k
//TC O(nlogk) --> O(n) * (O(logk) + O(logk))
class Solution {
 public boolean containsNearbyAlmostDuplicate(int[] nums, int k, int t)
{

 TreeSet<Long> window = new TreeSet<>();

 for (int i = 0; i < nums.length; i++) { //--> O(n)

 Long currentElement = new Long(nums[i]);
 //learned below 2 new things...
 Long floorElement = window.floor(new Long(nums[i])); //nums[i]
ka floor jo set me hai O(logk) coz k element h at a time
 Long ceilElement = window.ceiling(new Long(nums[i])); //nums[i]
ka ceil jo set me hai O(logk)
 //staring me floor and ceil dono null hoga he
 if (floorElement != null && Math.abs(floorElement - nums[i]) <
t) {
 return true;
 }
 else if (ceilElement != null && Math.abs(ceilElement - nums[i])
<= t) {
 return true;
 }
 window.add(new Long(nums[i])); //element add karte chalo
 if (window.size() > k) { //maintain window size..to apne aap i-j
maintain rahaega
 window.remove(new Long(nums[i-k])); //left se element remove
kar rai
 }
 }
 return false;
}
}

```

## 221. Maximal Square ↗

Given an  $m \times n$  binary matrix filled with 0's and 1's, find the largest square containing only 1's and return its area.

### Example 1:

|   |   |   |   |   |
|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 |

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

**Output:** 4

### Example 2:

|   |   |
|---|---|
| 0 | 1 |
| 1 | 0 |

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

**Output:** 1

### Example 3:

**Input:** matrix = [[ "0"]]

**Output:** 0

**Constraints:**

- $m == \text{matrix.length}$
  - $n == \text{matrix}[i].length$
  - $1 \leq m, n \leq 300$
  - $\text{matrix}[i][j]$  is '0' or '1'.
-

```

class Solution {
 public int maximalSquare(char[][] matrix)
 {
 int m = matrix.length;
 int n = matrix[0].length;
 int [][] dp = new int[m][n];
 int maxSq = 0;

 //last row : last row me max 1 size ka he square ho skta h
 for (int i = 0; i < n; i++)
 {
 if (matrix[m-1][i] == '1') dp[m-1][i] = 1; //last row fixed
 maxSq = Math.max(maxSq, dp[m-1][i]);
 }

 //last col : last col me max 1 size ka he square ho skta h
 for (int i = 0; i < m; i++)
 {
 if (matrix[i][n-1] == '1') dp[i][n-1] = 1; //last col fixed
 maxSq = Math.max(maxSq, dp[i][n-1]);
 }

 //rest of matrix
 for (int i = m-2; i >= 0; i--)
 {
 for (int j = n-2; j >=0; j--)
 {
 if (matrix[i][j] == '1')
 {
 int min = dp[i][j+1];//aise he hum koi bhi minimum rakh s
 kte h starting me

 min = Math.min(min, Math.min(dp[i+1][j+1], dp[i+1]
 [j]));//total 3 direction check karna h.. right, down and down-right dia
 dp[i][j] = min +1;
 maxSq = Math.max(maxSq, dp[i][j]);
 }
 }
 }

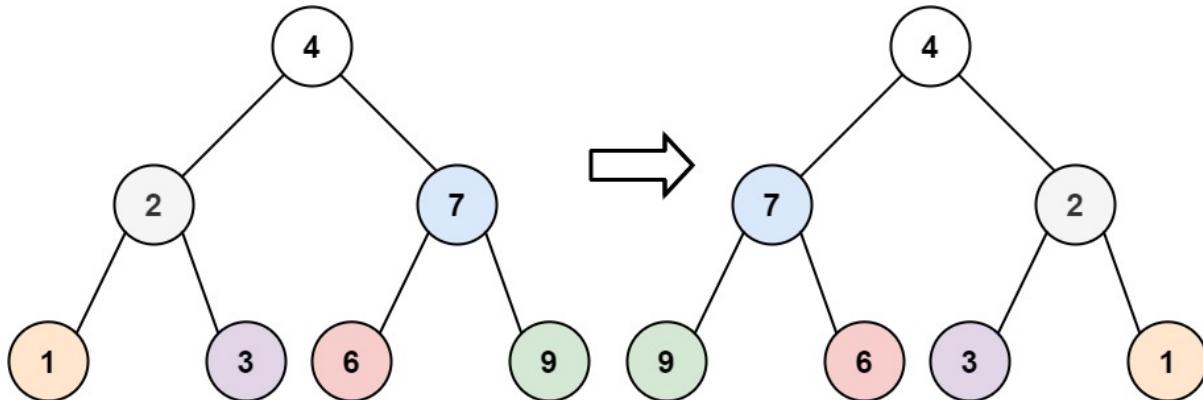
 //maxSq ye bata raha hai ki vaha se kitta bada square ban skta h, i
 slia number of elements k lia maxSq*maxSq
 return maxSq*maxSq;
 }
}

```

# 226. Invert Binary Tree ↗

Given the `root` of a binary tree, invert the tree, and return *its root*.

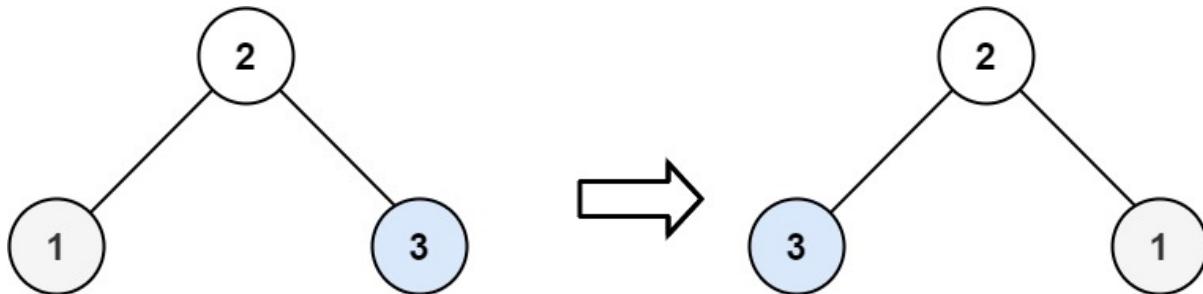
## Example 1:



**Input:** root = [4,2,7,1,3,6,9]

**Output:** [4,7,2,9,6,3,1]

## Example 2:



**Input:** root = [2,1,3]

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

## Example 3:

**Input:** root = []

**Output:** []

## Constraints:

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

```

class Solution {
 public TreeNode invertTree(TreeNode root) {
 if (root == null) return null;

 TreeNode leftSide = invertTree(root.left);
 TreeNode rightSide = invertTree(root.right);

 root.left = rightSide;
 root.right = leftSide;
 return root;
 }

}

```

```

class Solution {
 public TreeNode invertTree(TreeNode root)
 {
 if (root == null) return null;

 TreeNode left = root.left;

 root.left = invertTree(root.right); //left me right set
 root.right = invertTree(left); //right me left
 return root;
 }
}

```

## 228. Summary Ranges ↗

You are given a **sorted unique** integer array `nums`.

Return *the smallest sorted list of ranges that cover all the numbers in the array exactly*. That is, each element of `nums` is covered by exactly one of the ranges, and there is no integer `x` such that `x` is in one of the ranges but not in `nums`.

Each range `[a,b]` in the list should be output as:

- "a->b" if `a != b`
- "a" if `a == b`

### Example 1:

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

**Output:** ["0->2","4->5","7"]

**Explanation:** The ranges are:

[0,2] --> "0->2"

[4,5] --> "4->5"

[7,7] --> "7"

### Example 2:

**Input:** nums = [0,2,3,4,6,8,9]

**Output:** ["0","2->4","6","8->9"]

**Explanation:** The ranges are:

[0,0] --> "0"

[2,4] --> "2->4"

[6,6] --> "6"

[8,9] --> "8->9"

### Example 3:

**Input:** nums = []

**Output:** []

### Example 4:

**Input:** nums = [-1]

**Output:** ["-1"]

### Example 5:

**Input:** nums = [0]

**Output:** ["0"]

### Constraints:

- $0 \leq \text{nums.length} \leq 20$
- $-2^{31} \leq \text{nums}[i] \leq 2^{31} - 1$
- All the values of `nums` are **unique**.
- `nums` is sorted in ascending order.

```
class Solution {
 public List<String> summaryRanges(int[] nums) {

 if (nums == null || nums.length == 0) return new ArrayList<>();

 ArrayList<String> list = new ArrayList<>();
 int curr = nums[0];
 int lowerLimit = nums[0]; //maintains the lower element

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

 while (i < nums.length && curr+1 == nums[i]){ //increasing
 curr = nums[i];
 i++;
 }

 if(i < nums.length) {
 if (curr != lowerLimit){ //range is there

 // String temp = lowerLimit + "->" + curr;

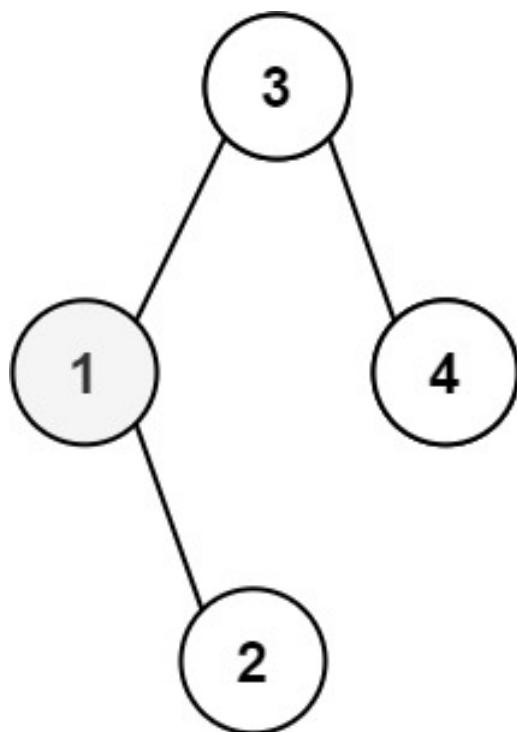
 list.add(lowerLimit + "->" + curr);
 lowerLimit = nums[i];
 curr = nums[i];
 }
 else {
 list.add(lowerLimit+"");
 lowerLimit = nums[i];
 curr = nums[i];
 }
 }
 }
 if (curr != lowerLimit){ //range is there
 String temp = lowerLimit + "->" + curr;
 list.add(temp);
 }

 else {
 list.add(lowerLimit+"");
 }
 return list;
 }
}
```

# 230. Kth Smallest Element in a BST

Given the root of a binary search tree, and an integer  $k$ , return the  $k^{\text{th}}$  smallest value (**1-indexed**) of all the values of the nodes in the tree.

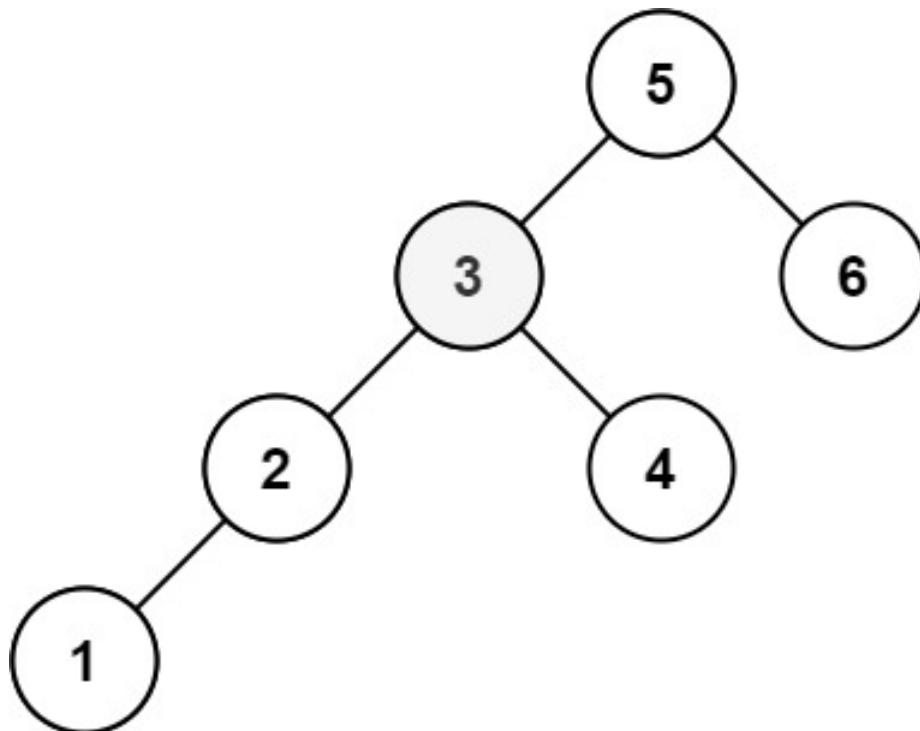
**Example 1:**



**Input:** root = [3,1,4,null,2], k = 1

**Output:** 1

**Example 2:**



**Input:** root = [5,3,6,2,4,null,null,1], k = 3

**Output:** 3

### Constraints:

- The number of nodes in the tree is n .
- $1 \leq k \leq n \leq 10^4$
- $0 \leq \text{Node.val} \leq 10^4$

**Follow up:** If the BST is modified often (i.e., we can do insert and delete operations) and you need to find the kth smallest frequently, how would you optimize?

```
class Solution {
 public int kthSmallest(TreeNode root, int k) {
 PriorityQueue<Integer> maxHeap = preOder(root,k);
 return maxHeap.peek();
 }
 public PriorityQueue<Integer> preOder(TreeNode root, int k)
 {
 List<TreeNode> al = new ArrayList<TreeNode>();
 PriorityQueue<Integer> q = new PriorityQueue<Integer>(Collections.r
everseOrder());
 if (root != null)
 {
 q.add(root.val);

 q.addAll(preOder(root.left, k));
 q.addAll(preOder(root.right, k));

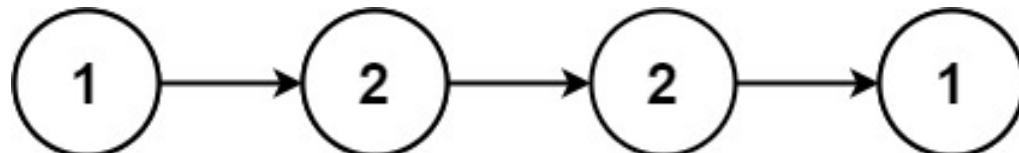
 while (!q.isEmpty() && q.size() > k)
 {
 q.poll();
 }
 }
 return q;
 }
}
```

# 234. Palindrome Linked List ↗



Given the `head` of a singly linked list, return `true` if it is a palindrome.

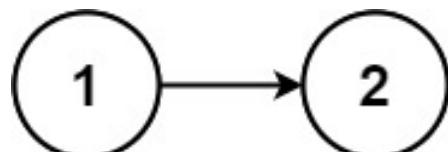
## Example 1:



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

**Output:** true

## Example 2:



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

**Output:** false

## Constraints:

- The number of nodes in the list is in the range  $[1, 10^5]$ .
- $0 \leq \text{Node.val} \leq 9$

**Follow up:** Could you do it in  $O(n)$  time and  $O(1)$  space?

```

class Solution {
 public boolean isPalindrome(ListNode head)
 {

 ListNode sp = head, curr = head; //slow pointer
 ListNode fp = head; //fast pointer

 while (fp != null && fp.next != null)
 {
 sp = sp.next;
 fp = fp.next.next;
 }
 if (fp != null)
 {
 sp = sp.next;
 }
 ListNode mid = reverseList (sp);

 while (mid != null && curr != null)
 {
 if(mid.val != curr.val)
 {
 return false;
 }
 mid = mid.next;
 curr = curr.next;
 }

 return true;
 }
}

```

```

public ListNode reverseList(ListNode head)
{
 ListNode prevNode = null;
 ListNode currNode = head;
 ListNode nextNode = head;

 while (currNode != null)
 {
 nextNode = currNode.next;
 currNode.next = prevNode;
 prevNode = currNode;
 currNode = nextNode;
 }
 head = prevNode;
}

```

```
 return head;
}

}
```

## 236. Lowest Common Ancestor of a Binary Tree

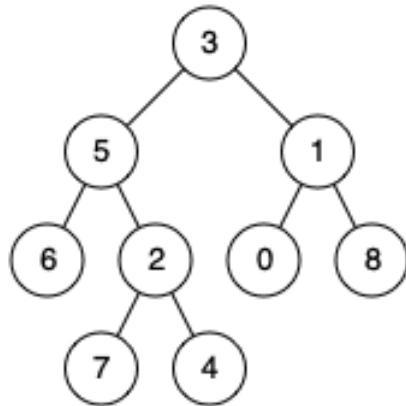


Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree.

According to the definition of LCA on Wikipedia

([https://en.wikipedia.org/wiki/Lowest\\_common\\_ancestor](https://en.wikipedia.org/wiki/Lowest_common_ancestor)): "The lowest common ancestor is defined between two nodes  $p$  and  $q$  as the lowest node in  $T$  that has both  $p$  and  $q$  as descendants (where we allow a node to be a descendant of itself)."

### Example 1:

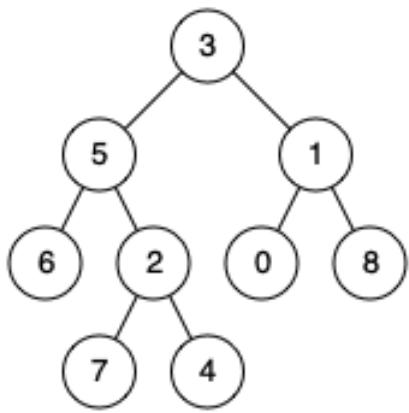


**Input:** root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1

**Output:** 3

**Explanation:** The LCA of nodes 5 and 1 is 3.

### Example 2:



**Input:** root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4

**Output:** 5

**Explanation:** The LCA of nodes 5 and 4 is 5, since a node can be a descendant

### Example 3:

**Input:** root = [1,2], p = 1, q = 2

**Output:** 1

### Constraints:

- The number of nodes in the tree is in the range  $[2, 10^5]$ .
- $-10^9 \leq \text{Node.val} \leq 10^9$
- All `Node.val` are **unique**.
- $p \neq q$
- $p$  and  $q$  will exist in the tree.

```

/**
 * Definition for a binary tree node.
 * public class TreeNode {
 * int val;
 * TreeNode left;
 * TreeNode right;
 * TreeNode(int x) { val = x; }
 * }
 */
class Solution {
 static ArrayList<TreeNode> psf;
 public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q)
 {
 psf = new ArrayList<TreeNode>();
 List<TreeNode> ans1 = new ArrayList<>();
 List<TreeNode> ans2 = new ArrayList<>();

 if (hasPath(root, p)) //p hai kya...hai to psf aa jaega
 {
 ans1 = new ArrayList<>(psf);
 }
 psf = new ArrayList<TreeNode>();
 if (hasPath(root, q)) //q hai kya...hai to psf me aa jaega
 {
 ans2 = new ArrayList<>(psf);
 }
 TreeNode possibleNode = null;
 int size1 = ans1.size()-1;
 int size2 = ans2.size()-1;
 //reverse me traverse kar rai qki list ulti aati h
 while(size1 >= 0 && size2 >= 0 && ans1.get(size1) == ans2.get(size2))
 {
 possibleNode = ans1.get(size1); //possibleNode me last same node
rakte ja rai
 size1--;
 size2--;
 }

 return possibleNode;
 }
 public static boolean hasPath(TreeNode root, TreeNode n)//find n ka pat
h
{
 if (root == null) return false;
}

```

```

 if (root == n)//n mil gaya
 {
 psf.add(root);
 return true;
 }

 boolean left = hasPath(root.left, n);

 if (left) {
 psf.add(root);
 return true;
 }

 boolean right = hasPath(root.right, n);

 if (right) {
 psf.add(root);
 return true;
 }

 return false;
}
}

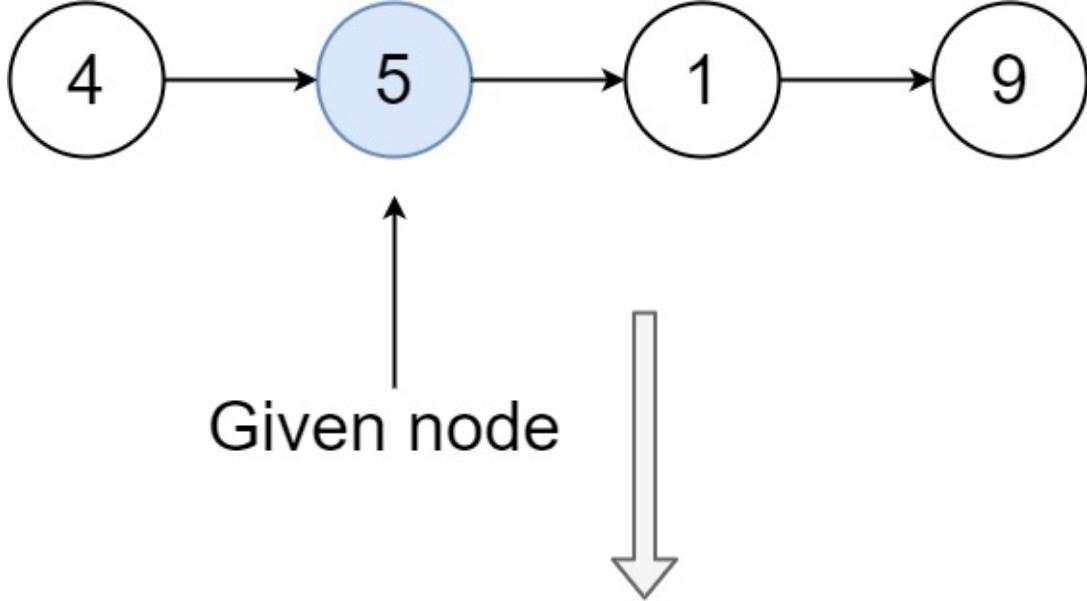
```

## 237. Delete Node in a Linked List ↗

Write a function to **delete a node** in a singly-linked list. You will **not** be given access to the head of the list, instead you will be given access to **the node to be deleted** directly.

It is **guaranteed** that the node to be deleted is **not a tail node** in the list.

**Example 1:**

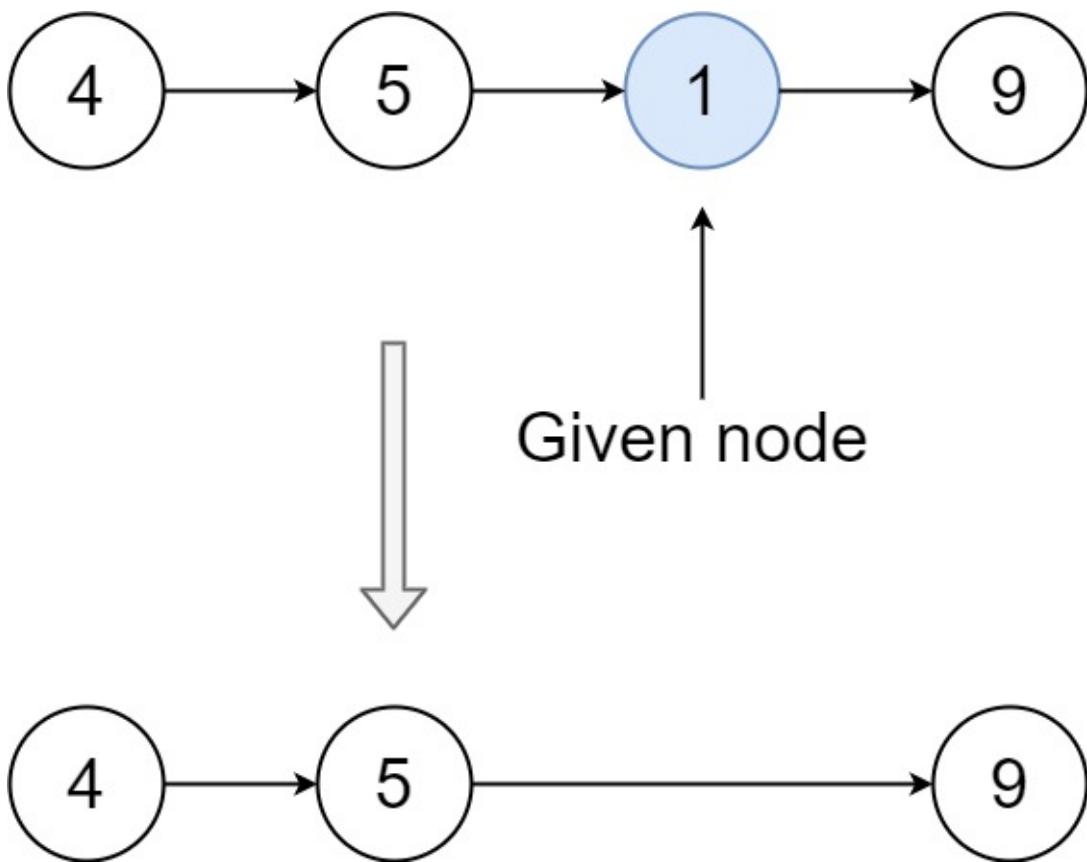


**Input:** head = [4,5,1,9], node = 5

**Output:** [4,1,9]

**Explanation:** You are given the second node with value 5, the linked list shou

### Example 2:



**Input:** head = [4,5,1,9], node = 1

**Output:** [4,5,9]

**Explanation:** You are given the third node with value 1, the linked list shoul

### Example 3:

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

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

### Example 4:

**Input:** head = [0,1], node = 0

**Output:** [1]

### Example 5:

**Input:** head = [-3,5,-99], node = -3

**Output:** [5,-99]

### Constraints:

- The number of the nodes in the given list is in the range [2, 1000].
- $-1000 \leq \text{Node.val} \leq 1000$
- The value of each node in the list is **unique**.
- The node to be deleted is **in the list** and is **not a tail** node

```
class Solution {
 public void deleteNode(ListNode node) {
 node.val=node.next.val;
 node.next=node.next.next;
 }
}
```

## 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 temp = 1;
 int res[] = new int[nums.length];
 for (int i = 0; i < nums.length; i++)
 {
 res[i] = temp;
 temp = temp * nums[i];
 }
 temp = 1;
 for (int j = nums.length -1; j >=0; j--)
 {
 res[j] = res[j] * temp;
 temp = temp * nums[j];
 }

 return res;
 }
}

```

## 240. Search a 2D Matrix II ↗

Write an efficient algorithm that searches for a `target` value in an  $m \times n$  integer matrix. The matrix has the following properties:

- Integers in each row are sorted in ascending from left to right.
- Integers in each column are sorted in ascending from top to bottom.

**Example 1:**

|    |    |    |    |    |
|----|----|----|----|----|
| 1  | 4  | 7  | 11 | 15 |
| 2  | 5  | 8  | 12 | 19 |
| 3  | 6  | 9  | 16 | 22 |
| 10 | 13 | 14 | 17 | 24 |
| 18 | 21 | 23 | 26 | 30 |

**Input:** matrix = [[1,4,7,11,15],[2,5,8,12,19],[3,6,9,16,22],[10,13,14,17,24],]  
**Output:** true

#### Example 2:

|    |    |    |    |    |
|----|----|----|----|----|
| 1  | 4  | 7  | 11 | 15 |
| 2  | 5  | 8  | 12 | 19 |
| 3  | 6  | 9  | 16 | 22 |
| 10 | 13 | 14 | 17 | 24 |
| 18 | 21 | 23 | 26 | 30 |

**Input:** matrix = [[1,4,7,11,15],[2,5,8,12,19],[3,6,9,16,22],[10,13,14,17,24],]  
**Output:** false

#### Constraints:

- $m == \text{matrix.length}$
- $n == \text{matrix}[i].length$
- $1 \leq n, m \leq 300$
- $-10^9 \leq \text{matrix}[i][j] \leq 10^9$
- All the integers in each row are **sorted** in ascending order.
- All the integers in each column are **sorted** in ascending order.
- $-10^9 \leq \text{target} \leq 10^9$

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

 if (matrix == null) return false;

 int row = 0;
 int col = matrix[0].length - 1;

 if (target < matrix[0][0] || target > matrix[matrix.length-1][col])
 return false;

 while (row <= (matrix.length-1) && col >= 0)
 {
 if (target == matrix[row][col]) return true; //target found

 else if (target < matrix[row][col])
 {
 col--;
 }
 else if (target > matrix[row][col])
 {
 row++;
 }

 }

 return false;
 }
}
```

## 242. Valid Anagram ↗



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

### **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) {

 if (s.length() != t.length()) return false;
 char[] schar = s.toCharArray();
 char[] tchar = t.toCharArray();

 char[] chars = new char[26];

 for (char sc : schar)
 {
 chars[sc-'a']++;
 }
 for (char tc : tchar)
 {
 chars[tc-'a']--;
 }
 for (int i = 0; i < chars.length; i++)
 {
 if (chars[i] != 0) return false;
 }
 return true;
 }
}
```

```

class Solution {
 public boolean isAnagram(String s, String t) {

 if (s.length() != t.length())
 {
 return false;
 }

 ArrayList<Character> al = new ArrayList<Character>();
 int i;
 for (i = 0; i < s.length(); i++)
 {
 al.add(s.charAt(i));
 }
 for (i = 0; i < t.length(); i++)
 {
 Character c = t.charAt(i);
 al.remove(c);
 }
 if (al.size() == 0)
 {
 return true;
 }

 return false;
 }
}

```

## 253. Meeting Rooms II



Given an array of meeting time intervals `intervals` where `intervals[i] = [starti, endi]`, return *the minimum number of conference rooms required*.

### Example 1:

**Input:** `intervals = [[0,30],[5,10],[15,20]]`  
**Output:** 2

### Example 2:

**Input:** intervals = [[7,10],[2,4]]

**Output:** 1

### Constraints:

- $1 \leq \text{intervals.length} \leq 10^4$
- $0 \leq \text{start}_i < \text{end}_i \leq 10^6$

```
class Solution {
 public int minMeetingRooms(int[][] intervals) {

 Arrays.sort(intervals, (a,b) -> a[0] - b[0]);

 PriorityQueue<int[]> pq = new PriorityQueue<>((a,b) -> a[1]-b[1]);
 //jo meeting pahle ovr ho re vo upper

 for (int[] meeting : intervals) {

 if (!pq.isEmpty()) {

 int[] top = pq.peek(); //upper ki meeting se compare karege
 if (meeting[0] >= top[1]) { //existing meeting k baad start
 ho re to iski jarurat nahi remove karo...room khali
 pq.remove();
 }
 }
 pq.add(meeting);
 }
 return pq.size(); //jita size hoga utni meeting chal rahi hogi...qki
 jo meeting ovr ho gae thi use humne baher kar dia tha
 }
}
```

## 256. Paint House ↗



There is a row of  $n$  houses, where each house can be painted one of three colors: red, blue, or green. The cost of painting each house with a certain color is different. You have to paint all the houses such that no two adjacent houses have the same color.

The cost of painting each house with a certain color is represented by an `n x 3` cost matrix `costs`.

- For example, `costs[0][0]` is the cost of painting house 0 with the color red; `costs[1][2]` is the cost of painting house 1 with color green, and so on...

Return *the minimum cost to paint all houses.*

### Example 1:

**Input:** `costs = [[17,2,17],[16,16,5],[14,3,19]]`

**Output:** 10

**Explanation:** Paint house 0 into blue, paint house 1 into green, paint house 2 into red.

Minimum cost:  $2 + 5 + 3 = 10$ .

### Example 2:

**Input:** `costs = [[7,6,2]]`

**Output:** 2

### Constraints:

- `costs.length == n`
- `costs[i].length == 3`
- $1 \leq n \leq 100$
- $1 \leq \text{costs}[i][j] \leq 20$

```

/*
17 2 17
16 6 5
14 3 1

 R B G
H1 17 2 17
H2 16 + 2 = 18 6 + 17 = 23 5 + 2 = 7
H3 14 + 7 = 21 3 + 7 = 10 1 + 18 = 19 Min in last Row = 1
0
*/

```

```

class Solution {
 public int minCost(int[][] costs) {

 int[][] houseColor = new int[costs.length][3];
 int costPerRow = 0;
 for (int j = 0; j < 3; j++) { //1st house
 houseColor[0][j] = costs[0][j];

 }

 for (int i = 1; i < costs.length; i++) {
 for (int j = 0; j < 3; j++) {
 if (j == 0){
 houseColor[i][j] = costs[i][j] + Math.min(houseColor[i-1][1], houseColor[i-1][2]);
 }
 else if (j == 1) {
 houseColor[i][j] = costs[i][j] + Math.min(houseColor[i-1][0], houseColor[i-1][2]);
 }
 else if (j == 2) {
 houseColor[i][j] = costs[i][j] + Math.min(houseColor[i-1][0], houseColor[i-1][1]);
 }

 }
 }

 // System.out.println(Arrays.deepToString(houseColor));
 int minCosts = Integer.MAX_VALUE;
 for (int j = 0; j < 3; j++) { //min in row not col***
 minCosts = Math.min(minCosts, houseColor[costs.length-1][j]);
 }
 }
}

```

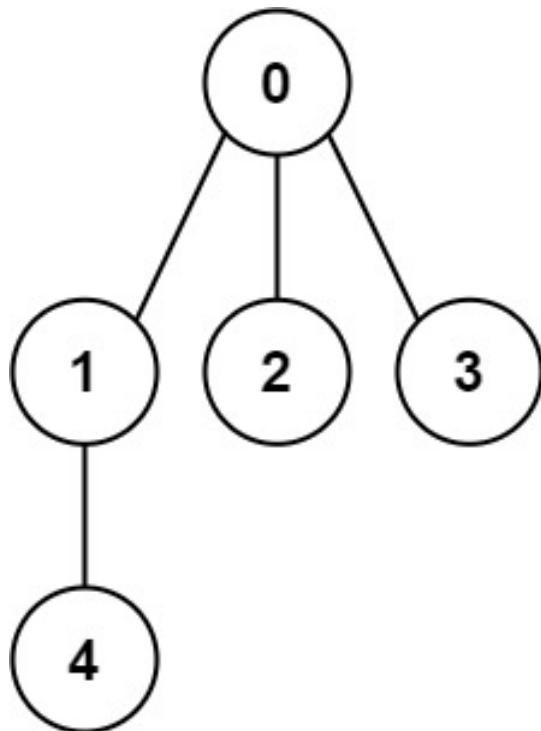
```
 }
 return minCosts;
}
}
```

## 261. Graph Valid Tree ↗

You have a graph of  $n$  nodes labeled from  $0$  to  $n - 1$ . You are given an integer  $n$  and a list of edges where  $\text{edges}[i] = [a_i, b_i]$  indicates that there is an undirected edge between nodes  $a_i$  and  $b_i$  in the graph.

Return `true` if the edges of the given graph make up a valid tree, and `false` otherwise.

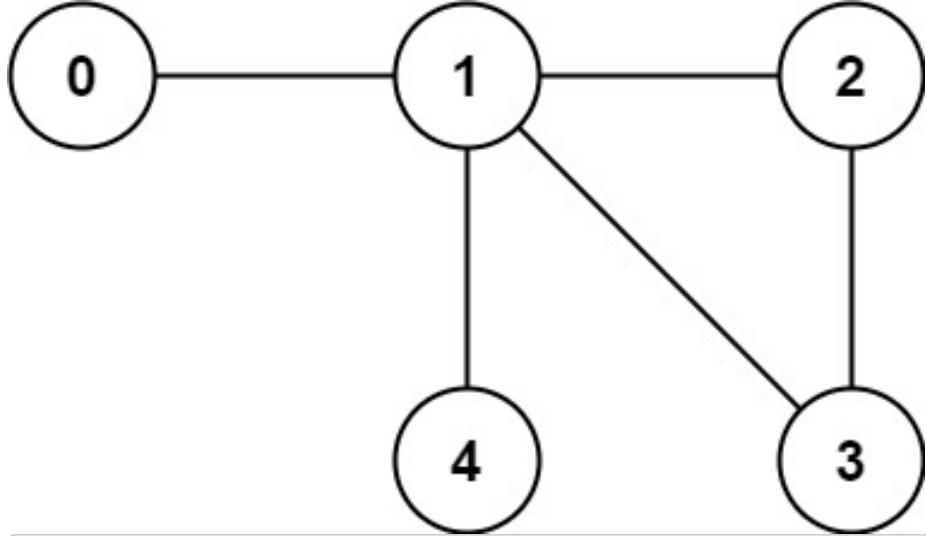
### Example 1:



**Input:**  $n = 5$ ,  $\text{edges} = [[0,1],[0,2],[0,3],[1,4]]$

**Output:** `true`

### Example 2:



**Input:** n = 5, edges = [[0,1],[1,2],[2,3],[1,3],[1,4]]

**Output:** false

### Constraints:

- $1 \leq n \leq 2000$
- $0 \leq \text{edges.length} \leq 5000$
- $\text{edges}[i].length == 2$
- $0 \leq a_i, b_i < n$
- $a_i \neq b_i$
- There are no self-loops or repeated edges.

```

//for a graph to be tree: it should be non cyclic and single component
class Solution {
 public boolean validTree(int n, int[][] edges) {
 if (n > 1 && edges.length == 0) return false;
 List<List<Integer>> adj = new ArrayList<>();
 int[] parent = new int[n];
 Arrays.fill(parent, -1);

 boolean[] visited = new boolean[n];

 //list of list
 for (int i = 0; i < n; i++) {
 adj.add(new ArrayList<Integer>());
 }

 for (int[] edge : edges) {
 adj.get(edge[0]).add(edge[1]);
 adj.get(edge[1]).add(edge[0]);
 parent[edge[1]] = edge[0]; //1 ka parent 0 hai
 }

 int topNode = Integer.MAX_VALUE; //topParent
 for (int i = 0; i < n; i++) {
 if (parent[i] == -1) topNode = i;
 }
 if (topNode == Integer.MAX_VALUE) return false; //top parent he na
hi h*****
 //start from topParent and we should be able to visit all nodes
 if (!visited[0]) {
 boolean isCycle = dfs(adj, topNode, parent, visited);
 if(isCycle) return false;
 }

 for (int i = 1 ; i< n ; i++) {
 if (!visited[i]) return false; //unable to visit all nodes
 }
 return true;
 }

 private boolean dfs(List<List<Integer>> adj, int currNode, int[] parent, boolean[] visited) {

 visited[currNode] = true;
 // List<Integer> neighbour = adj.get(currNode);
 for (int neighbour : adj.get(currNode)) { //for each loop hai ek ek
karke dega****int hai dhyaan

```

```

 //visit ho chuka hai and parent nahi h currnode ka
 if (visited[neighbour] && neighbour != parent[currNode]){
 return true; //cycle
 }

 //visit ho chuka hai and parent h currnode ka islia ignore
 else if (visited[neighbour] && neighbour == parent[currNode]) c
ontinue;

 else {
 dfs(adj, neighbour, parent, visited); //neighbour k lia dek
h rai
 }
 }
 return false; //no cycle
}
}

```

## 268. Missing Number ↗

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

### Example 1:

**Input:** `nums = [3,0,1]`

**Output:** 2

**Explanation:**  $n = 3$  since there are 3 numbers, so all numbers are in the range

### Example 2:

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

**Output:** 2

**Explanation:**  $n = 2$  since there are 2 numbers, so all numbers are in the range

### Example 3:

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

**Output:** 8

**Explanation:** n = 9 since there are 9 numbers, so all numbers are in the range [0, 9].

#### Example 4:

**Input:** nums = [0]

**Output:** 1

**Explanation:** n = 1 since there is 1 number, so all numbers are in the range [0, 1].

#### Constraints:

- $n == \text{nums.length}$
- $1 \leq n \leq 10^4$
- $0 \leq \text{nums}[i] \leq n$
- All the numbers of `nums` are **unique**.

**Follow up:** Could you implement a solution using only  $O(1)$  extra space complexity and  $O(n)$  runtime complexity?

```

class Solution {
 public int missingNumber(int[] nums) {

 int l = nums.length;
 int temp;
 for (int i = 0; i < l; i++)
 {
 while (nums[i] != 0 && nums[i] != i+1)
 {
 temp = nums[i]; //9
 nums[i] = nums[temp-1]; //8 index pr 9 store kia
 nums[temp-1] = temp;
 }
 }

 for (int i = 0; i < l; i++)
 {
 if (nums[i] ==0) return i+1;
 }
 return 0;
 }
}

```

## Second Approach

```

class Solution {
 public int missingNumber(int[] nums) {

 int l = nums.length;
 int totalSum = 0;
 int expectedSum = l*(l+1)/2;

 for(int i = 0; i < nums.length; i++)
 {
 totalSum += nums[i];
 }

 return expectedSum-totalSum;
 }
}

```

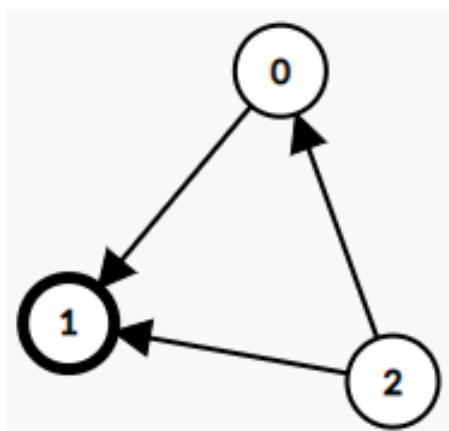
# 277. Find the Celebrity ↗

Suppose you are at a party with  $n$  people (labeled from  $0$  to  $n - 1$ ), and among them, there may exist one celebrity. The definition of a celebrity is that all the other  $n - 1$  people know him/her, but he/she does not know any of them.

Now you want to find out who the celebrity is or verify that there is not one. The only thing you are allowed to do is to ask questions like: "Hi, A. Do you know B?" to get information about whether A knows B. You need to find out the celebrity (or verify there is not one) by asking as few questions as possible (in the asymptotic sense).

You are given a helper function `bool knows(a, b)` which tells you whether A knows B. Implement a function `int findCelebrity(n)`. There will be exactly one celebrity if he/she is in the party. Return the celebrity's label if there is a celebrity in the party. If there is no celebrity, return `-1`.

## Example 1:

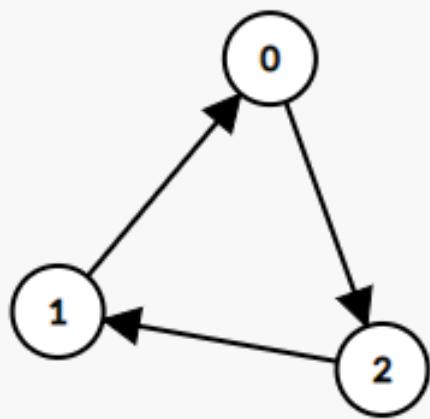


**Input:** graph = [[1,1,0],[0,1,0],[1,1,1]]

**Output:** 1

**Explanation:** There are three persons labeled with 0, 1 and 2. `graph[i][j] = 1`

## Example 2:



**Input:** graph = [[1,0,1],[1,1,0],[0,1,1]]

**Output:** -1

**Explanation:** There is no celebrity.

### Constraints:

- $n == \text{graph.length}$
- $n == \text{graph}[i].length$
- $2 \leq n \leq 100$
- $\text{graph}[i][j]$  is 0 or 1.
- $\text{graph}[i][i] == 1$

**Follow up:** If the maximum number of allowed calls to the API `knows` is  $3 * n$ , could you find a solution without exceeding the maximum number of calls?

```

/* The knows API is defined in the parent class Relation.
 boolean knows(int a, int b); */

public class Solution extends Relation {
 public int findCelebrity(int n) {

 int possibleCeleb = 0;
 for (int i = 1; i < n; i++){
 if (knows(possibleCeleb, i)) { //possibleCeleb agr i ko janta
 h to vo to celeb nahi h...ab i ho skta h islia change kar de rai
 possibleCeleb = i;
 }
 }
 //ab humare pass ek possible celeb h
 //check: kya ise sb log jante h and ye kise ko nahi jante
 for (int i = 0; i < n; i++) {

 if (i!= possibleCeleb && !knows(i, possibleCeleb))
 {
 return -1;//qki i nahi janta ise
 }
 if (i!= possibleCeleb && knows(possibleCeleb, i)) {

 return -1; //qki celeb kise ko nahi janna chaea yaha pr ko
i ko jaan raha
 }
 }

 return possibleCeleb;
 }
}

```

## 278. First Bad Version ↗

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

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

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

### Example 1:

**Input:** n = 5, bad = 4

**Output:** 4

**Explanation:**

call `isBadVersion(3)` -> false

call `isBadVersion(5)` -> true

call `isBadVersion(4)` -> true

Then 4 is the first bad version.

### Example 2:

**Input:** n = 1, bad = 1

**Output:** 1

### Constraints:

- $1 \leq \text{bad} \leq n \leq 2^{31} - 1$

```
/* The isBadVersion API is defined in the parent class VersionControl.
 boolean isBadVersion(int version); */

public class Solution extends VersionControl {
 public int firstBadVersion(int n) {

 int l = 1, h = n, mid;
 int possibleAns = 0;
 while(l<=h)
 {
 mid = l+ (h-l)/2;
 if(isBadVersion(mid))
 {
 possibleAns = mid;
 h = mid-1;
 }
 else
 {
 l = mid + 1;
 }
 }

 return possibleAns;
 }
}
```

```

/* The isBadVersion API is defined in the parent class VersionControl.
 boolean isBadVersion(int version); */

public class Solution extends VersionControl {
 public int firstBadVersion(int n) {

 int l = 1, h = n, mid;
 while(l<=h)
 {
 mid = l+ (h-1)/2;
 if(isBadVersion(mid))
 {
 h = mid-1;
 }
 else
 {
 l = mid + 1;
 }
 }

 return l;
 }
}

```

## 279. Perfect Squares ↗

Given an integer  $n$ , return *the least number of perfect square numbers that sum to  $n$* .

A **perfect square** is an integer that is the square of an integer; in other words, it is the product of some integer with itself. For example, 1, 4, 9, and 16 are perfect squares while 3 and 11 are not.

### Example 1:

**Input:**  $n = 12$

**Output:** 3

**Explanation:**  $12 = 4 + 4 + 4$ .

### Example 2:

**Input:** n = 13

**Output:** 2

**Explanation:** 13 = 4 + 9.

### Constraints:

- $1 \leq n \leq 10^4$

```
//Using dp tabular
class Solution
{
 public int numSquares(int n)
 {
 int dp[] = new int[n+1];
 dp[0] = 0; //0 ka square is 0
 dp[1] = 1; //1 ka sq is 1

 for (int i = 2; i < dp.length; i++)
 {
 int minRem = Integer.MAX_VALUE;//baar baar reinitialize hoga
 for (int j = 1; j*j <= i; j++)
 {
 int rem = i -(j*j);
 minRem = Math.min(minRem, dp[rem]); //remaining karne k minimum tareeke
 }
 dp[i] = minRem +1; //+1 islia qki ek step to humnelia tha... chahey ko 1^2 ho ya 2^2
 }

 return dp[dp.length-1];
 }
}
```

## 283. Move Zeroes ↗



Given an integer array `nums`, move all `0`'s to the end of it while maintaining the relative order of the non-zero elements.

**Note** that you must do this in-place without making a copy of the array.

### Example 1:

```
Input: nums = [0,1,0,3,12]
Output: [1,3,12,0,0]
```

### Example 2:

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

### Constraints:

- $1 \leq \text{nums.length} \leq 10^4$
- $-2^{31} \leq \text{nums}[i] \leq 2^{31} - 1$

**Follow up:** Could you minimize the total number of operations done?

```
class Solution {
 public void moveZeroes(int[] nums) {
 int count = 0;
 int temp;

 // Traverse the array. If arr[i] is
 // non-zero, then swap the element at
 // index 'count' with the element at
 // index 'i'
 for (int i = 0; i < nums.length; i++) {
 if ((nums[i] != 0)) {
 temp = nums[count];
 nums[count] = nums[i];
 nums[i] = temp;
 count = count + 1;
 }
 }
 }
}
```

# 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

## Example 3:

**Input:** `nums = [1,1]`

**Output:** 1

## Example 4:

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

**Output:** 1

## 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?

---

**//Approach: making index negative, if index is already negative that means value was already encountered**

```
class Solution
{
 public int findDuplicate(int[] nums)
 {
 for (int i = 0; i < nums.length; i++)
 {
 if (nums[Math.abs(nums[i]) - 1] > 0)
 {
 nums[Math.abs(nums[i]) - 1] = nums[Math.abs(nums[i]) - 1] *
-1;
 }
 else
 {
 return Math.abs(nums[i]);
 }
 }
 return -1;
 }
}
```

## 289. Game of Life ↗

According to Wikipedia's article ([https://en.wikipedia.org/wiki/Conway%27s\\_Game\\_of\\_Life](https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life)): "The **Game of Life**, also known simply as **Life**, is a cellular automaton devised by the British mathematician John Horton Conway in 1970."

The board is made up of an `m x n` grid of cells, where each cell has an initial state: **live** (represented by a `1`) or **dead** (represented by a `0`). Each cell interacts with its eight neighbors ([https://en.wikipedia.org/wiki/Moore\\_neighborhood](https://en.wikipedia.org/wiki/Moore_neighborhood)) (horizontal, vertical, diagonal) using the following four rules (taken from the above Wikipedia article):

1. Any live cell with fewer than two live neighbors dies as if caused by under-population.

2. Any live cell with two or three live neighbors lives on to the next generation.
3. Any live cell with more than three live neighbors dies, as if by over-population.
4. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

The next state is created by applying the above rules simultaneously to every cell in the current state, where births and deaths occur simultaneously. Given the current state of the  $m \times n$  grid board , return *the next state*.

### Example 1:

The diagram illustrates a 4x3 grid of binary values representing the current state of a board. An arrow points to the right, indicating the transition to the next state. The initial state is as follows:

|   |   |   |
|---|---|---|
| 0 | 1 | 0 |
| 0 | 0 | 1 |
| 1 | 1 | 1 |
| 0 | 0 | 0 |

The final state (next state) is:

|   |   |   |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 0 | 1 | 0 |

**Input:** board = [[0,1,0],[0,0,1],[1,1,1],[0,0,0]]

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

### Example 2:

The diagram illustrates a 2x2 grid of binary values representing the current state of a board. An arrow points to the right, indicating the transition to the next state. The initial state is as follows:

|   |   |
|---|---|
| 1 | 1 |
| 1 | 0 |

The final state (next state) is:

|   |   |
|---|---|
| 1 | 1 |
| 1 | 1 |

**Input:** board = [[1,1],[1,0]]

**Output:** [[1,1],[1,1]]

## **Constraints:**

- $m == \text{board.length}$
- $n == \text{board}[i].length$
- $1 \leq m, n \leq 25$
- $\text{board}[i][j]$  is 0 or 1.

## **Follow up:**

- Could you solve it in-place? Remember that the board needs to be updated simultaneously: You cannot update some cells first and then use their updated values to update other cells.
  - In this question, we represent the board using a 2D array. In principle, the board is infinite, which would cause problems when the active area encroaches upon the border of the array (i.e., live cells reach the border). How would you address these problems?
-

```

class Solution {
 public void gameOfLife(int[][] board) {

 for (int i = 0; i < board.length; i++)
 {
 for (int j = 0; j < board[0].length; j++)
 {
 int aliveCount = checkNumber(board, i, j);

 //Any live cell with fewer than two live neighbors dies
 if (board[i][j] == 1 && aliveCount < 2) board[i][j] = 2;
 //Any live cell with more than three live neighbors dies, a
 s if by over-population
 if (board[i][j] == 1 && aliveCount > 3) board[i][j] = 2;
 //Any dead cell with exactly three live neighbors becomes a
 live cell
 if (board[i][j] == 0 && aliveCount == 3) board[i][j] = 3;
 }
 }
 setBoard(board);
 }

 //Alive to dead make 2
 private int checkNumber(int[][] board, int i, int j)
 {
 int aliveCount = 0;
 int[][] directions = {{1,0}, {-1,0}, {0,1}, {0,-1}, {1,1}, {1,-1},
 {-1,-1}, {-1,1}};
 int row = board.length -1;
 int col = board[0].length -1;
 int alive = 0;

 for (int[] d : directions)
 {
 //out of board
 if (i + d[0] > row || i + d[0] < 0 || j + d[1] > col || j + d
 [1] < 0) continue;
 else if (board[i+ d[0]][j+ d[1]] == 1 || board[i+ d[0]][j+ d
 [1]] == 2) alive++;
 }
 aliveCount = alive;

 return aliveCount;
 }

 private static void setBoard(int[][] board)
}

```

```

 {
 for (int i = 0; i < board.length; i++)
 {
 for (int j = 0; j < board[0].length; j++)
 {
 /*
 2 means was alive now dead
 3 means was dead now alive
 */
 if (board[i][j] == 2) board[i][j] = 0;
 if (board[i][j] == 3) board[i][j] = 1;
 }
 }
 }
}

/*
i+1, j : down row
i-1, j : up row
i, j+1 : right col
i, j-1 : left col
i+1, j+1 : down right dia
i+1, j-1 : down left dia
i-1, j-1 : up left dia
i-1, j+1 : up right dia
System.out.println(Arrays.deepToString(board));
*/

```

## 299. Bulls and Cows ↗

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

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

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

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

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

### Example 1:

**Input:** `secret = "1807"`, `guess = "7810"`

**Output:** "1A3B"

**Explanation:** Bulls are connected with a '|' and cows are underlined:

"1807"

|

"7810"

### Example 2:

**Input:** `secret = "1123"`, `guess = "0111"`

**Output:** "1A1B"

**Explanation:** Bulls are connected with a '|' and cows are underlined:

"1123"            "1123"

|

or

|

"0111"            "0111"

Note that only one of the two unmatched 1s is counted as a cow since the non-

### Example 3:

**Input:** `secret = "1"`, `guess = "0"`

**Output:** "0A0B"

### Example 4:

**Input:** `secret = "1"`, `guess = "1"`

**Output:** "1A0B"

### Constraints:

- $1 \leq \text{secret.length}, \text{guess.length} \leq 1000$
- $\text{secret.length} == \text{guess.length}$
- `secret` and `guess` consist of digits only.

```

class Solution {
 public String getHint(String secret, String guess) {

 int[] cowsInSecret = new int[10]; //0 to 9
 int[] cowsInGuess = new int[10]; //0 to 9

 int totCows = 0;
 int totBulls = 0;

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

 int fromSecret = Character.getNumericValue(secret.charAt(i));
 int fromGuess = Character.getNumericValue(guess.charAt(i));

 if (fromSecret == fromGuess) totBulls++;

 else { //if equal nahi h matlab cows hai
 cowsInSecret[fromSecret]++;
 cowsInGuess[fromGuess]++;
 }
 }

 // System.out.println(Arrays.toString(cowsInSecret));
 // System.out.println(Arrays.toString(cowsInGuess));
 for (int i = 0; i < cowsInSecret.length; i++) {

 totCows += Math.min(cowsInSecret[i], cowsInGuess[i]);
 }

 String result = totBulls + "A" + totCows + "B";
 return result;
 }
}

```

## 300. Longest Increasing Subsequence ↗

Given an integer array `nums`, return the length of the longest strictly increasing subsequence.

A **subsequence** is a sequence that can be derived from an array by deleting some or no elements without changing the order of the remaining elements. For example, `[3,6,2,7]` is a subsequence of the array `[0,3,1,6,2,2,7]`.

### **Example 1:**

**Input:** nums = [10,9,2,5,3,7,101,18]

**Output:** 4

**Explanation:** The longest increasing subsequence is [2,3,7,101], therefore the length is 4.

### **Example 2:**

**Input:** nums = [0,1,0,3,2,3]

**Output:** 4

### **Example 3:**

**Input:** nums = [7,7,7,7,7,7,7]

**Output:** 1

### **Constraints:**

- $1 \leq \text{nums.length} \leq 2500$
- $-10^4 \leq \text{nums}[i] \leq 10^4$

**Follow up:** Can you come up with an algorithm that runs in  $O(n \log(n))$  time complexity?

```

class Solution {
 //Runtime: 73 ms
 public int lengthOfLIS(int[] nums) // [10,9,2,5,3,7,101,18] --> [2,3,7,1
01]
 {
 int dp[] = new int[nums.length];
 dp[0] = 1;

 for (int i = 1; i < nums.length; i++)
 {
 for (int j = i-1; j >= 0; j--)
 {
 if (nums[i] > nums[j])//tb peche lag skta h
 {
 dp[i] = Math.max(dp[i], dp[j] +1);
 }
 else
 {
 dp[i] = Math.max(dp[i], 1);
 //apna seq...agar pahle hum koi value mile hue h to use
 miss ni karna islia Math.max..(hum j ki her val k lia check kar rai...possible h ki kabhi if false ho tb else me direct 1 ni karna)
 }
 }
 }
 int max = Integer.MIN_VALUE;
 for (int i = 0; i < dp.length; i++)
 {
 max = Math.max(max, dp[i]);
 }
 return max;
 }
}

```

## 322. Coin Change ↗

You are given an integer array `coins` representing coins of different denominations and an integer `amount` representing a total amount of money.

Return *the fewest number of coins that you need to make up that amount*. If that amount of money cannot be made up by any combination of the coins, return `-1`.

You may assume that you have an infinite number of each kind of coin.

### **Example 1:**

**Input:** coins = [1,2,5], amount = 11

**Output:** 3

**Explanation:** 11 = 5 + 5 + 1

### **Example 2:**

**Input:** coins = [2], amount = 3

**Output:** -1

### **Example 3:**

**Input:** coins = [1], amount = 0

**Output:** 0

### **Example 4:**

**Input:** coins = [1], amount = 1

**Output:** 1

### **Example 5:**

**Input:** coins = [1], amount = 2

**Output:** 2

### **Constraints:**

- $1 \leq \text{coins.length} \leq 12$
- $1 \leq \text{coins}[i] \leq 2^{31} - 1$
- $0 \leq \text{amount} \leq 10^4$

when we want to print combinations

```
class Solution {

 public int coinChange(int[] coins, int amount) {
 int ans = Integer.MAX_VALUE;
 List<List<Integer>> res = new ArrayList<>();
 generateComb(coins, 0, amount, new ArrayList<Integer>(), res);
 for (List<Integer> l : res){
 ans = Math.min(ans, l.size());
 }
 // System.out.println(res);
 return (ans== Integer.MAX_VALUE)? -1:ans;
 }

 void generateComb(int[] coins, int start, int target, List<Integer> curr, List<List<Integer>> res) {

 if (target == 0) {
 res.add(new ArrayList<Integer>(curr));
 return;
 }

 if (target < 0) return;//ab to koi chance nahi bacha

 for (int i = start; i < coins.length; i++) {
 curr.add(coins[i]);
 generateComb(coins, i, target-coins[i], curr, res);
 curr.remove(curr.size()-1);
 }

 }
}
```

```

class Solution {
 public int coinChange(int[] coins, int amount) {

 int dp[] = new int[amount+1];
 Arrays.fill(dp, amount+1);
 dp[0] = 0; //0 ko py karne k 0 tareke

 for (int i = 1; i < dp.length; i++)
 {
 for (int j = 0; j < coins.length; j++)
 {
 if (coins[j] <= i)//i hai jo pay karna h
 {
 dp[i] = Math.min(dp[i], 1 + dp[i - coins[j]]); //basically 1 rs dene k baad hum dekh rai hai ki bacha hua amt dene ka minimum tareeka
 }
 }
 }

 return (dp[amount] == amount+1)? -1:dp[amount];
 }
}

```

**Used Array.sort to minimize time**

```

class Solution {
 public int coinChange(int[] coins, int amount) {
 Arrays.sort(coins);
 int dp[] = new int[amount+1];
 Arrays.fill(dp, amount+1);
 dp[0] = 0; //0 ko py karne k 0 tareke

 for (int i = 1; i < dp.length; i++)
 {
 for (int j = 0; j < coins.length; j++)
 {
 if (coins[j] <= i)//i hai jo pay karna h
 {
 dp[i] = Math.min(dp[i], 1 + dp[i - coins[j]]); //basically 1 rs dene k baad hum dekh rai hai ki bacha hua amt dene ka minimum tareeka
 }
 else
 {
 break;
 }
 }
 }

 return (dp[amount] == amount+1)? -1:dp[amount];
 }
}

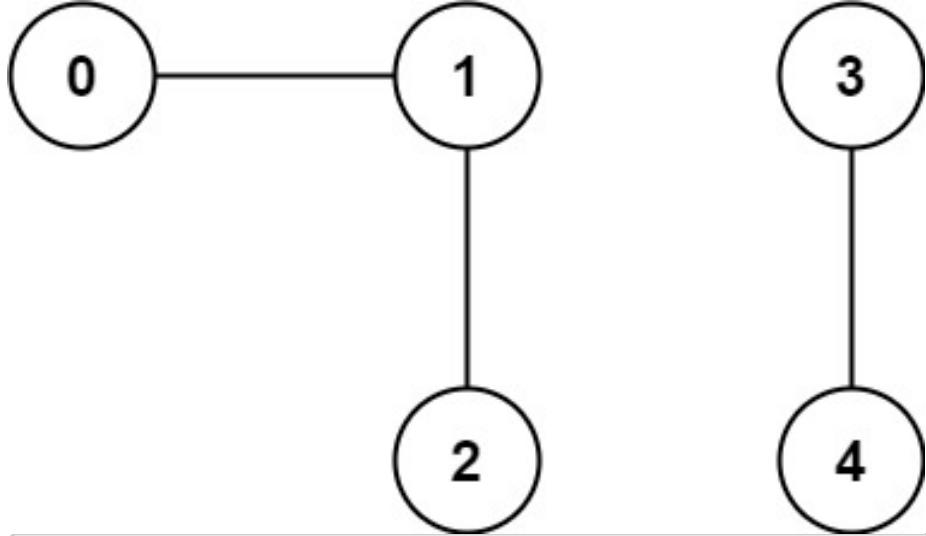
```

## 323. Number of Connected Components in an Undirected Graph ↴ ▼

You have a graph of  $n$  nodes. You are given an integer  $n$  and an array  $\text{edges}$  where  $\text{edges}[i] = [a_i, b_i]$  indicates that there is an edge between  $a_i$  and  $b_i$  in the graph.

Return *the number of connected components in the graph.*

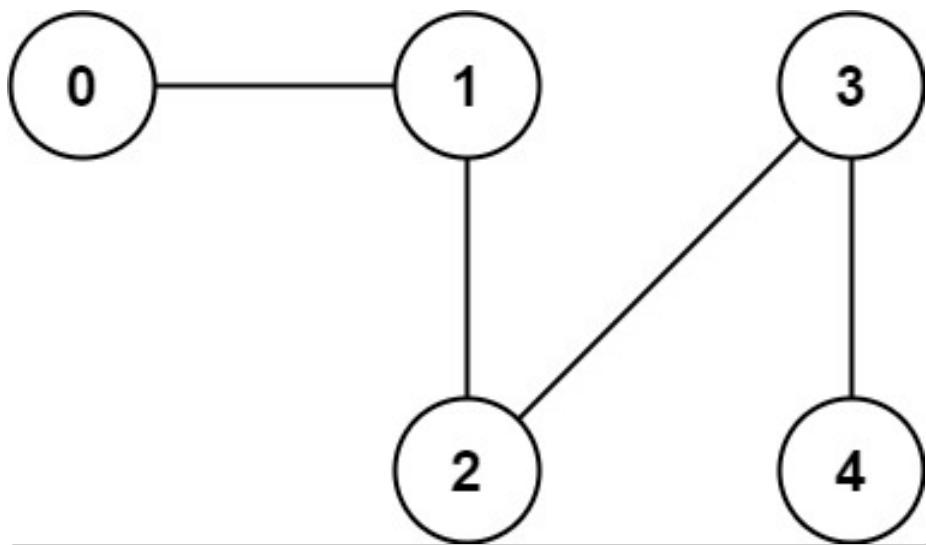
**Example 1:**



**Input:** n = 5, edges = [[0,1],[1,2],[3,4]]

**Output:** 2

### Example 2:



**Input:** n = 5, edges = [[0,1],[1,2],[2,3],[3,4]]

**Output:** 1

### Constraints:

- $1 \leq n \leq 2000$
- $1 \leq \text{edges.length} \leq 5000$
- $\text{edges}[i].length == 2$
- $0 \leq a_i \leq b_i < n$
- $a_i \neq b_i$
- There are no repeated edges.

```

class Solution {
 public int countComponents(int n, int[][] edges) {

 List<List<Integer>> adj = new ArrayList<>();
 boolean[] visited = new boolean[n];
 int count = 0;

 for (int i = 0; i < n; i++) {
 adj.add(new ArrayList<>());
 }
 for (int[] i : edges) {
 adj.get(i[0]).add(i[1]);
 adj.get(i[1]).add(i[0]);
 }

 for (int i = 0; i < n; i++){
 if (!visited[i]) { // jitni baar me pura visit hua utne components hai
 count++;
 dfs(adj, i, visited);
 }
 }
 return count;
 }

 public void dfs(List<List<Integer>> adj, int currNode, boolean[] visited){

 for (int neighbour : adj.get(currNode)) {
 if (!visited[neighbour]) { // visit nahi h to explore karo
 visited[neighbour] = true;
 dfs(adj, neighbour, visited);
 }
 }
 }
}

```



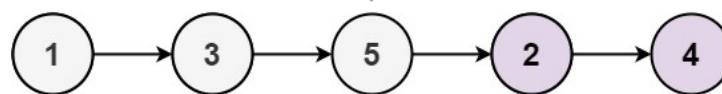
Given the **head** of a singly linked list, group all the nodes with odd indices together followed by the nodes with even indices, and return *the reordered list*.

The **first** node is considered **odd**, and the **second** node is **even**, and so on.

Note that the relative order inside both the even and odd groups should remain as it was in the input.

You must solve the problem in  $O(1)$  extra space complexity and  $O(n)$  time complexity.

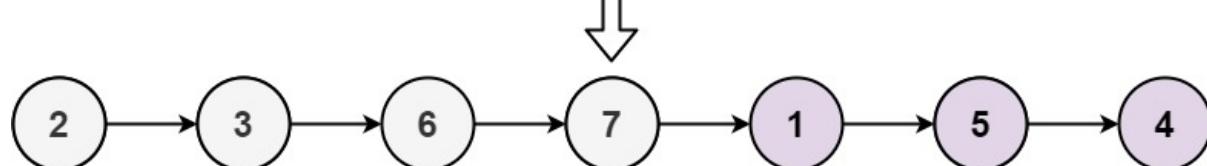
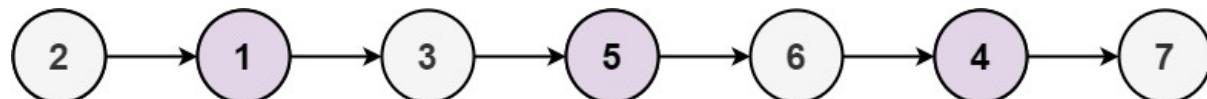
### Example 1:



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

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

### Example 2:



**Input:** head = [2,1,3,5,6,4,7]

**Output:** [2,3,6,7,1,5,4]

### Constraints:

- $n ==$  number of nodes in the linked list
- $0 \leq n \leq 10^4$
- $-10^6 \leq \text{Node.val} \leq 10^6$

```

class Solution {
 public ListNode oddEvenList(ListNode head)
 {

 ListNode evenNode = null;
 ListNode oddNode = head;
 ListNode evenHead = null;

 if (head == null)
 {
 return null;
 }
 if(head.next != null)
 {
 evenNode = head.next;
 evenHead = evenNode;
 }
 while (oddNode.next != null && evenNode.next != null)
 {
 oddNode.next = oddNode.next.next;
 evenNode.next = evenNode.next.next;
 oddNode = oddNode.next;
 evenNode = evenNode.next;

 }
 oddNode.next = evenHead;
 return head;
 }
}

```

## 332. Reconstruct Itinerary ↗

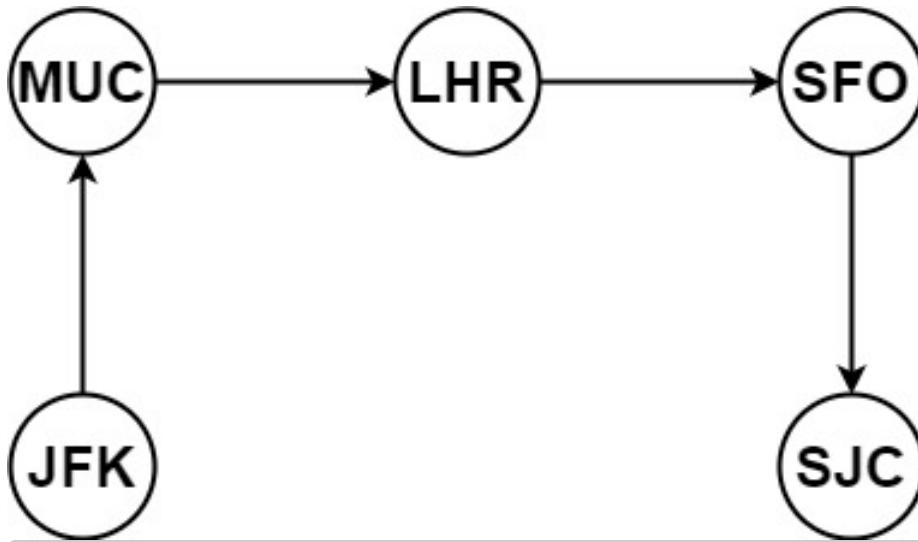
You are given a list of airline tickets where `tickets[i] = [fromi, toi]` represent the departure and the arrival airports of one flight. Reconstruct the itinerary in order and return it.

All of the tickets belong to a man who departs from "JFK", thus, the itinerary must begin with "JFK". If there are multiple valid itineraries, you should return the itinerary that has the smallest lexical order when read as a single string.

- For example, the itinerary ["JFK", "LGA"] has a smaller lexical order than ["JFK", "LGB"].

You may assume all tickets form at least one valid itinerary. You must use all the tickets once and only once.

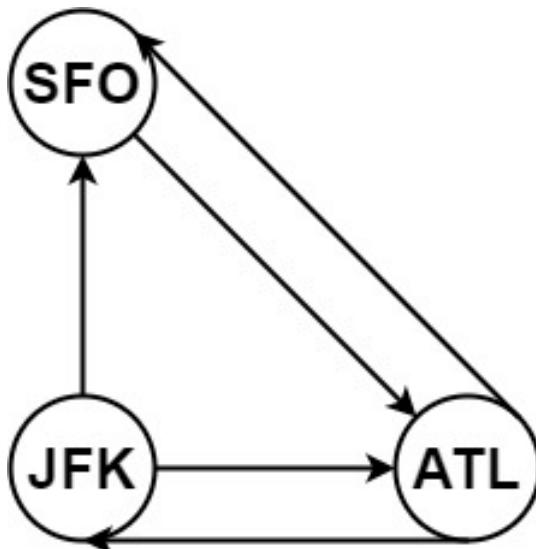
**Example 1:**



**Input:** tickets = [[ "MUC", "LHR" ], [ "JFK", "MUC" ], [ "SFO", "SJC" ], [ "LHR", "SFO" ]]

**Output:** [ "JFK", "MUC", "LHR", "SFO", "SJC" ]

**Example 2:**



**Input:** tickets = [[ "JFK", "SFO" ], [ "JFK", "ATL" ], [ "SFO", "ATL" ], [ "ATL", "JFK" ], [ "ATL", "SFO" ]]

**Output:** [ "JFK", "ATL", "JFK", "SFO", "ATL", "SFO" ]

**Explanation:** Another possible reconstruction is [ "JFK", "SFO", "ATL", "JFK", "ATL", "SFO" ]

**Constraints:**

- $1 \leq \text{tickets.length} \leq 300$
- $\text{tickets}[i].length == 2$

- $\text{from}_i.length == 3$
  - $\text{to}_i.length == 3$
  - $\text{from}_i$  and  $\text{to}_i$  consist of uppercase English letters.
  - $\text{from}_i != \text{to}_i$
-

```

//yaha pr mapping string and pq of strin ki hai to humare pass nodes nahi h
number me
//islia easy karne k lia hashmap banao jo string and PQ ka ho
class Solution {
 public List<String> findItinerary(List<List<String>> tickets) {

 HashMap<String, PriorityQueue<String>> graph = new HashMap<>();
 //find how many places are there

 for (List<String> possdest : tickets) {
 String source = possdest.get(0);
 String dest = possdest.get(1);
 PriorityQueue<String> temp = graph.getOrDefault(source, new PriorityQueue<String>());//0 par source hai..source k samne ki pq lao
 temp.add(dest); //us pq me ye new dest jo 1 pr hai store karo
 graph.put(source, temp); //is source k aage pq put kar dia
 }

 LinkedList<String> path = new LinkedList<>();

 dfs(graph, path, "JFK");
 return path;
 }

 public void dfs(HashMap<String, PriorityQueue<String>> graph, LinkedList<String> path, String currCity) {

 //explore all cities where we can go...and find complete path
 PriorityQueue<String> pq = graph.get(currCity); //map ne pq de dia
 ab isme se remove karo
 while (pq != null && !pq.isEmpty()){ //qki humne hashmap use kia tha

 String currCityyy = pq.remove();
 dfs(graph, path, currCityyy);
 }
 //we have done that city..to ab add karo ans me ...hume lautate time
 //add karna h
 path.addFirst(currCity);
 }
}

```



Given an integer array `nums`, return `true` if there exists a triple of indices  $(i, j, k)$  such that  $i < j < k$  and  $\text{nums}[i] < \text{nums}[j] < \text{nums}[k]$ . If no such indices exists, return `false`.

### Example 1:

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

**Output:** `true`

**Explanation:** Any triplet where  $i < j < k$  is valid.

### Example 2:

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

**Output:** `false`

**Explanation:** No triplet exists.

### Example 3:

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

**Output:** `true`

**Explanation:** The triplet  $(3, 4, 5)$  is valid because  $\text{nums}[3] == 0 < \text{nums}[4] == 4$

### Constraints:

- $1 \leq \text{nums.length} \leq 5 * 10^5$
- $-2^{31} \leq \text{nums}[i] \leq 2^{31} - 1$

**Follow up:** Could you implement a solution that runs in  $O(n)$  time complexity and  $O(1)$  space complexity?

```
class Solution {
 //Runtime: 0 ms
 public boolean increasingTriplet(int[] nums)
 {
 int smallest = Integer.MAX_VALUE, medium = Integer.MAX_VALUE;

 for (int i = 0; i < nums.length; i++)
 {
 if (nums[i] <= smallest) smallest = nums[i]; //first samllest
 "==" kis jisse baraber pr update na ho
 else if (nums[i] <= medium) medium = nums[i];//smallest se bada
 tha medium se chota
 else return true;//uper walo se bada h
 }

 return false;
 }
}
```

## Second Appraoch: Time is more

```

class Solution {
 //Runtime: 111 ms
 public boolean increasingTriplet(int[] nums)
 {
 int dp[] = new int[nums.length];
 dp[0] = 1;
 for (int i = 1; i < nums.length; i++)
 {
 for (int j = 0; j < i; j++)
 {
 if(nums[j] < nums[i])
 {
 dp[i] = Math.max(dp[i], (dp[j]+1));
 }else
 {
 dp[i] = Math.max(dp[i],1); //apna single element..Mat.ma
x arna imp h jisse purani val override na ho jb j increase ho [1,5,0,4,1,3]
 }
 }
 }

 for (int i = 0; i < dp.length; i++)
 {
 if (dp[i] >= 3) return true;
 }
 return false;
 }
}

```

## 340. Longest Substring with At Most K Distinct Characters ↗

Given a string  $s$  and an integer  $k$ , return *the length of the longest substring of  $s$  that contains at most  $k$  distinct characters*.

**Example 1:**

**Input:** s = "eceba", k = 2

**Output:** 3

**Explanation:** The substring is "ece" with length 3.

### Example 2:

**Input:** s = "aa", k = 1

**Output:** 2

**Explanation:** The substring is "aa" with length 2.

### Constraints:

- $1 \leq s.length \leq 5 * 10^4$
  - $0 \leq k \leq 50$
-

```

//variable size sliding window
//while (j < end)
//if (sum < k) end++
//if (sum == k) calculate end++
//if (sum > k) --> while (sum > k) start++ end++
class Solution {
 public int lengthOfLongestSubstringKDistinct(String s, int k) {

 if (k == 0) return 0;
 if (k >= s.length()) return s.length();

 int count = 0;
 int maxCount = Integer.MIN_VALUE;
 HashMap<Character, Integer> map = new HashMap<>();

 int start = 0;
 int end = 0;

 while (end < s.length()) {

 char currChar = s.charAt(end);
 map.put(currChar, map.getOrDefault(currChar, 0) + 1);
 count++;

 if (map.size() < k) { //unique characters < k
 end++;
 }
 else if (map.size() == k) { //found k unique character ab calculation karo

 maxCount = Math.max(maxCount, count);
 end++;
 }
 else if (map.size() > k){ //more unique character remove

 while (map.size() > k) { //move start jb tk window normal nahi hoti

 char startChar = s.charAt(start);
 int freq = map.get(startChar);
 if (freq == 1){
 map.remove(startChar);
 count--;
 }
 else {
 map.put(startChar, map.get(startChar) -1);
 }
 }
 }
 }
 }
}

```

```

 count--;
 }
 start++;
}
end++; //put to upper ho he raha h, bs end badhana kafe h
}
maxCount = Math.max(maxCount, count); //jb nahi add kia tha tb
ek tc fail ho raha tha...qki pure string ki length he unique count k barabe
r t
}
return maxCount;
}
}

```

## 344. Reverse String ↗

Write a function that reverses a string. The input string is given as an array of characters `s`.

### Example 1:

```

Input: s = ["h","e","l","l","o"]
Output: ["o","l","l","e","h"]

```

### Example 2:

```

Input: s = ["H","a","n","n","a","h"]
Output: ["h","a","n","n","a","H"]

```

### Constraints:

- $1 \leq s.length \leq 10^5$
- $s[i]$  is a printable ascii character  
([https://en.wikipedia.org/wiki/ASCII#Printable\\_characters](https://en.wikipedia.org/wiki/ASCII#Printable_characters)).

**Follow up:** Do not allocate extra space for another array. You must do this by modifying the input array in-place ([https://en.wikipedia.org/wiki/In-place\\_algorithm](https://en.wikipedia.org/wiki/In-place_algorithm)) with  $O(1)$  extra memory.

```
class Solution {
 public void reverseString(char[] s) {

 int f = 0, l = s.length -1;
 char c;

 while (f < l)
 {
 c = s[f];
 s[f] = s[l];
 s[l] = c;
 f++;
 l--;
 }
 System.out.println(s);

 }
}
```

## 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:

- $1 \leq \text{nums.length} \leq 10^5$
- $k$  is in the range  $[1, \text{the number of unique elements in the array}]$ .

- 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 i = 0 ; i < nums.length; i++) {
 if (map.containsKey(nums[i])) map.put(nums[i], map.get(nums[i]) +1);
 else map.put(nums[i], 1);
 }

 PriorityQueue<Integer> pq = new PriorityQueue<>((a,b) -> (map.get(a)-map.get(b)));
 for (Map.Entry<Integer, Integer> e : map.entrySet()) {
 pq.add(e.getKey());
 if (pq.size() > k) pq.remove();

 }

 int[] ans = new int[k];
 int i = 0;
 while (pq.size() > 0) {
 ans[i] = pq.remove();
 i++;
 }
 return ans;
 }
}
```

```
class Solution {
 public int[] topKFrequent(int[] nums, int k)
 {
 HashMap<Integer, Integer> hm = new HashMap<>();
 ArrayList<Integer> list = new ArrayList<>();

 for (int i = 0; i < nums.length; i++)
 {
 if (!hm.containsKey(nums[i])) hm.put(nums[i], 1);

 else hm.put(nums[i], hm.get(nums[i]) +1);
 }

 PriorityQueue<Integer> q = new PriorityQueue<Integer>(new Comparator<Integer>()
 {
 @Override
 public int compare(Integer a, Integer b)
 {
 return -(hm.get(a)-hm.get(b));
 }
 });

 for (Map.Entry<Integer, Integer> s : hm.entrySet())
 {
 q.add(s.getKey());
 }
 while (k > 0 && q.size() > 0)
 {
 list.add(q.poll());
 k--;
 }

 int[] ans = new int[list.size()];
 for (int i = 0; i < list.size(); i++)
 {
 ans[i] = list.get(i);
 }

 return ans;
 }
}
```

# 350. Intersection of Two Arrays II

Given two integer arrays `nums1` and `nums2`, return *an array of their intersection*. Each element in the result must appear as many times as it shows in both arrays and you may return the result in **any order**.

## Example 1:

**Input:** `nums1 = [1,2,2,1]`, `nums2 = [2,2]`

**Output:** `[2,2]`

## Example 2:

**Input:** `nums1 = [4,9,5]`, `nums2 = [9,4,9,8,4]`

**Output:** `[4,9]`

**Explanation:** `[9,4]` is also accepted.

## Constraints:

- $1 \leq \text{nums1.length}, \text{nums2.length} \leq 1000$
- $0 \leq \text{nums1}[i], \text{nums2}[i] \leq 1000$

## Follow up:

- What if the given array is already sorted? How would you optimize your algorithm?
- What if `nums1`'s size is small compared to `nums2`'s size? Which algorithm is better?
- What if elements of `nums2` are stored on disk, and the memory is limited such that you cannot load all elements into the memory at once?

```
class Solution {
 public int[] intersect(int[] nums1, int[] nums2)
 {

 ArrayList<Integer> al = new ArrayList<Integer>();
 int l1 = nums1.length;
 int l2 = nums2.length;

 HashMap<Integer, Integer> hm = new HashMap<Integer, Integer>();

 for (int i : nums1)
 {
 if (hm.containsKey(i))
 {
 hm.put(i, hm.get(i)+1);
 }
 else
 {
 hm.put(i, 1);
 }
 }

 for (int i : nums2)

 {
 if (hm.containsKey(i) && hm.get(i) > 0)
 {
 hm.put(i, hm.get(i)-1);
 al.add(i);
 }
 }

 int[] result = new int[al.size()];
 for (int i = 0; i < al.size(); i++)
 {
 result[i] = al.get(i);
 //System.out.print(al.get(i));
 }

 return result;
 }
}
```

# 359. Logger Rate Limiter ↗

Design a logger system that receives a stream of messages along with their timestamps. Each **unique** message should only be printed **at most every 10 seconds** (i.e. a message printed at timestamp  $t$  will prevent other identical messages from being printed until timestamp  $t + 10$ ).

All messages will come in chronological order. Several messages may arrive at the same timestamp.

Implement the `Logger` class:

- `Logger()` Initializes the logger object.
- `bool shouldPrintMessage(int timestamp, string message)` Returns true if the message should be printed in the given timestamp, otherwise returns false .

## Example 1:

### Input

```
["Logger", "shouldPrintMessage", "shouldPrintMessage", "shouldPrintMessage",
[], [1, "foo"], [2, "bar"], [3, "foo"], [8, "bar"], [10, "foo"], [11, "foo"]]
```

### Output

```
[null, true, true, false, false, true]
```

### Explanation

```
Logger logger = new Logger();

logger.shouldPrintMessage(1, "foo"); // return true, next allowed timestamp
logger.shouldPrintMessage(2, "bar"); // return true, next allowed timestamp
logger.shouldPrintMessage(3, "foo"); // 3 < 11, return false
logger.shouldPrintMessage(8, "bar"); // 8 < 12, return false
logger.shouldPrintMessage(10, "foo"); // 10 < 11, return false
logger.shouldPrintMessage(11, "foo"); // 11 >= 11, return true, next allowed
```

## Constraints:

- $0 \leq \text{timestamp} \leq 10^9$
- Every timestamp will be passed in non-decreasing order (chronological order).
- $1 \leq \text{message.length} \leq 30$
- At most  $10^4$  calls will be made to `shouldPrintMessage` .

```

class Logger {

 HashMap<String, Integer> log;
 public Logger() {
 log = new HashMap();
 }

 public boolean shouldPrintMessage(int timestamp, String message) {

 if (!log.containsKey(message)) {
 log.put(message, timestamp);
 return true;
 }
 int time = log.get(message);
 if (timestamp - time >= 10) {
 log.put(message, timestamp);
 return true;
 }

 return false;
 }
}

/**
 * Your Logger object will be instantiated and called as such:
 * Logger obj = new Logger();
 * boolean param_1 = obj.shouldPrintMessage(timestamp,message);
 */

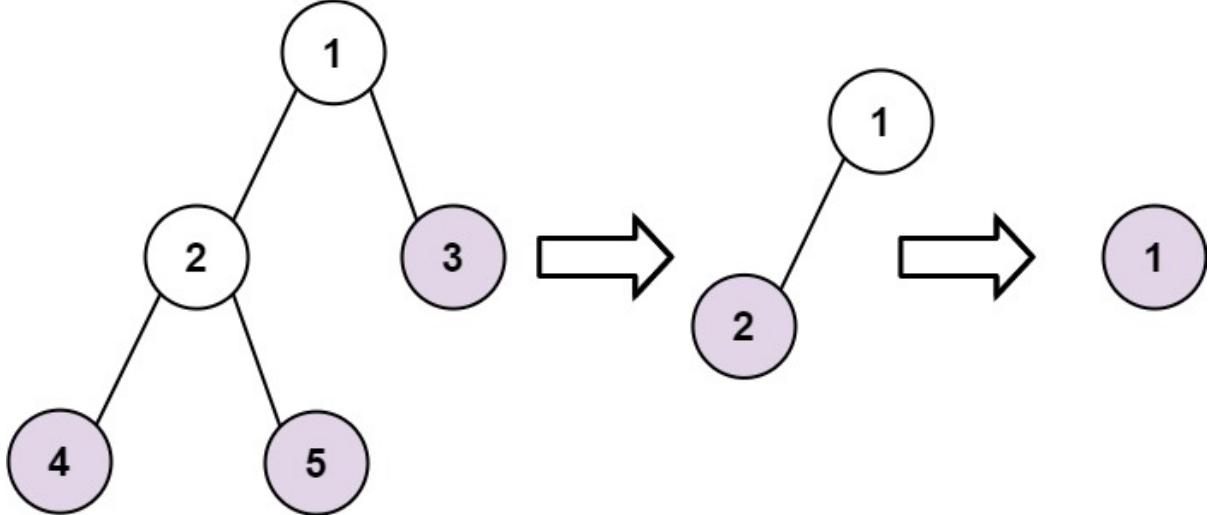
```

## 366. Find Leaves of Binary Tree ↗

Given the `root` of a binary tree, collect a tree's nodes as if you were doing this:

- Collect all the leaf nodes.
- Remove all the leaf nodes.
- Repeat until the tree is empty.

### Example 1:



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

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

**Explanation:**

[[3,5,4],[2],[1]] and [[3,4,5],[2],[1]] are also considered correct answers s

### Example 2:

**Input:** root = [1]

**Output:** [[1]]

### Constraints:

- The number of nodes in the tree is in the range [1, 100].
- $-100 \leq \text{Node.val} \leq 100$

```

//Mistakes: I tried adding in empty list which gave me error
//hume ek ek level he mil raha tha isliya her new level pr index badha do
/*
if (res.size() < height) {
 res.add(new ArrayList<>());
}
*/
class Solution {
 public List<List<Integer>> findLeaves(TreeNode root) {

 List<List<Integer>> res = new ArrayList<>(100);

 findHeight(root, res);
 return res;
 }
 int findHeight(TreeNode node, List<List<Integer>> res) {

 if (node == null) return 0;

 int leftHeight = findHeight(node.left, res);
 int rightHeight = findHeight(node.right, res);

 int height = 1+ Math.max(leftHeight, rightHeight);
 if (res.size() < height) {
 res.add(new ArrayList<>());
 }
 res.get(height-1).add(node.val);
 return height;
 }
}

```

## 378. Kth Smallest Element in a Sorted Matrix ↗

Given an  $n \times n$  matrix where each of the rows and columns are sorted in ascending order, return the  $k^{\text{th}}$  smallest element in the matrix.

Note that it is the  $k^{\text{th}}$  smallest element **in the sorted order**, not the  $k^{\text{th}}$  **distinct** element.

**Example 1:**

**Input:** matrix = [[1,5,9],[10,11,13],[12,13,15]], k = 8

**Output:** 13

**Explanation:** The elements in the matrix are [1,5,9,10,11,12,13,13,15], and th

### Example 2:

**Input:** matrix = [[-5]], k = 1

**Output:** -5

### Constraints:

- $n == \text{matrix.length}$
- $n == \text{matrix}[i].length$
- $1 \leq n \leq 300$
- $-10^9 \leq \text{matrix}[i][j] \leq 10^9$
- All the rows and columns of `matrix` are **guaranteed** to be sorted in **non-decreasing order**.
- $1 \leq k \leq n^2$

```

class Solution {
 public int kthSmallest(int[][] matrix, int k)
 {
 //max heap
 PriorityQueue<Integer> pq = new PriorityQueue<Integer>(Collections.reverseOrder());

 for (int i = 0; i < matrix.length; i++)
 {
 for (int j = 0; j < matrix.length; j++)
 {
 pq.add(matrix[i][j]);
 if (pq.size() > k)
 {
 pq.poll();
 }
 }
 }
 return pq.peek();
 }
}

```

## 380. Insert Delete GetRandom O(1) ↗

Implement the RandomizedSet class:

- `RandomizedSet()` Initializes the `RandomizedSet` object.
- `bool insert(int val)` Inserts an item `val` into the set if not present. Returns `true` if the item was not present, `false` otherwise.
- `bool remove(int val)` Removes an item `val` from the set if present. Returns `true` if the item was present, `false` otherwise.
- `int getRandom()` Returns a random element from the current set of elements (it's guaranteed that at least one element exists when this method is called). Each element must have the **same probability** of being returned.

You must implement the functions of the class such that each function works in **average**  $O(1)$  time complexity.

**Example 1:**

### Input

```
["RandomizedSet", "insert", "remove", "insert", "getRandom", "remove", "inser
[[], [1], [2], [2], [], [1], [2], []]
```

### Output

```
[null, true, false, true, 2, true, false, 2]
```

### Explanation

```
RandomizedSet randomizedSet = new RandomizedSet();
randomizedSet.insert(1); // Inserts 1 to the set. Returns true as 1 was inser
randomizedSet.remove(2); // Returns false as 2 does not exist in the set.
randomizedSet.insert(2); // Inserts 2 to the set, returns true. Set now conta
randomizedSet.getRandom(); // getRandom() should return either 1 or 2 randoml
randomizedSet.remove(1); // Removes 1 from the set, returns true. Set now con
randomizedSet.insert(2); // 2 was already in the set, so return false.
randomizedSet.getRandom(); // Since 2 is the only number in the set, getRandc
```

### Constraints:

- $-2^{31} \leq \text{val} \leq 2^{31} - 1$
- At most  $2 * 10^5$  calls will be made to `insert`, `remove`, and `getRandom`.
- There will be **at least one** element in the data structure when `getRandom` is called.

```
class RandomizedSet {

 Map<Integer, Integer> map;
 List<Integer> list;

 public RandomizedSet() {

 map = new HashMap<Integer, Integer>();
 list = new ArrayList<Integer>();
 }

 public boolean insert(int val) { //Key:val :::: val:index

 if (map.containsKey(val)) {
 return false;
 }
 int index = list.size();
 list.add(val);

 map.put(val, index);
 return true;
 }

 public boolean remove(int val) {

 if (!map.containsKey(val)) return false;

 int indexOfVal = map.get(val);
 int lastValueInList = list.get(list.size()-1);

 list.set(indexOfVal, lastValueInList); //copy last value to val index
 list.remove(list.size()-1); //remove last value in list

 map.put(lastValueInList, indexOfVal); //update index in map
 map.remove(val); //remove val from map

 return true;
 }

 public int getRandom() {
 Random random = new Random();
 int n = random.nextInt(list.size()); //generate index
 return list.get(n); //return value at that index
 }
}
```

```
/**
 * Your RandomizedSet object will be instantiated and called as such:
 * RandomizedSet obj = new RandomizedSet();
 * boolean param_1 = obj.insert(val);
 * boolean param_2 = obj.remove(val);
 * int param_3 = obj.getRandom();
 */
```

## 384. Shuffle an Array ↗

Given an integer array `nums`, design an algorithm to randomly shuffle the array. All permutations of the array should be **equally likely** as a result of the shuffling.

Implement the `Solution` class:

- `Solution(int[] nums)` Initializes the object with the integer array `nums`.
- `int[] reset()` Resets the array to its original configuration and returns it.
- `int[] shuffle()` Returns a random shuffling of the array.

### Example 1:

#### Input

```
["Solution", "shuffle", "reset", "shuffle"]
[[[1, 2, 3]], [], [], []]
```

#### Output

```
[null, [3, 1, 2], [1, 2, 3], [1, 3, 2]]
```

#### Explanation

```
Solution solution = new Solution([1, 2, 3]);
solution.shuffle(); // Shuffle the array [1,2,3] and return its result.
 // Any permutation of [1,2,3] must be equally likely
 // Example: return [3, 1, 2]
solution.reset(); // Resets the array back to its original configuration
solution.shuffle(); // Returns the random shuffling of array [1,2,3]. Exam
```

## Constraints:

- $1 \leq \text{nums.length} \leq 200$
  - $-10^6 \leq \text{nums}[i] \leq 10^6$
  - All the elements of `nums` are **unique**.
  - At most  $5 * 10^4$  calls **in total** will be made to `reset` and `shuffle`.
-

```
class Solution {

 int[] nums;
 Random random;
 public Solution(int[] nums) {
 this.nums = nums;
 random = new Random();

 }

 /** Resets the array to its original configuration and return it. */
 public int[] reset() {
 return this.nums;

 }

 /** Returns a random shuffling of the array. */
 public int[] shuffle() {

 int start = 0, end = this.nums.length-1;
 int[] copy = this.nums.clone();

 while (end > 0) {

 int index = random.nextInt(end+1); //The nextInt(int n) is used
 to get a random number between 0(inclusive) and the number passed in this a
 rgument(n), exclusive

 int temp = copy[end];
 copy[end] = copy[index];
 copy[index] = temp;
 end--;
 }

 return copy;
 }
}

/**
 * Your Solution object will be instantiated and called as such:
 * Solution obj = new Solution(nums);
 * int[] param_1 = obj.reset();
 * int[] param_2 = obj.shuffle();
 */
```

# 387. First Unique Character in a String ↗



Given a string `s`, find the first non-repeating character in it and return its index. If it does not exist, return `-1`.

## Example 1:

**Input:** s = "leetcode"

**Output:** 0

## Example 2:

**Input:** s = "loveleetcode"

**Output:** 2

## Example 3:

**Input:** s = "aabb"

**Output:** -1

## Constraints:

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

```
class Solution {
 public int firstUniqChar(String s) {

 if (null == s || s.isEmpty ()) return -1;

 int a[] = new int [26];

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

 char c = s.charAt(i);
 int hash = c - 'a';

 if (a[hash] == 0)
 {
 a[hash] = 1;
 }
 else if (a[hash] == 1)
 {
 a[hash] = -(1);
 }
 }

 int minIndex = Integer.MAX_VALUE;
 for (int i = 0; i < a.length; i++)
 {
 char ch = 'a';
 if (a[i] == 1)
 {
 ch = (char)(ch + i);

 if (s.indexOf(ch) == 0) return s.indexOf(ch);

 int idx = s.indexOf(ch);
 if (idx > 0)
 {
 if (minIndex > idx)
 {
 minIndex = idx;
 }
 }
 }
 }
 }

 return minIndex == Integer.MAX_VALUE ? -1 : minIndex;
}
```

}

## Second Approach

```
class Solution {
 public int firstUniqChar(String s) {

 int l = s.length();
 boolean found = false;
 for(int i = 0; i < s.length(); i++)
 {
 CharSequence c = s.subSequence(i, i+1);
 String temp = s.substring(0, i) + s.substring(i+1, l);
 if(!temp.contains(c))
 {
 //System.out.println("First Non Repeating Character " + c +
"at index " + i);
 found = true;

 return i;
 }
 }
 return -1;

 }
}
```

...

## 394. Decode String ↗

Given an encoded string, return its decoded string.

The encoding rule is:  $k[encoded\_string]$  , where the `encoded_string` inside the square brackets is being repeated exactly `k` times. Note that `k` is guaranteed to be a positive integer.

You may assume that the input string is always valid; No extra white spaces, square brackets are well-formed, etc.

Furthermore, you may assume that the original data does not contain any digits and that digits are only for those repeat numbers,  $k$ . For example, there won't be input like `3a` or `2[4]`.

### Example 1:

**Input:** `s = "3[a]2[bc]"`

**Output:** `"aaabcabc"`

### Example 2:

**Input:** `s = "3[a2[c]]"`

**Output:** `"accaccacc"`

### Example 3:

**Input:** `s = "2[abc]3[cd]ef"`

**Output:** `"abcabccdcdcdef"`

### Example 4:

**Input:** `s = "abc3[cd]xyz"`

**Output:** `"abcccdcdxyz"`

### Constraints:

- $1 \leq s.length \leq 30$
- `s` consists of lowercase English letters, digits, and square brackets `'[]'`.
- `s` is guaranteed to be a **valid** input.
- All the integers in `s` are in the range `[1, 300]`.

```

/*
Character.getNumericValue --> character to int
Integer.parseInt(num) ----> String to int
*/

```

```

class Solution {
 public String decodeString(String s) {

 Stack<Character> st = new Stack<>();
 for (int i = 0; i < s.length(); i++) {

 if (s.charAt(i) != ']') { //jb tk end nahi mil raha push karo
 st.push(s.charAt(i));
 }
 else if (s.charAt(i) == ']') { //ab process karna h
 String temp = "";

 while (st.peek() != '[') { //jb tk [nahi h tb tk string
 temp += st.pop();
 }
 st.pop(); //poping [isko bhi baher nikalo

 //now get number
 String number = "";
 while (!st.isEmpty() && Character.getNumericValue(st.peek()) >= 0 && Character.getNumericValue(st.peek()) <= 9) { //valid number

 number = st.pop() + number;
 }

 String received = multiplyString(number, temp); //now put this back in stack
 //put ans back aage k lia... character by character
 for (int k = received.length() - 1; k >= 0; k--) { //***ulta
push karna h
 st.push(received.charAt(k));
 }
 }
 }
 StringBuffer sb = new StringBuffer();

 while (!st.isEmpty()) {
 sb.append(st.pop());
 }
 sb.reverse();
 return sb.toString();
 }
}

```

```

 }

private String multiplyString(String num, String s) {

 int resultNum = Integer.parseInt(num);

 StringBuffer ans = new StringBuffer();
 for (int i = 0; i < resultNum; i++) {
 ans.append(s);
 }
 return ans.toString();
}

```

## 410. Split Array Largest Sum ↗

Given an array `nums` which consists of non-negative integers and an integer `m`, you can split the array into `m` non-empty continuous subarrays.

Write an algorithm to minimize the largest sum among these `m` subarrays.

### Example 1:

**Input:** `nums = [7,2,5,10,8]`, `m = 2`

**Output:** 18

**Explanation:**

There are four ways to split `nums` into two subarrays.

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

### Example 2:

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

**Output:** 9

### Example 3:

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

**Output:** 4

### **Constraints:**

- $1 \leq \text{nums.length} \leq 1000$
  - $0 \leq \text{nums}[i] \leq 10^6$
  - $1 \leq m \leq \min(50, \text{nums.length})$
-

```

//Binary Search : many blunders I did..though I knew the solution [forgot to store possible and in possible sum and returned mid]
class Solution {
 public int splitArray(int[] nums, int m) {
 int max = Integer.MIN_VALUE;
 int sum = 0;
 //Find Range
 for (int i = 0; i < nums.length; i++) {
 max = Math.max(max, nums[i]);
 sum += nums[i];
 }

 int low = max;
 int high = sum;
 int mid = 0;
 int possibleSum = 0;
 while (low <= high) { //<= imp hai...jb bhi mid+1 and mid-1 kar rai tb <= use karo
 mid = low + (high - low)/2;
 if (isPossible(nums, m , mid)) { //matlab ye mid possible candidate tha..ab dekh rai isse chota possible hai kya
 high = mid -1;
 possibleSum = mid;
 }
 else {
 low = mid +1;
 }
 }
 return possibleSum;
 }

 private boolean isPossible(int[] nums, int m, int maxSum)
 {
 int count = 1;
 int sum = 0;
 for (int i = 0; i < nums.length; i++) {
 sum += nums[i];
 if (sum > maxSum) {
 count++;
 sum = nums[i]; //next starting sum will include this
 }
 }

 //#[7,2,5,10,8], m = 2 is example me suppose maxSum 15 aaya tb 3 subarray lage...islia keh rai ki maxSum chota hai... use badhao...18 k lia a socho... negative condition socho(15)
 }
}

```

```
 return (count > m)? false : true; //count > m --> maxSum kam ha
i.... 15
 }
}
```

## 414. Third Maximum Number ↗

Given an integer array `nums`, return *the third distinct maximum number in this array. If the third maximum does not exist, return the maximum number.*

### Example 1:

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

**Output:** 1

**Explanation:**

The first distinct maximum is 3.

The second distinct maximum is 2.

The third distinct maximum is 1.

### Example 2:

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

**Output:** 2

**Explanation:**

The first distinct maximum is 2.

The second distinct maximum is 1.

The third distinct maximum does not exist, so the maximum (2) is returned instead.

### Example 3:

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

**Output:** 1

**Explanation:**

The first distinct maximum is 3.

The second distinct maximum is 2 (both 2's are counted together since they have the same value).

The third distinct maximum is 1.

## Constraints:

- $1 \leq \text{nums.length} \leq 10^4$
- $-2^{31} \leq \text{nums}[i] \leq 2^{31} - 1$

**Follow up:** Can you find an  $O(n)$  solution?

```
class Solution { public int thirdMax(int[] nums) {
```

```
 int temp = 0;
 //Using min heap
 PriorityQueue<Integer> pQueue = new PriorityQueue<Integer>();
 for(int i = 0; i < nums.length; i++)
 {
 if (!pQueue.contains(nums[i]))
 pQueue.add(nums[i]);

 if (pQueue.size() > 3)
 {
 pQueue.poll();
 }
 }
 if (pQueue.size() > 2)
 {
 return pQueue.poll();
 }
 else
 {
 while(!pQueue.isEmpty())
 {
 temp = pQueue.poll();
 }
 }
 return temp;
}
```

```
}
```

# 415. Add Strings ↗

Given two non-negative integers, `num1` and `num2` represented as string, return *the sum of num1 and num2 as a string*.

You must solve the problem without using any built-in library for handling large integers (such as `BigInteger`). You must also not convert the inputs to integers directly.

## Example 1:

**Input:** `num1 = "11", num2 = "123"`

**Output:** `"134"`

## Example 2:

**Input:** `num1 = "456", num2 = "77"`

**Output:** `"533"`

## Example 3:

**Input:** `num1 = "0", num2 = "0"`

**Output:** `"0"`

## Constraints:

- $1 \leq \text{num1.length}, \text{num2.length} \leq 10^4$
- `num1` and `num2` consist of only digits.
- `num1` and `num2` don't have any leading zeros except for the zero itself.

```
class Solution {
 public String addStrings(String num1, String num2) {

 StringBuffer sb = new StringBuffer() //

 int i = num1.length() -1;
 int j = num2.length() -1;
 int carry = 0;

 while (i >= 0 || j >= 0 || carry != 0) {

 int ivalue = (i >= 0)? num1.charAt(i) - '0' : 0; //if indexOutO
fBound then put 0
 int jvalue = (j >= 0)? num2.charAt(j) - '0' : 0;

 int sum = ivalue + jvalue + carry;
 int rem = sum %10;
 carry = sum/10;

 sb.append(rem);

 i--;
 j--;
 }

 return sb.reverse().toString();
 }
}
```

```
class Solution {
 public String addStrings(String num1, String num2) {

 int i = num1.length() -1; //last digit in num1
 int j = num2.length() -1; //last digit in num2
 int carry = 0, sum = 0;
 StringBuilder sb = new StringBuilder();
 while (i >= 0 && j >= 0) //jb tk dono number hai
 {
 int n1 = num1.charAt(i--) - '0'; //way to convert char'5' to its int 5
 int n2 = num2.charAt(j--) - '0';

 sum = carry + n1 + n2;

 carry = 0; //carry use ho gaya to ab 0 karo
 if (sum >9)
 {
 sb.append(sum%10);
 carry = sum/10;
 }
 else
 {
 sb.append(sum);
 }
 }
 while (i >= 0) //n1 bacha hai n2 over
 {
 int n1 = num1.charAt(i--) - '0';
 sum = carry + n1;
 carry = 0; //carry use ho gaya to ab 0 karo
 if (sum >9)
 {
 sb.append(sum%10);
 carry = sum/10;
 }
 else
 {
 sb.append(sum);
 }
 }
 while (j >= 0) //n2 bacha hai n1 over
 {
 int n2 = num2.charAt(j--) - '0';
 sum = carry + n2;
 carry = 0; //carry use ho gaya to ab 0 karo
 }
 return sb.reverse().toString();
 }
}
```

```

 if (sum >9)
 {
 sb.append(sum%10);
 carry = sum/10;
 }
 else
 {
 sb.append(sum);
 }
 }

 if (carry != 0) //agar add karne k baad bhi carry h to use add kar
 o...eg.. 22+44 will give carry and we have to add it
 {
 sb.append(carry);
 }

 return sb.reverse().toString();
}

}

```

## 416. Partition Equal Subset Sum ↗

Given a **non-empty** array `nums` containing **only positive integers**, find if the array can be partitioned into two subsets such that the sum of elements in both subsets is equal.

### Example 1:

**Input:** `nums = [1,5,11,5]`

**Output:** `true`

**Explanation:** The array can be partitioned as `[1, 5, 5]` and `[11]`.

### Example 2:

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

**Output:** `false`

**Explanation:** The array cannot be partitioned into equal sum subsets.

## Constraints:

- $1 \leq \text{nums.length} \leq 200$
- $1 \leq \text{nums}[i] \leq 100$

```
class Solution {
 public boolean canPartition(int[] nums) {
 int totalSum = 0;
 for (int i = 0; i < nums.length; i++) {
 totalSum += nums[i];
 }
 if (totalSum%2 != 0) return false;

 return generateSubsets(0, nums, totalSum/2, new HashMap<Integer, Boolean>());
 }

 private boolean generateSubsets(int start, int[] nums, int target, HashMap<Integer, Boolean> memo) {

 if (memo.containsKey(target)) return memo.get(target);
 if (target == 0) return true;
 if (target < 0) return false;

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

 boolean isPossible = generateSubsets(i+1, nums, target - nums[i], memo);
 memo.put(target, isPossible); //true false jo bhi h store karo
 if(isPossible) return true;

 }
 return false;
 }
}
```

## 419. Battleships in a Board ↗

Given an  $m \times n$  matrix board where each cell is a battleship 'X' or empty '.', return the number of the **battleships** on board.

**Battleships** can only be placed horizontally or vertically on board . In other words, they can only be made of the shape  $1 \times k$  ( 1 row,  $k$  columns) or  $k \times 1$  (  $k$  rows, 1 column), where  $k$  can be of any size. At least one horizontal or vertical cell separates between two battleships (i.e., there are no adjacent battleships).

**Example 1:**

|   |  |  |   |
|---|--|--|---|
| X |  |  | X |
|   |  |  | X |
|   |  |  | X |
|   |  |  |   |

**Input:** board = [["X",".",".","X"],[".",".",".","X"],[".",".",".","X"]]

**Output:** 2

**Example 2:**

**Input:** board = [["."]]

**Output:** 0

**Constraints:**

- $m == \text{board.length}$
- $n == \text{board}[i].length$
- $1 \leq m, n \leq 200$
- $\text{board}[i][j]$  is either '.' or 'X' .

**Follow up:** Could you do it in one-pass, using only  $O(1)$  extra memory and without modifying the values board ?

```
//negative case nahi h ques me mention h
//afr left me 'x' hai tb value nahi badhege
//agr up pr 'X' hai tb bhi value nahi badhege
//if k ander if is better than combining both
class Solution {
 public int countBattleships(char[][] board) {

 int row = board.length;
 int col = board[0].length;
 int battleships = 0;
 for (int i = 0; i < row; i++) {
 for (int j = 0; j < col; j++) {

 if (board[i][j] == 'X') {

 if (i == 0){ //bs left he
 if (j == 0 || board[i][j-1] != 'X')
 battleships++;
 }
 else if (j == 0) { //bs up he
 if (i == 0 || board[i-1][j] != 'X')
 battleships++;
 }
 else{
 if (board[i-1][j] != 'X' && board[i][j-1] != 'X')
 battleships++;
 }
 }
 }
 }
 return battleships;
 }
}
```

## 438. Find All Anagrams in a String ↗

Given two strings  $s$  and  $p$ , return an array of all the start indices of  $p$ 's anagrams in  $s$ . You may 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:** s = "cbaebabacd", p = "abc"

**Output:** [0,6]

**Explanation:**

The substring with start index = 0 is "cba", which is an anagram of "abc".

The substring with start index = 6 is "bac", which is an anagram of "abc".

### Example 2:

**Input:** s = "abab", p = "ab"

**Output:** [0,1,2]

**Explanation:**

The substring with start index = 0 is "ab", which is an anagram of "ab".

The substring with start index = 1 is "ba", which is an anagram of "ab".

The substring with start index = 2 is "ab", which is an anagram of "ab".

### Constraints:

- $1 \leq s.length, p.length \leq 3 * 10^4$
- s and p consist of lowercase English letters.

```

//Sliding Window
class Solution {
 public List<Integer> findAnagrams(String s, String p) {

 int lenP = p.length();
 int lenS = s.length();
 ArrayList<Integer> ans = new ArrayList<>();

 int[] arrP = new int[26];
 int[] arrS = new int[26];

 for (char c : p.toCharArray())//fill array with p string freq
 {
 int index = c - 'a';
 arrP[index]++;
 }

 int i = 0, j = 0;
 while (j < lenS)
 {
 if (j - i + 1 <= lenP)
 {
 int index = s.charAt(j) - 'a';
 arrS[index]++;
 if (Arrays.equals(arrS, arrP)) ans.add(i); //both array equal annagram found
 j++;
 }
 else if (j - i + 1 > lenP) //moving i forward
 {
 int index = s.charAt(i) - 'a';
 arrS[index]--;
 i++;
 }
 }
 return ans;
 }
}

```

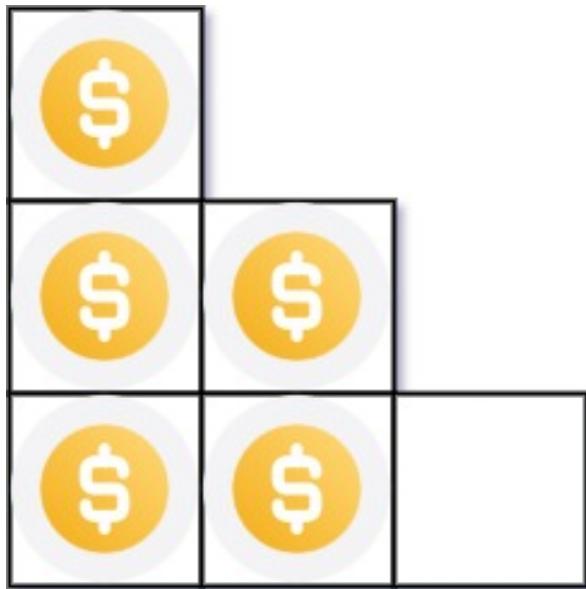
## 441. Arranging Coins ↗



You have  $n$  coins and you want to build a staircase with these coins. The staircase consists of  $k$  rows where the  $i^{\text{th}}$  row has exactly  $i$  coins. The last row of the staircase **may be** incomplete.

Given the integer  $n$ , return *the number of complete rows of the staircase you will build*.

### Example 1:

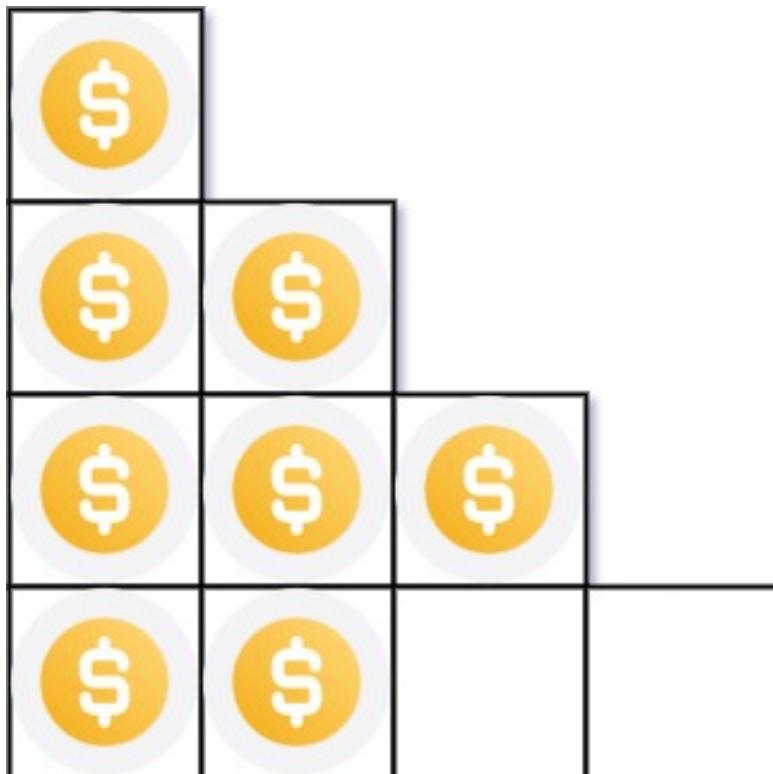


**Input:**  $n = 5$

**Output:** 2

**Explanation:** Because the 3<sup>rd</sup> row is incomplete, we return 2.

### Example 2:



**Input:** n = 8

**Output:** 3

**Explanation:** Because the 4<sup>th</sup> row is incomplete, we return 3.

### Constraints:

- $1 \leq n \leq 2^{31} - 1$

```
class Solution {
 public int arrangeCoins(int n) {

 int c = 1;
 while (n-c >= 0)
 {
 n = n-c;
 c++;
 }
 return (c-1);
 }
}
```

## 442. Find All Duplicates in an Array ↗

Given an integer array `nums` of length `n` where all the integers of `nums` are in the range `[1, n]` and each integer appears **once** or **twice**, return *an array of all the integers that appears twice*.

You must write an algorithm that runs in `O(n)` time and uses only constant extra space.

### Example 1:

**Input:** `nums` = [4,3,2,7,8,2,3,1]

**Output:** [2,3]

### Example 2:

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

**Output:** [1]

### Example 3:

**Input:** nums = [1]

**Output:** []

### Constraints:

- n == nums.length
- 1 <= n <=  $10^5$
- 1 <= nums[i] <= n
- Each element in nums appears **once** or **twice**.

```
class Solution {
 public List<Integer> findDuplicates(int[] nums) {
 List<Integer> ans = new ArrayList<>();
 for (int i = 0; i < nums.length; i++) {

 int index = Math.abs(nums[i]); //coz number may be negative
 if (nums[index-1] < 0) { //already encountered and we have made
it negative that time
 ans.add(index);
 }
 else { //make negative when see 1st time
 nums[index-1] = nums[index-1] *-1;
 }
 }
 return ans;
 }
}
```

## 443. String Compression ↗



Given an array of characters `chars`, compress it using the following algorithm:

Begin with an empty string `s`. For each group of **consecutive repeating characters** in `chars`:

- If the group's length is `1`, append the character to `s`.
- Otherwise, append the character followed by the group's length.

The compressed string `s` **should not be returned separately**, but instead, be stored **in the input character array `chars`**. Note that group lengths that are `10` or longer will be split into multiple characters in `chars`.

After you are done **modifying the input array**, return *the new length of the array*.

You must write an algorithm that uses only constant extra space.

### Example 1:

**Input:** `chars = ["a", "a", "b", "b", "c", "c", "c"]`

**Output:** Return `6`, and the first `6` characters of the input array should be: `["`

**Explanation:** The groups are `"aa"`, `"bb"`, and `"ccc"`. This compresses to `"a2b2c3"`

### Example 2:

**Input:** `chars = ["a"]`

**Output:** Return `1`, and the first character of the input array should be: `["a"]`

**Explanation:** The only group is `"a"`, which remains uncompressed since it's a single character.

### Example 3:

**Input:** `chars = ["a", "b", "b", "b", "b", "b", "b", "b", "b", "b", "b"]`

**Output:** Return `4`, and the first `4` characters of the input array should be: `["`

**Explanation:** The groups are `"a"` and `"bbbbbbbbbb"`. This compresses to `"ab12"`

### Example 4:

**Input:** `chars = ["a", "a", "a", "b", "b", "a", "a"]`

**Output:** Return `6`, and the first `6` characters of the input array should be: `["`

**Explanation:** The groups are `"aaa"`, `"bb"`, and `"aa"`. This compresses to `"a3b2a2"`

### **Constraints:**

- $1 \leq \text{chars.length} \leq 2000$
  - $\text{chars}[i]$  is a lowercase English letter, uppercase English letter, digit, or symbol.
-

```

//so many corner cases
class Solution {
 public int compress(char[] chars) {

 if (chars.length == 1) return 1;

 int charindex = 0;

 int count = 1;

 char prevchar = chars[0];

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

 if (chars[i] != prevchar && count > 1) { //matlab fre likhna h

 chars[charindex++] = prevchar; //jo index pr character rakh
na h
 prevchar = chars[i]; //updated to new character which is di
ff from prev

 String cvalue = count + "";//better string me convert kar k
update karo..double digits bhi handle hogu
 for (int ii = 0; ii < cvalue.length(); ii++) {
 chars[charindex++] = cvalue.charAt(ii);
 }
 count = 1;
 }
 else if (chars[i] != prevchar && count == 1) { //matlab freq na
hi likhna

 chars[charindex++] = prevchar;
 prevchar = chars[i]; //updated to new character which is di
ff from prev
 count = 1;

 }
 else if (chars[i] == prevchar) {

 count++;
 }
 }
 chars[charindex++] = prevchar;
 }
}

```

```

 String cvalue = count + "";
 for (int ii = 0; ii < cvalue.length() && count > 1; ii++) { //yaha
count bhi check kia h
 chars[charindex++] = cvalue.charAt(ii);
 }
 return charindex;
 }
}

```

## 448. Find All Numbers Disappeared in an Array



Given an array `nums` of  $n$  integers where `nums[i]` is in the range  $[1, n]$ , return *an array of all the integers in the range  $[1, n]$  that do not appear in `nums`*.

### Example 1:

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

### Example 2:

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

### Constraints:

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

**Follow up:** Could you do it without extra space and in  $O(n)$  runtime? You may assume the returned list does not count as extra space.

```

class Solution {
 public List<Integer> findDisappearedNumbers(int[] nums)
 {
 ArrayList<Integer> al = new ArrayList<Integer>();
 int[] count = new int[nums.length+1];
 for(int i=0;i<nums.length;i++){
 count[nums[i]]++;
 }
 for(int i=1;i<=nums.length;i++){
 if(count[i]==0){
 al.add(i);
 }
 }
 return al;
 }
}

```

## 451. Sort Characters By Frequency ↗

Given a string `s`, sort it in **decreasing order** based on the **frequency** of the characters. The **frequency** of a character is the number of times it appears in the string.

Return *the sorted string*. If there are multiple answers, return *any of them*.

### Example 1:

**Input:** s = "tree"

**Output:** "eert"

**Explanation:** 'e' appears twice while 'r' and 't' both appear once.  
So 'e' must appear before both 'r' and 't'. Therefore "eetr" is also a valid

### Example 2:

**Input:** s = "cccaaa"

**Output:** "aaaccc"

**Explanation:** Both 'c' and 'a' appear three times, so both "cccaaa" and "aaacc

Note that "cacaca" is incorrect, as the same characters must be together.

### Example 3:

**Input:** s = "Aabb"

**Output:** "bbAa"

**Explanation:** "bbaA" is also a valid answer, but "Aabb" is incorrect.

Note that 'A' and 'a' are treated as two different characters.

### Constraints:

- $1 \leq s.length \leq 5 * 10^5$
- s consists of uppercase and lowercase English letters and digits.

### Used PriorityQueue for MaxHeap

```

class Solution {
 public String frequencySort(String s) {

 String st= "";
 PriorityQueue<Map.Entry<Character, Integer>> maxHeap = new PriorityQueue
 Queue<>((a,b)->(b.getValue()-a.getValue()));

 HashMap<Character, Integer> map = new HashMap<Character, Integer>
 ();
 for (int i = 0; i < s.length(); i++)
 {
 map.put(s.charAt(i), map.getOrDefault(s.charAt(i), 0)+1);
 }

 maxHeap.addAll(map.entrySet());

 while(maxHeap.size() > 0)
 {
 java.util.Map.Entry<Character, Integer> e = maxHeap.poll();
 for (int i = 0; i < e.getValue(); i++)
 {
 st = st + (e.getKey());
 //System.out.print(e.getKey());
 }
 }
 return st;
 }
}

```

## 462. Minimum Moves to Equal Array Elements



Given an integer array `nums` of size `n`, return *the minimum number of moves required to make all array elements equal*.

In one move, you can increment or decrement an element of the array by `1`.

Test cases are designed so that the answer will fit in a **32-bit** integer.

**Example 1:**

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

**Output:** 2

**Explanation:**

Only two moves are needed (remember each move increments or decrements one element)  
[1,2,3] => [2,2,3] => [2,2,2]

## Example 2:

**Input:** nums = [1,10,2,9]

**Output:** 16

## Constraints:

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

```

class Solution {
 public int minMoves2(int[] nums) {

 Arrays.sort(nums);

 int size = nums.length;
 int mid;
 if (size%2 == 0) //even
 {
 mid = nums[size/2];
 }
 else
 {
 mid = nums[Math.min((int)Math.ceil(size/2), (size/2))];
 }

 int minSteps = 0;
 for (int i = 0; i < nums.length; i++)
 {
 minSteps = minSteps + (Math.abs(nums[i] - mid));
 }

 return minSteps;
 }
}

```

## 482. License Key Formatting ↗

You are given a license key represented as a string `s` that consists of only alphanumeric characters and dashes. The string is separated into  $n + 1$  groups by  $n$  dashes. You are also given an integer `k`.

We want to reformat the string `s` such that each group contains exactly `k` characters, except for the first group, which could be shorter than `k` but still must contain at least one character. Furthermore, there must be a dash inserted between two groups, and you should convert all lowercase letters to uppercase.

Return *the reformatted license key*.

**Example 1:**

**Input:** s = "5F3Z-2e-9-w", k = 4

**Output:** "5F3Z-2E9W"

**Explanation:** The string s has been split into two parts, each part has 4 char

Note that the two extra dashes are not needed and can be removed.

### **Example 2:**

**Input:** s = "2-5g-3-J", k = 2

**Output:** "2-5G-3J"

**Explanation:** The string s has been split into three parts, each part has 2 ch

### **Constraints:**

- $1 \leq s.length \leq 10^5$
- s consists of English letters, digits, and dashes '-' .
- $1 \leq k \leq 10^4$

```

class Solution {
 public String licenseKeyFormatting(String s, int k) {

 int end = s.length()-1; // "5F3Z-2e-9-w"
 int count = 0;
 StringBuffer ans = new StringBuffer();

 while (end >= 0) {

 char currChar = Character.toUpperCase(s.charAt(end));
 if (currChar == '-') end--; // skipping -

 else if (count == k) { // having grp of k character now add -
 ans.append('-');
 count = 0;
 }
 else if (count < k) {
 ans.append(currChar);
 count++;
 end--;
 }

 }
 return ans.reverse().toString();
 }
}

```

## 516. Longest Palindromic Subsequence ↗

Given a string  $s$ , find the longest palindromic **subsequence**'s length in  $s$ .

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

### Example 1:

**Input:**  $s = "bbbab"$

**Output:** 4

**Explanation:** One possible longest palindromic subsequence is "bbbb".

## **Example 2:**

**Input:** s = "cbbd"

**Output:** 2

**Explanation:** One possible longest palindromic subsequence is "bb".

### **Constraints:**

- $1 \leq s.length \leq 1000$
  - s consists only of lowercase English letters.
-

```

class Solution {
 public int longestPalindromeSubseq(String s)
 {
 int l = s.length();
 int dp[][] = new int[l][l];

 char[] chars = s.toCharArray();

 for (int gap = 0; gap < l; gap++)
 {
 for (int i = 0; i+gap < l; i++)
 {
 if (gap == 0)
 {
 dp[i][i+gap] = 1;
 }
 else if (gap ==1)
 {
 if (chars[i] == chars[i+gap])
 {
 dp[i][i+gap] = 2; //dono character equal hai
 }
 else dp[i][i+gap] = 1;
 }
 else
 {
 if (chars[i] == chars[i+gap]) //diagonal + 2 (down left dia)
 {
 dp[i][i+gap] = dp[i+1][i+gap-1] +2;
 }
 else
 {
 dp[i][i+gap] = Math.max(dp[i+1][i+gap], dp[i][i+gap-1]); //max down and left
 }
 }
 }
 }
 return dp[0][l-1];
 }
}

```

# 520. Detect Capital ↗



We define the usage of capitals in a word to be right when one of the following cases holds:

- All letters in this word are capitals, like "USA" .
- All letters in this word are not capitals, like "leetcode" .
- Only the first letter in this word is capital, like "Google" .

Given a string `word` , return `true` if the usage of capitals in it is right.

## Example 1:

**Input:** word = "USA"

**Output:** true

## Example 2:

**Input:** word = "FlaG"

**Output:** false

## Constraints:

- $1 \leq \text{word.length} \leq 100$
- `word` consists of lowercase and uppercase English letters.



```
class Solution {
 public boolean detectCapitalUse(String word) {

 if (word.length() == 1) return true;

 char[] charArr = word.toCharArray();
 boolean firstIsSmall = false;
 boolean secondIsSmall = false;

 if (charArr[0] >= 97 && charArr[0] <= 122){
 firstIsSmall = true;
 }
 if (charArr[1] >= 97 && charArr[1] <= 122){
 secondIsSmall = true;
 }

 if (charArr.length == 2 && firstIsSmall && !secondIsSmall) return f
else;
 else if (charArr.length <= 2) return true;

 if (firstIsSmall && secondIsSmall) { //both are small
 for (int i = 2; i < charArr.length; i++) {
 if (!(charArr[i] >= 97 && charArr[i] <= 122)) return false;
 }
 return true;
 }

 else if (!firstIsSmall && !secondIsSmall) { //both are cap
 for (int i = 2; i < charArr.length; i++) {
 if (!(charArr[i] >= 65 && charArr[i] <= 90)) return false;
 }
 return true;
 }

 else if (!firstIsSmall && secondIsSmall) { //1st cap and second sma
11
 for (int i = 2; i < charArr.length; i++) {
 if (!(charArr[i] >= 97 && charArr[i] <= 122)) return false;
 }
 return true;
 }
 else return false;

 }
}
```

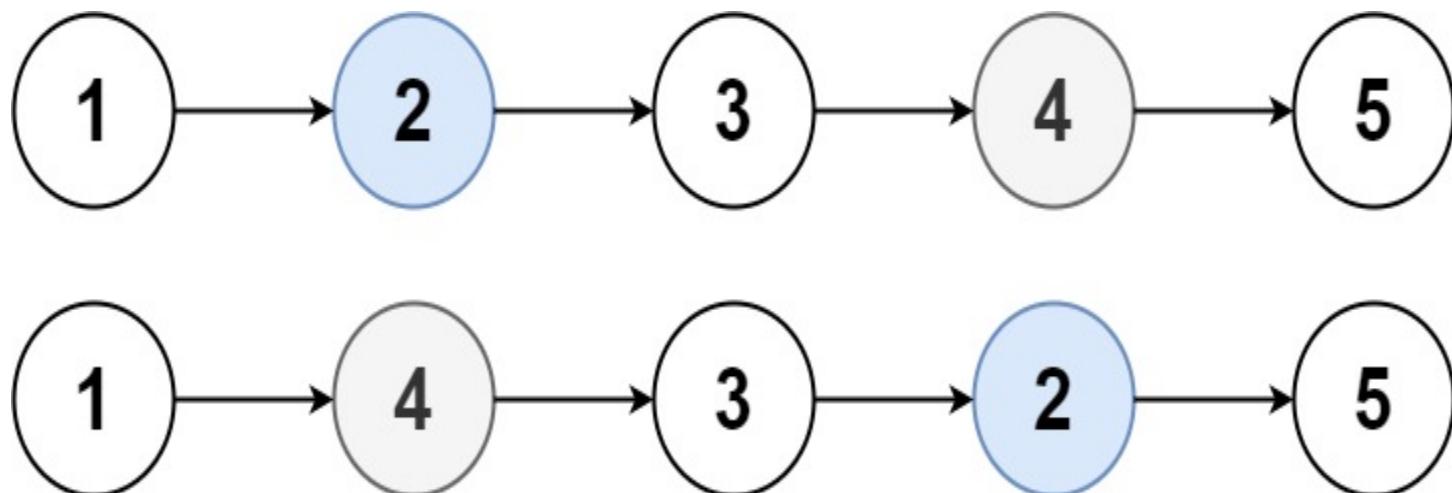
# 1721. Swapping Nodes in a Linked List ↗



You are given the head of a linked list, and an integer k.

Return *the head of the linked list after swapping the values of the k<sup>th</sup> node from the beginning and the k<sup>th</sup> node from the end (the list is 1-indexed)*.

## Example 1:



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

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

## Example 2:

**Input:** head = [7,9,6,6,7,8,3,0,9,5], k = 5

**Output:** [7,9,6,6,8,7,3,0,9,5]

## Example 3:

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

**Output:** [1]

## Example 4:

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

**Output:** [2,1]

## Example 5:

**Input:** head = [1,2,3], k = 2

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

**Constraints:**

- The number of nodes in the list is  $n$ .
- $1 \leq k \leq n \leq 10^5$
- $0 \leq \text{Node.val} \leq 100$

```
class Solution
{
 public ListNode swapNodes(ListNode head, int k)
 {
 if (head == null) return null;

 ListNode currNode = head; //slowPointer
 ListNode fp = head; //fastPointer

 while (fp != null && k > 1) //1 indexed LL
 {
 fp = fp.next;
 k--;
 }

 ListNode secondNode = fp;
 while (fp.next!= null)
 {
 fp =fp.next;
 currNode = currNode.next;
 }
 int tempVal = currNode.val;
 currNode.val = secondNode.val;
 secondNode.val = tempVal;

 return head;
 }
}
```

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

Return *the single element that appears only once*.

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

### Example 1:

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

**Output:** 2

### Example 2:

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

**Output:** 10

### Constraints:

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

```
class Solution {
 public int singleNonDuplicate(int[] nums) {

 int size = nums.length;
 if(size == 1)
 return nums[0];

 int i;
 for (i = 0; i < size-1; i=i+2)
 {
 if(nums[i] != nums[i+1])
 {
 return nums[i];
 }
 }
 if(i+1 == size)
 {
 return nums[i];
 }

 return 0;
 }
}
```

## 542. 01 Matrix ↗

Given an  $m \times n$  binary matrix  $\text{mat}$ , return *the distance of the nearest 0 for each cell.*

The distance between two adjacent cells is 1 .

**Example 1:**

|   |   |   |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 0 |

**Input:** mat = [[0,0,0],[0,1,0],[0,0,0]]

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

### Example 2:

|   |   |   |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 1 | 1 |

**Input:** mat = [[0,0,0],[0,1,0],[1,1,1]]

**Output:** [[0,0,0],[0,1,0],[1,2,1]]

### Constraints:

- $m == \text{mat.length}$
- $n == \text{mat}[i].length$
- $1 \leq m, n \leq 10^4$
- $1 \leq m * n \leq 10^4$
- $\text{mat}[i][j]$  is either 0 or 1.
- There is at least one 0 in mat.

```

//lot of new things
//Queue of integer array
//pahle sb zero ki position queue me daal do and 1 ko unvisited(-1) mark kar do
//jb bhi kise 1 ko visit karo use queue me daal do..that will act as virtual zero
class Solution {
 public int[][] updateMatrix(int[][] mat) {

 Queue<int[]> queue = new LinkedList<>();

 for (int i = 0; i < mat.length; i++) {
 for (int j = 0; j < mat[0].length; j++) {

 if (mat[i][j] == 0) { //catch all zero
 queue.add(new int[]{i,j});
 }
 else {
 mat[i][j] = -1;//making them unvisited for our bfs
 }
 }
 }

 //now we have all zero ka index ab inke bagal wale -1(unvisited) ko dist do
 int level = 0;
 while (!queue.isEmpty()) {
 level++;
 int size = queue.size();
 for (int i = 0; i < size; i++) {

 int[] currPosition = queue.poll(); //*****
 int currRow = currPosition[0];
 int currCol = currPosition[1];

 if (currRow +1 < mat.length) { //down

 if(mat[currRow +1][currCol]== -1){//-1 unvisited
 mat[currRow +1][currCol] = level;
 queue.add(new int[]{currRow +1, currCol}); //now this will act like virtual zero
 }
 }
 if (currRow -1 >= 0) { //up

 if(mat[currRow -1][currCol] == -1) {
 }
 }
 }
 }
}

```

```

 mat[currRow -1][currCol] = level;
 queue.add(new int[]{currRow -1, currCol});

 }

 if (currCol +1 < mat[0].length) { //right
 if(mat[currRow][currCol+1] == -1) {
 mat[currRow][currCol+1] = level;
 queue.add(new int[]{currRow, currCol+1});
 }
 }

 if (currCol -1 >= 0) { //left
 if(mat[currRow][currCol-1] == -1){
 mat[currRow][currCol-1] = level;
 queue.add(new int[]{currRow, currCol-1});
 }
 }
}

return mat;
}
}

```

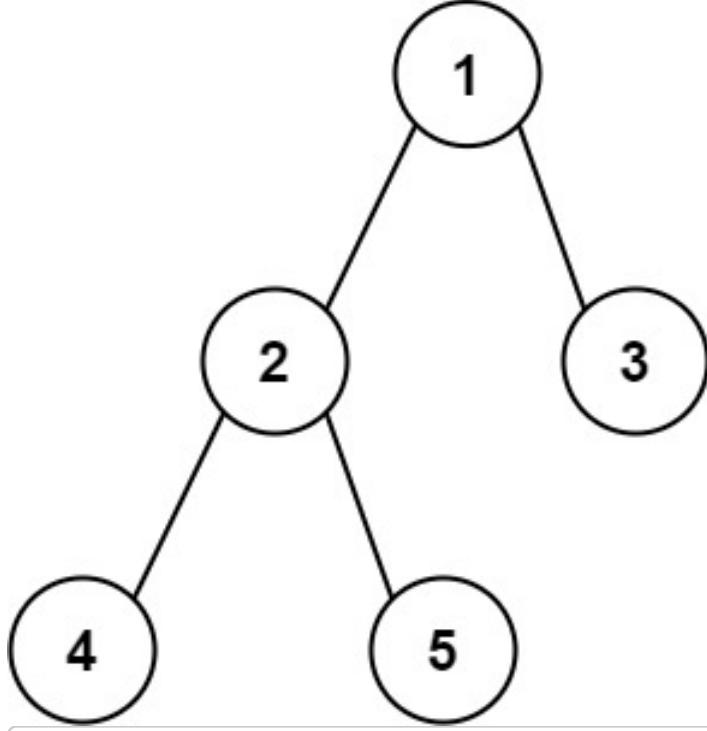
## 543. Diameter of Binary Tree ↗

Given the `root` of a binary tree, return *the length of the **diameter** of the tree*.

The **diameter** of a binary tree is the **length** of the longest path between any two nodes in a tree. This path may or may not pass through the `root`.

The **length** of a path between two nodes is represented by the number of edges between them.

**Example 1:**



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

**Output:** 3

**Explanation:** 3 is the length of the path [4,2,1,3] or [5,2,1,3].

### Example 2:

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

**Output:** 1

### Constraints:

- The number of nodes in the tree is in the range  $[1, 10^4]$ .
- $-100 \leq \text{Node.val} \leq 100$

```

/**
 * 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 {
 int max = 0;
 public int diameterOfBinaryTree(TreeNode root) {

 helper(root);
 return max;
 }

 private int helper(TreeNode root) {
 if (root == null) return 0; //null ki height 0 hogi

 int leftHeight = helper(root.left);
 int rightHeight = helper(root.right);

 int dia = (leftHeight + rightHeight);
 max = Math.max(max, dia);

 int height = 1+ Math.max(leftHeight, rightHeight); //+1 coz khud bh
 i to hai

 return height;
 }
}

```

```
/**
 * 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 {
 int max = 0;
 public int diameterOfBinaryTree(TreeNode root) {

 helper(root);
 return max;
 }

 private int helper(TreeNode root) {
 if (root == null) return -1;

 int leftHeight = helper(root.left);
 int rightHeight = helper(root.right);

 int dia = (leftHeight + rightHeight +2); // +2 for 2 stems/arms of
node
 max = Math.max(max, dia);

 int height = 1+ Math.max(leftHeight, rightHeight);

 return height;
 }
}
```

# 547. Number of Provinces ↗

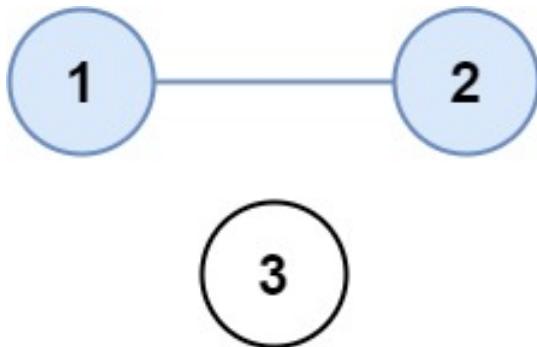
There are  $n$  cities. Some of them are connected, while some are not. If city  $a$  is connected directly with city  $b$ , and city  $b$  is connected directly with city  $c$ , then city  $a$  is connected indirectly with city  $c$ .

A **province** is a group of directly or indirectly connected cities and no other cities outside of the group.

You are given an  $n \times n$  matrix `isConnected` where `isConnected[i][j] = 1` if the  $i^{\text{th}}$  city and the  $j^{\text{th}}$  city are directly connected, and `isConnected[i][j] = 0` otherwise.

Return *the total number of provinces*.

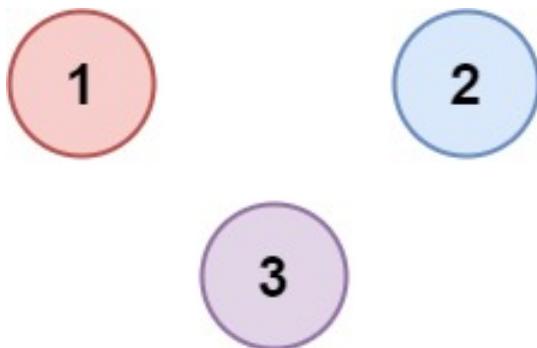
## Example 1:



**Input:** `isConnected = [[1,1,0],[1,1,0],[0,0,1]]`

**Output:** 2

## Example 2:



**Input:** `isConnected = [[1,0,0],[0,1,0],[0,0,1]]`

**Output:** 3

## Constraints:

- $1 \leq n \leq 200$

- $n == \text{isConnected.length}$
  - $n == \text{isConnected}[i].length$
  - $\text{isConnected}[i][j]$  is 1 or 0.
  - $\text{isConnected}[i][i] == 1$
  - $\text{isConnected}[i][j] == \text{isConnected}[j][i]$
- 

```
//DFS : isme 1 and 0 se starting me matlab nahi bs her row visit honi chaea
class Solution {
 public int findCircleNum(int[][] isConnected) {

 boolean visited[] = new boolean[isConnected.length];
 int count = 0;
 for (int i = 0; i < isConnected.length; i++) { //per row
 if (!visited[i]) { //row visited nahi h, ab iske sb friends dek
ho
 count++;
 dfs(isConnected, i, visited);
 }
 }
 return count;
 }

 private void dfs(int[][] isConnected, int i, boolean[] visited){

 for (int j = 0; j < isConnected.length; j++) { //per col
 if (isConnected[i][j] == 1 && !visited[j]) { //ye col visited na
hi h--> iski row explore karo
 visited[j] = true;
 dfs(isConnected, j, visited); //ab is col ki row explore ka
ro
 }
 }
 }
}
```

```

//BFS
class Solution {
 public int findCircleNum(int[][] isConnected) {
 boolean[] visited = new boolean[isConnected.length];
 int count = 0;
 for (int i = 0; i < isConnected.length; i++) {
 if (!visited[i]) { //row is not visited
 count++;
 bfs(isConnected, i, visited);
 }
 }
 return count;
 }
 //for each unvisited row
 private void bfs(int[][] isConnected, int i, boolean[] visited){
 Queue<Integer> queue = new LinkedList<>();
 for (int j = 0; j < isConnected.length; j++) { //ye sb uske friends
 if (isConnected[i][j] != 0 && !visited[i]) { //agr I ka koi frie
nd hai tb i ko queue me dalo
 queue.add(i);

 while (!queue.isEmpty()) {

 int r = queue.poll(); //i nikla first time
 visited[r] = true; //i visited

 for (int k = 0; k < isConnected.length; k++) //explore
this row as well
 {
 if (isConnected[r][k] == 1 && !visited[k]) {
 queue.add(k); //I ka jo jo friend hai, next us
row ko dekho
 }
 }
 }
 }
 }
 }
}

```

You are given a string `s` representing an attendance record for a student where each character signifies whether the student was absent, late, or present on that day. The record only contains the following three characters:

- '`A`' : Absent.
- '`L`' : Late.
- '`P`' : Present.

The student is eligible for an attendance award if they meet **both** of the following criteria:

- The student was absent (`'A'`) for **strictly** fewer than 2 days **total**.
- The student was **never** late (`'L'`) for 3 or more **consecutive** days.

Return `true` *if the student is eligible for an attendance award, or false otherwise.*

### Example 1:

**Input:** `s = "PPALLP"`

**Output:** `true`

**Explanation:** The student has fewer than 2 absences and was never late 3 or more consecutive days.

### Example 2:

**Input:** `s = "PPALLL"`

**Output:** `false`

**Explanation:** The student was late 3 consecutive days in the last 3 days, so it did not get an attendance award.

### Constraints:

- `1 <= s.length <= 1000`
- `s[i]` is either '`A`', '`L`', or '`P`'.

```

//use ++lateCount and ++absent
class Solution {
 public boolean checkRecord(String s) {

 int lateCount = 0;
 int absentCount = 0;

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

 if (s.charAt(i) == 'A') {
 lateCount = 0; //in case 2 baar late kar k absent ho jae to
 late cancel
 if (++absentCount >= 2) return false;
 }
 else if (s.charAt(i) == 'L'){
 if (++lateCount == 3) return false;
 }
 else if (s.charAt(i) == 'P') {
 lateCount = 0; //jb jb present hoga tb tb late cancel
 }
 }
 return true;
 }
}

```

## 560. Subarray Sum Equals K ↗

Given an array of integers `nums` and an integer `k`, return *the total number of continuous subarrays whose sum equals to `k`*.

### Example 1:

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

### Example 2:

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

**Output:** 2

### Constraints:

- $1 \leq \text{nums.length} \leq 2 * 10^4$
- $-1000 \leq \text{nums}[i] \leq 1000$
- $-10^7 \leq k \leq 10^7$

```
class Solution {
 public int subarraySum(int[] nums, int k) {

 HashMap<Integer, Integer> freqMap = new HashMap<>();

 int totalSum = 0, count = 0;;
 freqMap.put(0,1); //we got total sum = 0 once
 for (int i = 0; i < nums.length; i++)
 {
 totalSum += nums[i];
 int remSum = totalSum - k;
 if (freqMap.containsKey(remSum)) { //agr remaining hai tb he co
 count += freqMap.get(remSum);
 }
 if (freqMap.containsKey(totalSum)) { //ab tot sum bhi daal do
 freqMap.put(totalSum, freqMap.get(totalSum)+1);
 }
 else {
 freqMap.put(totalSum, 1);
 }
 }
 return count;
 }
}
```

```

//Approach : Prefix Sum
//hum dekege num[i] ka sum karte jaege
//her sum pr check karege ki nums[i]-s agr present hoga tb matlab hm ek sub
array mil gaya...or hum count ++ karege...
//suppose humara sum 10 ho gyaa... or S = 6 hai to hum check karge abhi tk
kya 4 aaya tha...agr han to matlab ek subarray aa gaaya jiska sum 6 h.
class Solution
{
 public int subarraySum(int[] nums, int k)
 {
 int sum = 0, count = 0;
 HashMap<Integer, Integer> map = new HashMap<>();
 map.put(0,1);
 for (int i = 0; i < nums.length; i++)
 {
 sum += nums[i];
 int rSum = sum - k; //remaining sum
 if (map.containsKey(rSum)) //agar map me 4 hai
 {
 count += map.get(rSum); //count me ab tk ki fre dalege
 }
 if (map.containsKey(sum))
 {
 map.put(sum, map.get(sum)+1);
 }
 else
 {
 map.put(sum, 1);
 }
 }
 return count;
 }
}

```

## 562. Longest Line of Consecutive One in Matrix

Given an  $m \times n$  binary matrix `mat`, return *the length of the longest line of consecutive one in the matrix*.

The line could be horizontal, vertical, diagonal, or anti-diagonal.

### Example 1:

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 |
| 0 | 0 | 0 | 1 |

Input: mat = [[0,1,1,0],[0,1,1,0],[0,0,0,1]]

Output: 3

### Example 2:

|   |   |   |   |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 |
| 0 | 0 | 0 | 1 |

Input: mat = [[1,1,1,1],[0,1,1,0],[0,0,0,1]]

Output: 4

### Constraints:

- $m == \text{mat.length}$
- $n == \text{mat}[i].length$
- $1 \leq m, n \leq 10^4$
- $1 \leq m * n \leq 10^4$

- `mat[i][j]` is either `0` or `1`.
-

```

class Solution {
 public int longestLine(int[][] mat) {

 Pair[][] dp = new Pair[mat.length][mat[0].length];

 for (int i = 0; i < mat.length; i++) {
 for (int j = 0; j < mat[0].length; j++) {

 Pair newPair = new Pair();
 if (j-1 >= 0 && mat[i][j] != 0) {
 newPair.left = mat[i][j] + (dp[i][j-1]).left;
 }
 else {
 newPair.left = mat[i][j];
 }
 if (i-1 >= 0 && mat[i][j] != 0) {
 newPair.up = mat[i][j] + (dp[i-1][j]).up;
 }
 else {
 newPair.up = mat[i][j];
 }
 if(i-1 >= 0 && j+1 < mat[0].length && mat[i][j] != 0) {//right up dia
 newPair.rightDia = mat[i][j] + (dp[i-1][j+1]).rightDia;
 }
 else {
 newPair.rightDia = mat[i][j];
 }
 if (i-1 >= 0 && j-1 >= 0 && mat[i][j] != 0) {//left up dia
 newPair.leftDia = mat[i][j] + (dp[i-1][j-1]).leftDia;
 }
 else {
 newPair.leftDia = mat[i][j];
 }

 dp[i][j] = newPair;
 }
 }

 int maxLeft = Integer.MIN_VALUE;
 int maxup = Integer.MIN_VALUE;
 int maxRightDia = Integer.MIN_VALUE;
 int maxLeftDia = Integer.MIN_VALUE;

 for (int i = 0; i < mat.length; i++) {
 for (int j = 0; j < mat[0].length; j++) {

```

```

 maxLeft = Math.max(maxLeft, dp[i][j].left);
 maxup = Math.max(maxup, dp[i][j].up);
 maxRightDia = Math.max(maxRightDia, dp[i][j].rightDia);
 maxLeftDia = Math.max(maxLeftDia, dp[i][j].leftDia);
 }
}

int totalMax = Math.max(maxLeft, Math.max(maxup, Math.max(maxRightDia, maxLeftDia)));

return totalMax;

}
private class Pair {
 int left;
 int up;
 int rightDia;
 int leftDia;
}
}

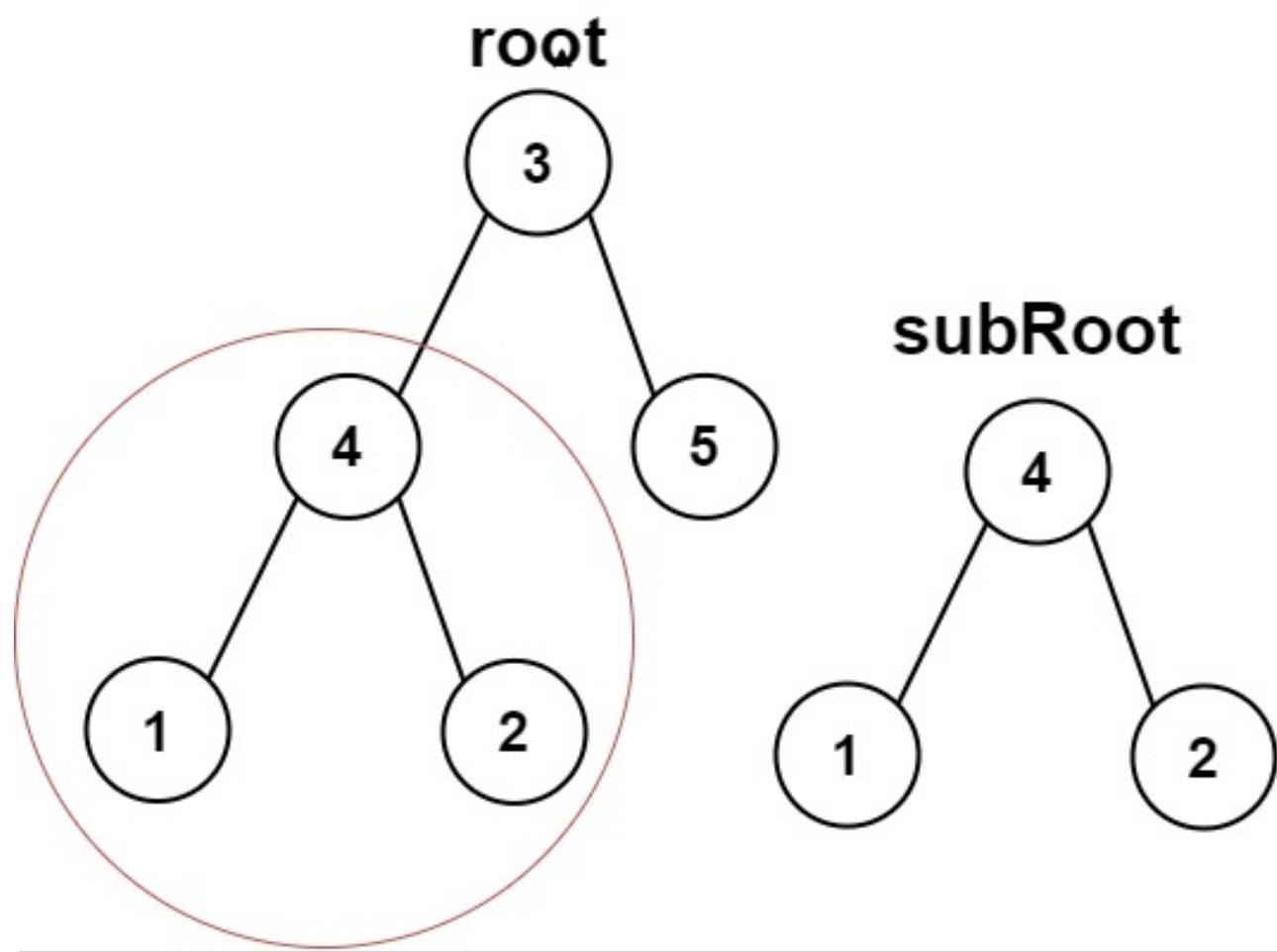
```

## 572. Subtree of Another Tree ↗

Given the roots of two binary trees `root` and `subRoot`, return `true` if there is a subtree of `root` with the same structure and node values of `subRoot` and `false` otherwise.

A subtree of a binary tree `tree` is a tree that consists of a node in `tree` and all of this node's descendants. The tree `tree` could also be considered as a subtree of itself.

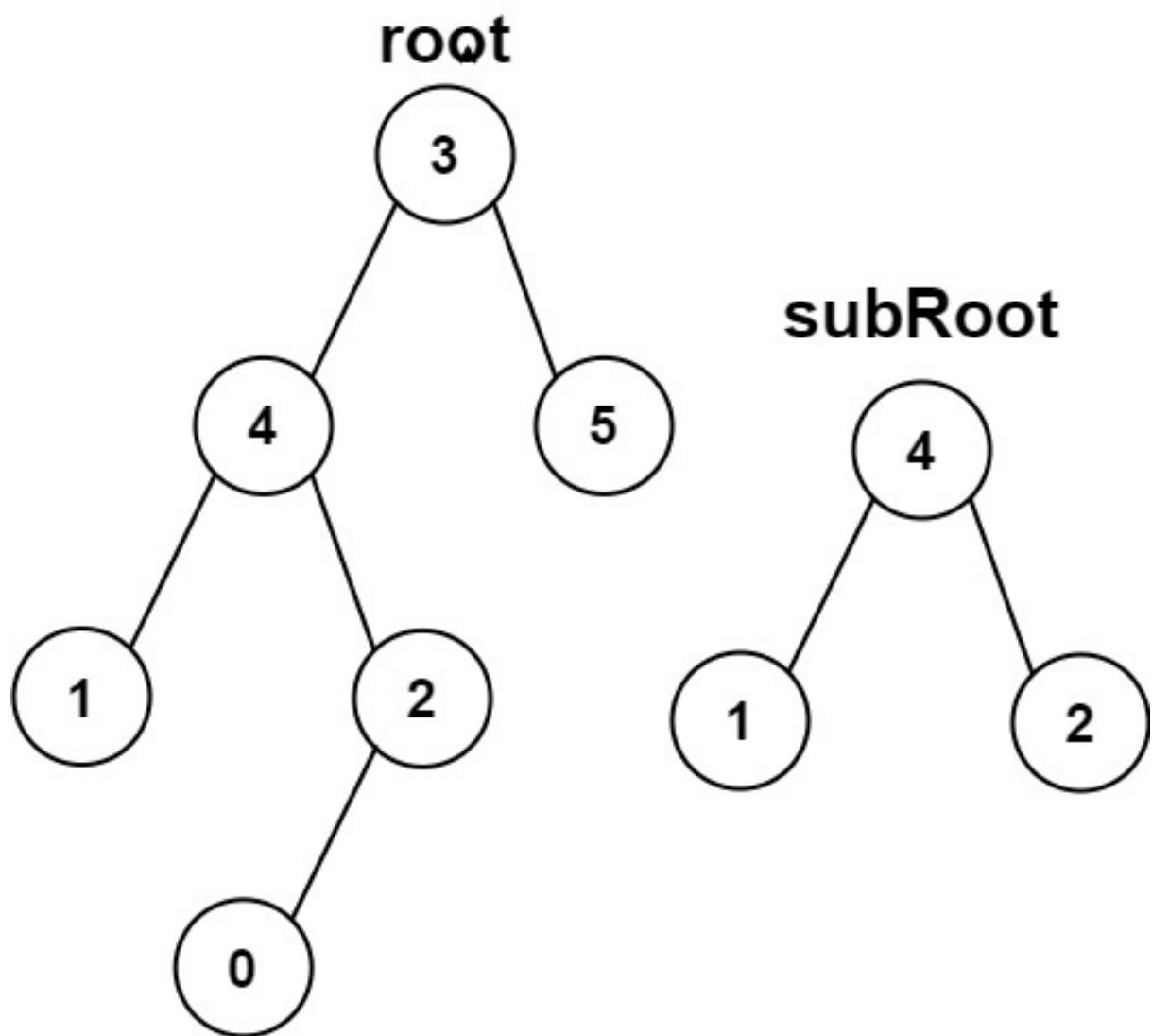
**Example 1:**



**Input:** root = [3,4,5,1,2], subRoot = [4,1,2]

**Output:** true

**Example 2:**



**Input:** root = [3,4,5,1,2,null,null,null,null,0], subRoot = [4,1,2]

**Output:** false

#### Constraints:

- The number of nodes in the root tree is in the range [1, 2000].
- The number of nodes in the subRoot tree is in the range [1, 1000].
- $-10^4 \leq \text{root.val} \leq 10^4$
- $-10^4 \leq \text{subRoot.val} \leq 10^4$

```

/**
 * 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;
 * }
 * }
 */

//Runtime : O(n*m) Space: min of m & n
class Solution {
 public boolean isSubtree(TreeNode s, TreeNode t)
 {
 if (s == null && t == null) return true;
 if (s != null && t == null) return false;
 if (s == null && t != null) return false;
 //agar value same hai to isSubTreeHelp me bejo ki aage bhi same h y
 a ni
 if (s.val == t.val)
 {
 boolean ans = isSubTreeHelp(s,t);
 if (ans) return true;
 }
 return (isSubtree(s.right, t) || isSubtree(s.left, t));
 }

 private boolean isSubTreeHelp(TreeNode s, TreeNode t)
 {
 if (s == null && t == null) return true;
 if (s != null && t == null) return false;
 if (s == null && t != null) return false;
 //value same h to left right compare karo
 if (s.val == t.val)
 {
 boolean left = isSubTreeHelp(s.left, t.left);
 boolean right = isSubTreeHelp(s.right, t.right);
 if (left == true && right == true) return true;
 else return false;
 }
 }
}

```

```
 }

 return false;

}
}
```

## 575. Distribute Candies ↗

Alice has  $n$  candies, where the  $i^{\text{th}}$  candy is of type `candyType[i]`. Alice noticed that she started to gain weight, so she visited a doctor.

The doctor advised Alice to only eat  $n / 2$  of the candies she has ( $n$  is always even). Alice likes her candies very much, and she wants to eat the maximum number of different types of candies while still following the doctor's advice.

Given the integer array `candyType` of length  $n$ , return *the **maximum** number of different types of candies she can eat if she only eats  $n / 2$  of them*.

### Example 1:

**Input:** `candyType = [1,1,2,2,3,3]`

**Output:** 3

**Explanation:** Alice can only eat  $6 / 2 = 3$  candies. Since there are only 3 typ

### Example 2:

**Input:** `candyType = [1,1,2,3]`

**Output:** 2

**Explanation:** Alice can only eat  $4 / 2 = 2$  candies. Whether she eats types [1,

### Example 3:

**Input:** `candyType = [6,6,6,6]`

**Output:** 1

**Explanation:** Alice can only eat  $4 / 2 = 2$  candies. Even though she can eat 2

## Constraints:

- $n == \text{candyType.length}$
- $2 \leq n \leq 10^4$
- $n$  is even.
- $-10^5 \leq \text{candyType}[i] \leq 10^5$

```
class Solution {
 public int distributeCandies(int[] candyType)
 {
 if (candyType.length == 0) return 0;
 //1,1,2,2,3,3 1,2,3
 //1,1,2,3
 //6,6,6,6 6
 int l = candyType.length;
 int maxCandy = l>>1;

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

 for (int i = 0; i < l; i++)
 {
 set.add(candyType[i]);
 }

 if (set.size() > maxCandy) return maxCandy;

 return set.size();
 }
}
```

## 616. Add Bold Tag in String

You are given a string `s` and an array of strings `words`. You should add a closed pair of bold tag `<b>` and `</b>` to wrap the substrings in `s` that exist in `words`. If two such substrings overlap, you should wrap them together with only one pair of closed bold-tag. If two substrings wrapped by bold tags are consecutive, you should combine them.

Return `s` after adding the bold tags.

### Example 1:

**Input:** s = "abcxyz123", words = ["abc", "123"]

**Output:** "<b>abc</b>xyz<b>123</b>"

### Example 2:

**Input:** s = "aaabbcc", words = ["aaa", "aab", "bc"]

**Output:** "<b>aaabbc</b>c"

### Constraints:

- $1 \leq s.length \leq 1000$
- $0 \leq \text{words.length} \leq 100$
- $1 \leq \text{words}[i].length \leq 1000$
- s and words[i] consist of English letters and digits.
- All the values of words are **unique**.

**Note:** This question is the same as 758: <https://leetcode.com/problems/bold-words-in-string/> (<https://leetcode.com/problems/bold-words-in-string/>)

```

//int indexOf(String substring, int fromIndex)
class Solution {
 public String addBoldTag(String s, String[] words) {

 int[] flagArr = new int[s.length()]; //jo jo number tag k ander hog
a vo 1 hoga
 for (String word : words) {

 int index = -1;

 while (true) {

 //O(n*m)
 index = s.indexOf(word, index+1); //qki ek word kae baar ho
skta hai to un sb ko include karna h
 if (index != -1) {

 int end = index+word.length();

 for (int ii = index; ii < end; ii++) {
 flagArr[ii] = 1;
 }
 }
 else {
 break; //index -ve ho gya, mataln ab ye word or nahi h
 }
 }

 }
 StringBuffer ans = new StringBuffer();
 int i = 0;
 for (; i < s.length();i++) { //O(n)

 if (i==0 && flagArr[i] == 1){ //pahle se he start karna h
 ans.append("");
 }

 if (i > 0 && flagArr[i] == 1 && flagArr[i-1] != 1) { //!=1 se c
onfirm ho raha ki strt karna h
 ans.append("");
 }
 if (i > 0 && flagArr[i] == 0 && flagArr[i-1] == 1) { //==1 se c
onfirm ho raha end karna h...1 tk ka bold me tha
 ans.append("");
 }
 }
 }
}

```

```

 ans.append(s.charAt(i)); //character to her baar he add hoga
 }
 if (flagArr[i-1] == 1) { //in case pure he string bold karna ho agr
 ans.append("");
 }
 return ans.toString();
}
}

```

## 617. Merge Two Binary Trees ↗

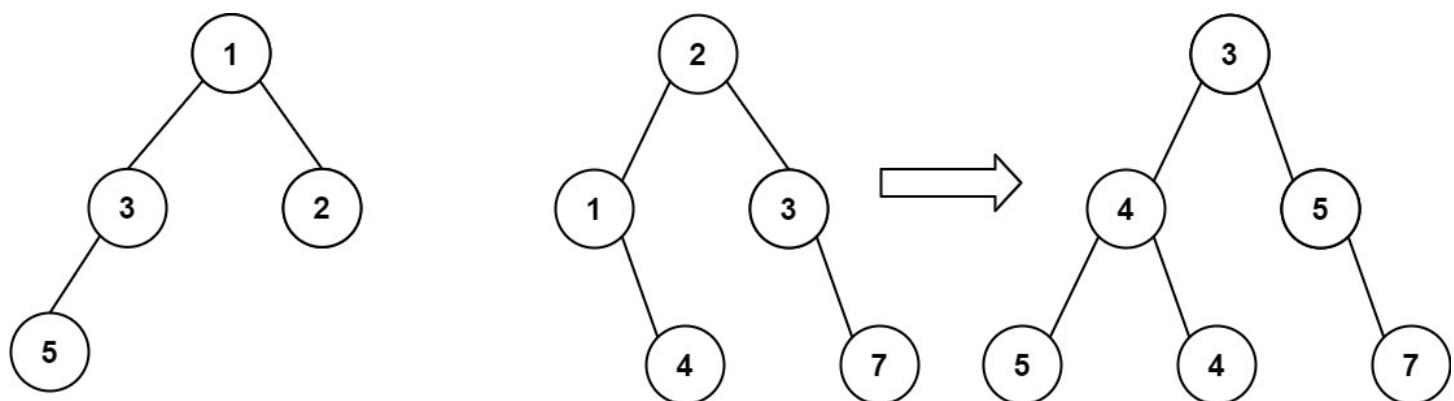
You are given two binary trees `root1` and `root2`.

Imagine that when you put one of them to cover the other, some nodes of the two trees are overlapped while the others are not. You need to merge the two trees into a new binary tree. The merge rule is that if two nodes overlap, then sum node values up as the new value of the merged node. Otherwise, the NOT null node will be used as the node of the new tree.

Return *the merged tree*.

**Note:** The merging process must start from the root nodes of both trees.

### Example 1:



**Input:** `root1 = [1,3,2,5]`, `root2 = [2,1,3,null,4,null,7]`

**Output:** `[3,4,5,5,4,null,7]`

### Example 2:

**Input:** `root1 = [1]`, `root2 = [1,2]`

**Output:** `[2,2]`

## Constraints:

- The number of nodes in both trees is in the range [0, 2000].
- $-10^4 \leq \text{Node.val} \leq 10^4$

```
class Solution {
 public TreeNode mergeTrees(TreeNode t1, TreeNode t2)
 {

 if (t1 == null && t2 == null) return null;

 TreeNode newNode = new TreeNode();
 if (t1 == null && t2 != null)
 {
 return t2;
 }
 if (t1 != null && t2 == null)
 {
 return t1;
 }

 if (t1 != null && t2 != null)
 {
 newNode.val = t1.val + t2.val;
 newNode.left = mergeTrees(t1.left, t2.left);
 newNode.right = mergeTrees(t1.right, t2.right);
 }
 return newNode;
 }
}
```

## 623. Add One Row to Tree ↗

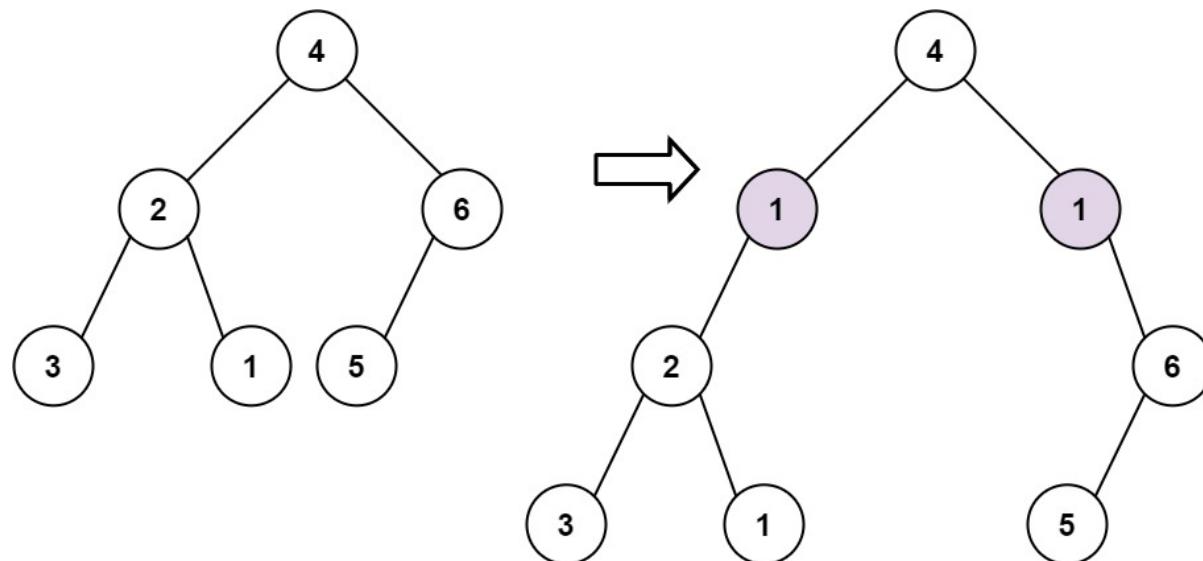
Given the `root` of a binary tree and two integers `val` and `depth`, add a row of nodes with value `val` at the given depth `depth`.

Note that the `root` node is at depth 1.

The adding rule is:

- Given the integer `depth`, for each not null tree node `cur` at the depth `depth - 1`, create two tree nodes with value `val` as `cur`'s left subtree root and right subtree root.
- `cur`'s original left subtree should be the left subtree of the new left subtree root.
- `cur`'s original right subtree should be the right subtree of the new right subtree root.
- If `depth == 1` that means there is no depth `depth - 1` at all, then create a tree node with value `val` as the new root of the whole original tree, and the original tree is the new root's left subtree.

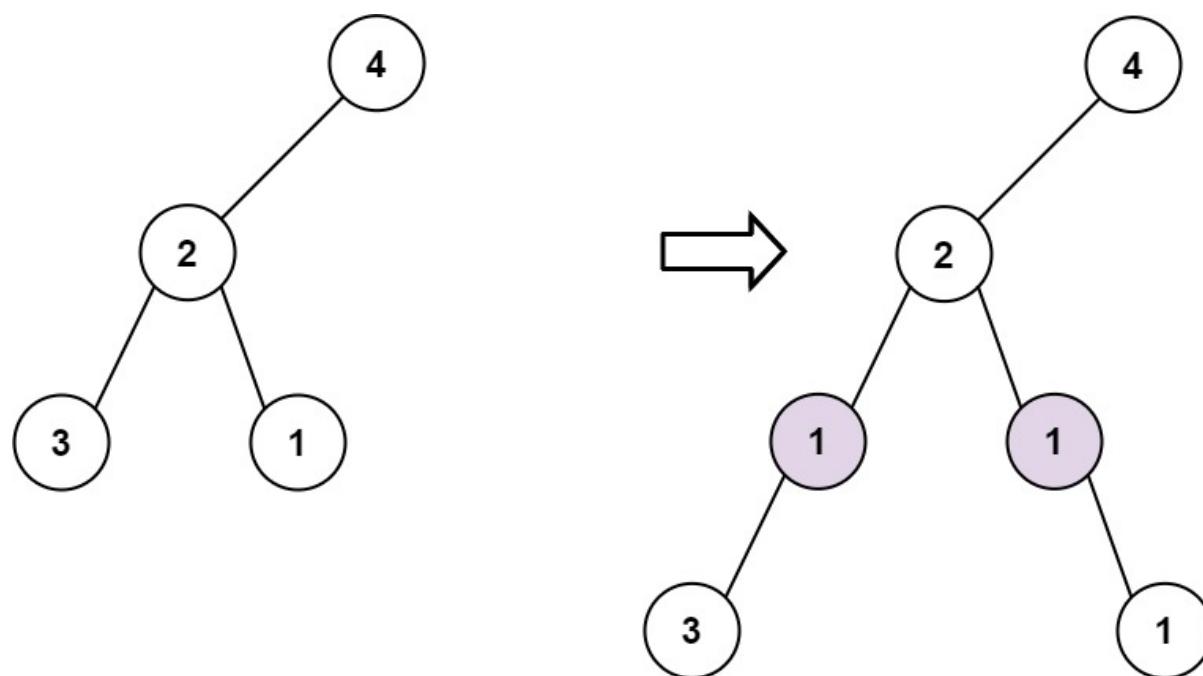
### Example 1:



**Input:** `root = [4,2,6,3,1,5]`, `val = 1`, `depth = 2`

**Output:** `[4,1,1,2,null,null,6,3,1,5]`

### Example 2:



**Input:** `root = [4,2,null,3,1]`, `val = 1`, `depth = 3`

**Output:** `[4,2,null,1,1,3,null,null,1]`

### **Constraints:**

- The number of nodes in the tree is in the range  $[1, 10^4]$  .
  - The depth of the tree is in the range  $[1, 10^4]$  .
  - $-100 \leq \text{Node.val} \leq 100$
  - $-10^5 \leq \text{val} \leq 10^5$
  - $1 \leq \text{depth} \leq \text{the depth of tree} + 1$
-

```
/**
 * 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 TreeNode addOneRow(TreeNode root, int v, int d)
 {
 if (root == null) return null;

 if (d == 1)
 {
 TreeNode node = new TreeNode(v);
 node.left = root;
 return node;
 }
 else
 {
 addOneRowhelp (root, v, d);
 }
 return root;
 }
 public void addOneRowhelp(TreeNode root, int v, int d)
 {

 if (root == null) return;
 if (d-1 == 1)
 {
 if (root.left != null)
 {
 TreeNode templeft = root.left;
 TreeNode node = new TreeNode(v);
 root.left = node;
 node.left = templeft;
 }
 else root.left = new TreeNode (v);
 }
 }
}
```

```
if (root.right != null)
{
 TreeNode tempRight = root.right;
 TreeNode node = new TreeNode(v);
 root.right = node;
 node.right = tempRight;
}
else root.right = new TreeNode (v);
return;
}

else if (d-1 < 1) return;

addOneRowhelp(root.left, v, d -1);
// d++;
addOneRowhelp(root.right, v, d -1);

return;

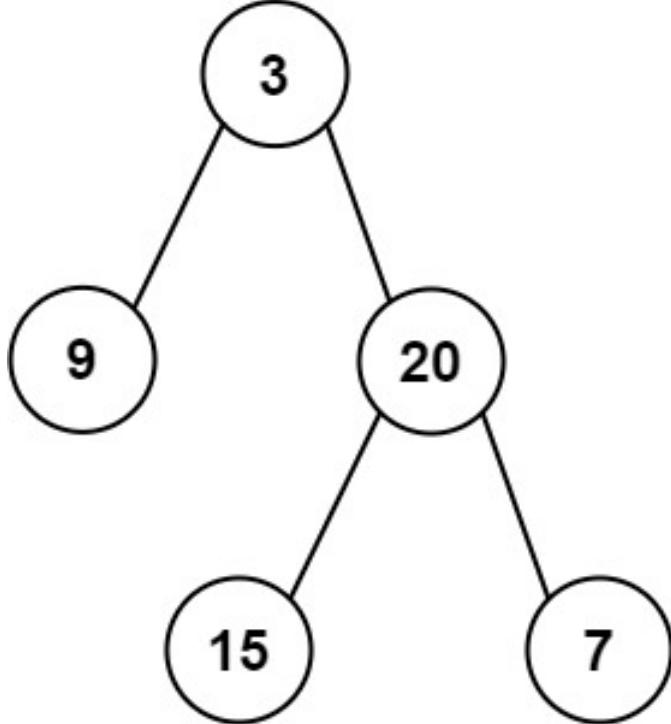
}

}
```

## 637. Average of Levels in Binary Tree ↗

Given the `root` of a binary tree, return *the average value of the nodes on each level in the form of an array*. Answers within  $10^{-5}$  of the actual answer will be accepted.

**Example 1:**

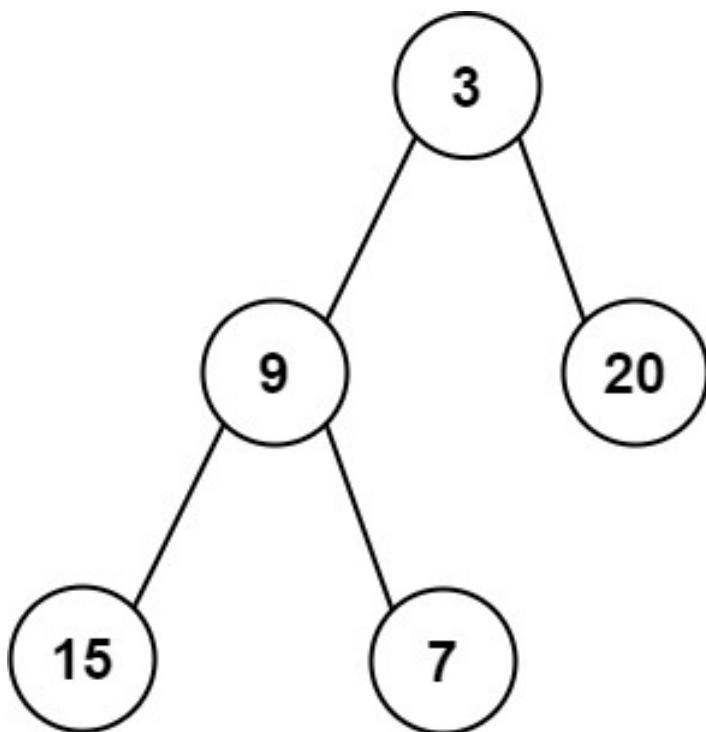


**Input:** root = [3,9,20,null,null,15,7]

**Output:** [3.00000,14.50000,11.00000]

**Explanation:** The average value of nodes on level 0 is 3, on level 1 is 14.5, Hence return [3, 14.5, 11].

### Example 2:



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

**Output:** [3.00000,14.50000,11.00000]

### Constraints:

- The number of nodes in the tree is in the range  $[1, 10^4]$  .
- $-2^{31} \leq \text{Node.val} \leq 2^{31} - 1$

```

/**
 * 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 List<Double> averageOfLevels(TreeNode root) {

 if (root == null) return null;
 List<Double> list = new ArrayList<Double>();
 Queue<TreeNode> q = new LinkedList<>();

 q.add(root);
 while (q.size() > 0)
 {
 double size = q.size();
 double sum = 0;
 for (int i = 0; i < size; i++)
 {
 TreeNode n = q.remove();
 sum += n.val;

 if (n.left != null) q.add(n.left);
 if (n.right != null) q.add(n.right);
 }
 list.add(sum/size);
 }
 return list;
 }
}

```

# 645. Set Mismatch



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

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

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

## Example 1:

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

**Output:** `[2,3]`

## Example 2:

**Input:** `nums = [1,1]`

**Output:** `[1,2]`

## Constraints:

- $2 \leq \text{nums.length} \leq 10^4$
- $1 \leq \text{nums}[i] \leq 10^4$

used extra space:

```

class Solution {
 public int[] findErrorNums(int[] nums)
 {
 int res[] = new int[2];
 int help[] = new int[nums.length+1];
 for (int i = 0; i < nums.length; i++)
 {
 if (help[nums[i]] == 1)
 {
 res[0] = nums[i]; //means that number had already appeared
so repeated
 }
 else
 {
 help[nums[i]] = 1;
 }
 }
 for (int i = 1; i < help.length; i++)
 {
 if (help[i] == 0) //means this number never came
 {
 res[1] = i;
 }
 }
 return res;
 }
}

```

## 647. Palindromic Substrings ↗

Given a string  $s$ , return *the number of palindromic substrings in it.*

A string is a **palindrome** when it reads the same backward as forward.

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

### Example 1:

**Input:**  $s = \text{"abc"}$

**Output:** 3

**Explanation:** Three palindromic strings: "a", "b", "c".

## **Example 2:**

**Input:** s = "aaa"

**Output:** 6

**Explanation:** Six palindromic strings: "a", "a", "a", "aa", "aa", "aaa".

### **Constraints:**

- $1 \leq s.length \leq 1000$
  - s consists of lowercase English letters.
-

```
//Runtime: 12 ms, faster than 27.95%
class Solution {

 public int countSubstrings(String s)
 {
 if (s == null || s == "") return 0;

 int l = s.length();
 if (l == 1) return 1;
 else if (l == 2 && s.charAt(0) == s.charAt(1)) return 3;//single, single, double

 int ans = 0;
 boolean dp [][] = new boolean[l][l];

 for (int gap = 0; gap < l; gap++)
 {
 for (int i = 0; i+gap < l; i++)
 {
 if (gap == 0)
 {
 dp[i][i+gap] = true; //single character
 ans++;
 }
 else if (gap == 1)
 {
 if (s.charAt(i) == s.charAt(i+gap))/2 characters
 {
 dp[i][i+gap] = true;
 ans++;
 }
 else
 {
 dp[i][i+gap] = false;
 }
 }
 else if (gap > 1)//first & last character check kar rai..be
ch ka atrix se mil ra
 {
 if (s.charAt(i) == s.charAt(i+gap) && dp[i+1][i+gap-1])
 {
 dp[i][i+gap] = true;
 ans++;
 }
 else
 {

```

```

 dp[i][i+gap] = false;
 }
}
}
return ans;
}
}

```

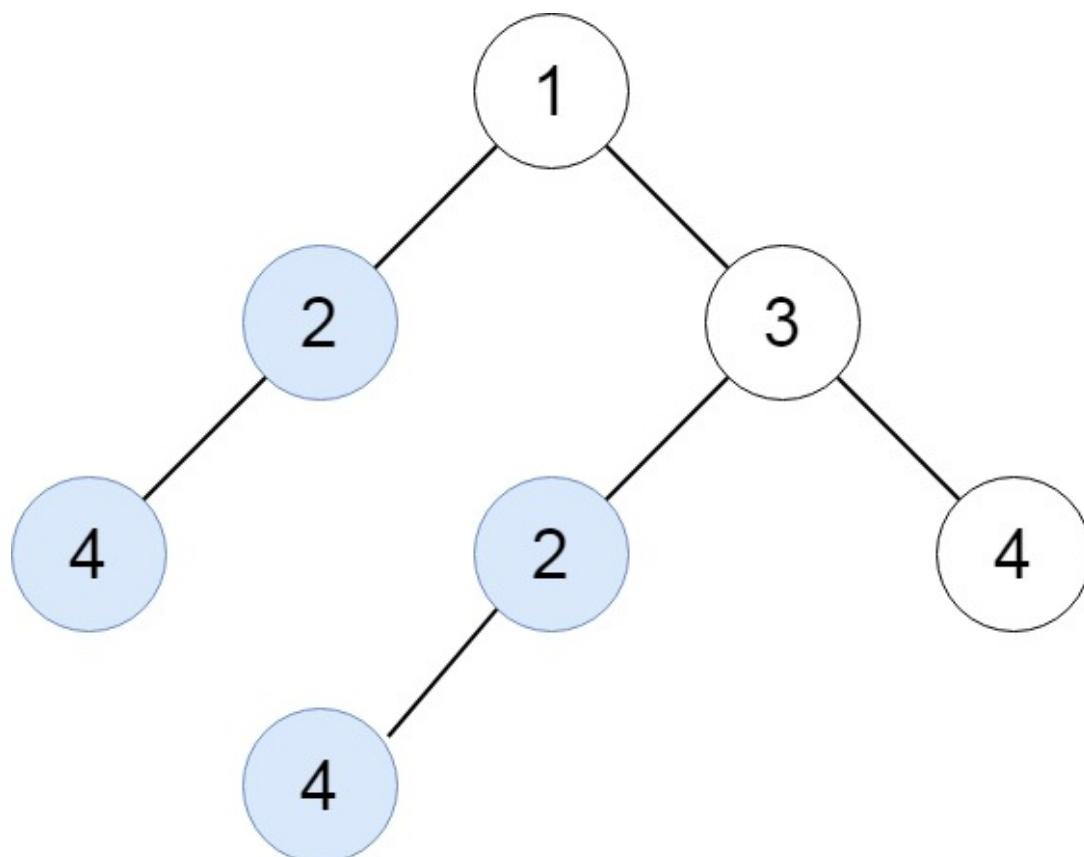
## 652. Find Duplicate Subtrees ↗

Given the `root` of a binary tree, return all **duplicate subtrees**.

For each kind of duplicate subtrees, you only need to return the root node of any **one** of them.

Two trees are **duplicate** if they have the **same structure** with the **same node values**.

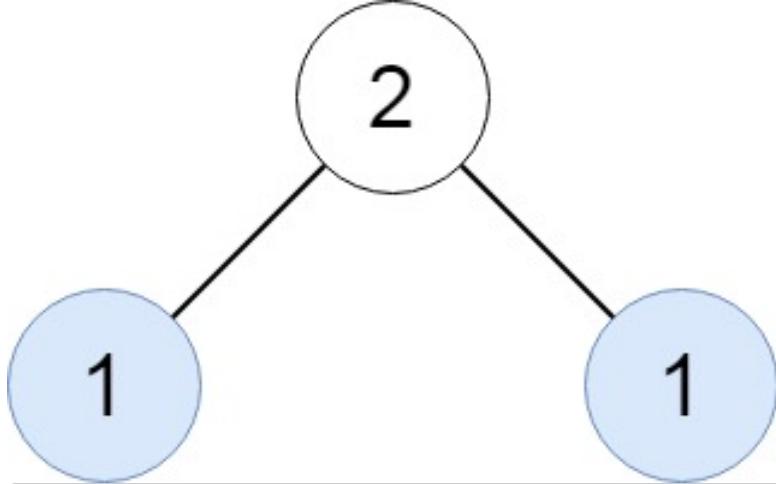
**Example 1:**



**Input:** `root = [1,2,3,4,null,2,4,null,null,4]`

**Output:** `[[2,4],[4]]`

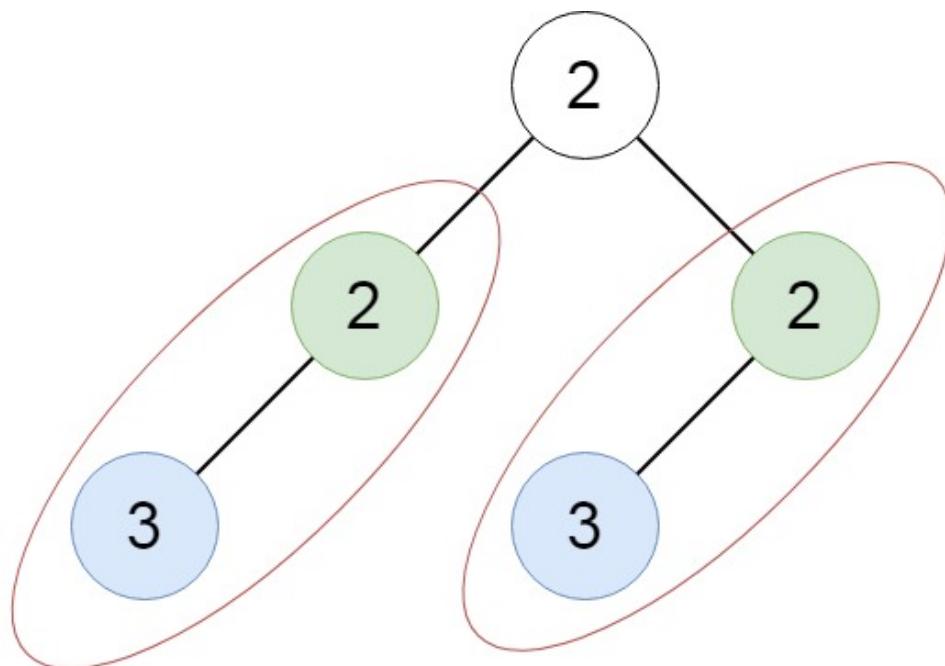
**Example 2:**



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

**Output:** [[1]]

### Example 3:



**Input:** root = [2,2,2,3,null,3,null]

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

### Constraints:

- The number of the nodes in the tree will be in the range [1, 10^4]
- $-200 \leq \text{Node.val} \leq 200$

```

/**
 * 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;
 * }
 * }
 */
//Dont forget to add (and) else it will not able to distinguish
//dont use set coz it will insert for every duplicate..if string is 3 time
it will insert two time
class Solution {
 public List<TreeNode> findDuplicateSubtrees(TreeNode root) {

 HashMap<String, Integer> map = new HashMap<>();

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

 helper(root, map, ans);
 return ans;
 }

 private String helper(TreeNode root, HashMap<String, Integer> map, List
<TreeNode> ans) {

 if (root == null) return "";

 String left = helper(root.left, map, ans);
 String right = helper(root.right, map, ans);

 String rootVal = "("+ left+ root.val+ right + ")";
 //System.out.println(map);

 if (map.getOrDefault(rootVal,0) == 1) ans.add(root); //second time
so duplicate

 map.put(rootVal, map.getOrDefault(rootVal, 0)+1);
 }
}

```

```
 return rootVal;
}
}
```

## 657. Robot Return to Origin ↗



There is a robot starting at the position  $(0, 0)$ , the origin, on a 2D plane. Given a sequence of its moves, judge if this robot **ends up at**  $(0, 0)$  after it completes its moves.

You are given a string `moves` that represents the move sequence of the robot where `moves[i]` represents its  $i^{\text{th}}$  move. Valid moves are 'R' (right), 'L' (left), 'U' (up), and 'D' (down).

Return `true` if the robot returns to the origin after it finishes all of its moves, or `false` otherwise.

**Note:** The way that the robot is "facing" is irrelevant. 'R' will always make the robot move to the right once, 'L' will always make it move left, etc. Also, assume that the magnitude of the robot's movement is the same for each move.

### Example 1:

**Input:** `moves = "UD"`

**Output:** `true`

**Explanation:** The robot moves up once, and then down once. All moves have the

### Example 2:

**Input:** `moves = "LL"`

**Output:** `false`

**Explanation:** The robot moves left twice. It ends up two "moves" to the left of

### **Example 3:**

**Input:** moves = "RRDD"

**Output:** false

### **Example 4:**

**Input:** moves = "LDRLRLUULR"

**Output:** false

### **Constraints:**

- $1 \leq \text{moves.length} \leq 2 * 10^4$
  - moves only contains the characters 'U' , 'D' , 'L' and 'R' .
-

```

class Solution {
 public boolean judgeCircle(String moves) {

 int lr= 0, ud = 0;

 char[] move = moves.toCharArray();

 for (int i = 0; i < move.length; i++)
 {
 if (move[i] == 'U')
 {
 ud++;
 }
 else if (move[i] == 'D')
 {
 ud--;
 }
 else if (move[i] == 'L')
 {
 lr++;
 }
 else if (move[i] == 'R')
 {
 lr--;
 }
 }

 if (lr == 0 && ud == 0)
 {
 return true;
 }
 return false;
 }
}

```

## 665. Non-decreasing Array ↗



Given an array `nums` with `n` integers, your task is to check if it could become non-decreasing by modifying **at most one element**.

We define an array is non-decreasing if `nums[i] <= nums[i + 1]` holds for every `i` (**0-based**) such that (`0 <= i <= n - 2`).

### Example 1:

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

**Output:** `true`

**Explanation:** You could modify the first 4 to 1 to get a non-decreasing array.

### Example 2:

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

**Output:** `false`

**Explanation:** You can't get a non-decreasing array by modify at most one element.

### Constraints:

- `n == nums.length`
- `1 <= n <= 104`
- `-105 <= nums[i] <= 105`

```

class Solution {
 public boolean checkPossibility(int[] nums)
 {
 boolean corrected = false;

 for (int i = 1; i < nums.length; i++)
 {
 if (i == 1 && nums[i] < nums[i-1])//boundary condition
 {
 nums[i-1] = nums[i];
 corrected = true;
 }
 if (nums[i] < nums[i-1]) //decrease mila
 {
 if (corrected) return false; //qki ek baar correct kar chuk
e hai

 if (nums[i] < nums[i-2]) //change to just prev
 {
 nums[i] = nums[i-1];
 corrected = true;
 }
 else if (nums[i] > nums[i-2]) //change prev
 {
 nums[i-1] = nums[i];
 corrected = true;
 }
 }
 }
 return true;
 }
}

```

## 680. Valid Palindrome II ↗

Given a string  $s$ , return `true` if the  $s$  can be palindrome after deleting **at most one** character from it.

**Example 1:**

**Input:** s = "aba"

**Output:** true

### Example 2:

**Input:** s = "abca"

**Output:** true

**Explanation:** You could delete the character 'c'.

### Example 3:

**Input:** s = "abc"

**Output:** false

### Constraints:

- $1 \leq s.length \leq 10^5$
  - s consists of lowercase English letters.
-

```

class Solution {
 public boolean validPalindrome(String s) {

 int i = 0, j = s.length() - 1;

 while (i < j) {

 if (s.charAt(i) != s.charAt(j)) {
 return (isPallindrome(i+1, j, s) || isPallindrome(i, j-1,
s)); //send to this after removing one character either from i or j
 }
 i++; j--;
 }
 return true;
 }

 private boolean isPallindrome(int i, int j, String s) {

 while (i < j) {
 if (s.charAt(i) != s.charAt(j)) return false; //second time unmatched character

 i++; j--;
 }

 return true;
 }
}

```

## 690. Employee Importance ↗

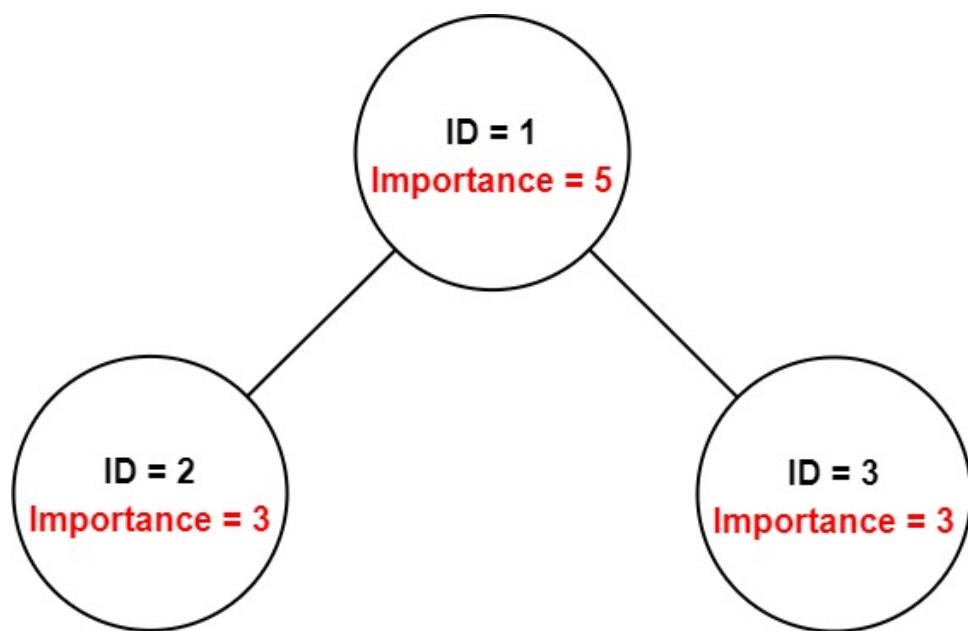
You have a data structure of employee information, which includes the employee's unique ID, their importance value, and their direct subordinates' IDs.

You are given an array of employees `employees` where:

- `employees[i].id` is the ID of the  $i^{\text{th}}$  employee.
- `employees[i].importance` is the importance value of the  $i^{\text{th}}$  employee.
- `employees[i].subordinates` is a list of the IDs of the direct subordinates of the  $i^{\text{th}}$  employee.

Given an integer `id` that represents the ID of an employee, return *the total importance value of this employee and all their direct subordinates*.

### Example 1:



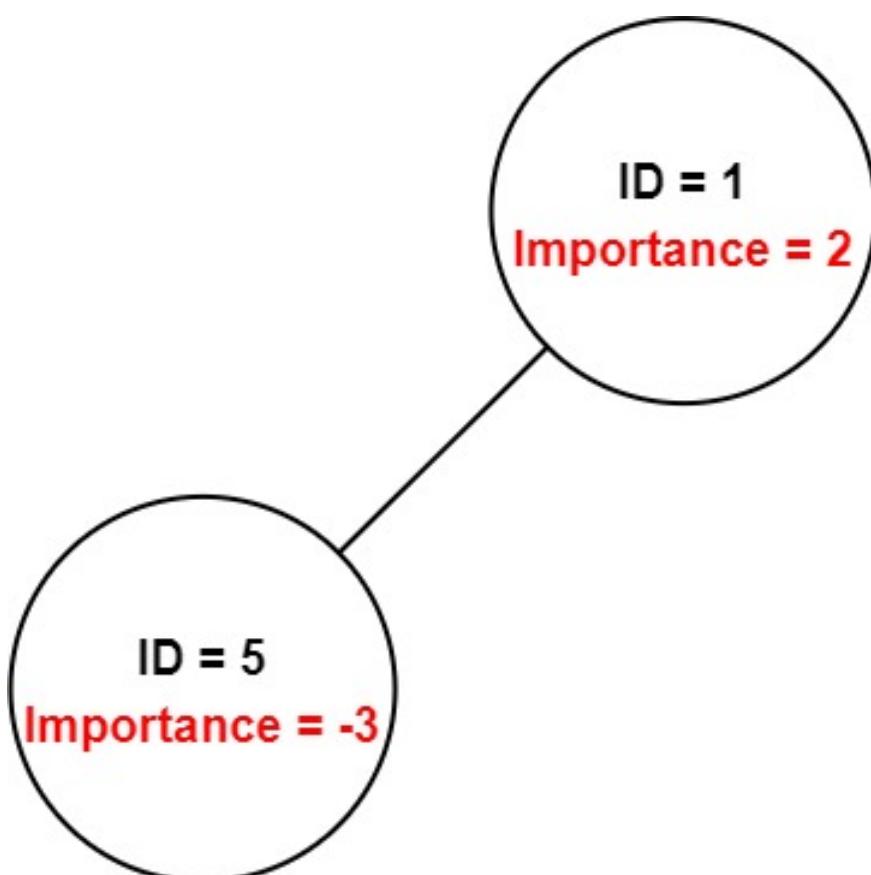
**Input:** employees = [[1,5,[2,3]],[2,3,[]],[3,3,[]]], id = 1

**Output:** 11

**Explanation:** Employee 1 has an importance value of 5 and has two direct subordinates. They both have an importance value of 3.

Thus, the total importance value of employee 1 is  $5 + 3 + 3 = 11$ .

### Example 2:



**Input:** employees = [[1,2,[5]],[5,-3,[]]], id = 5

**Output:** -3

**Explanation:** Employee 5 has an importance value of -3 and has no direct subordinates. Thus, the total importance value of employee 5 is -3.

### Constraints:

- $1 \leq \text{employees.length} \leq 2000$
  - $1 \leq \text{employees}[i].id \leq 2000$
  - All  $\text{employees}[i].id$  are **unique**.
  - $-100 \leq \text{employees}[i].importance \leq 100$
  - One employee has at most one direct leader and may have several subordinates.
  - The IDs in  $\text{employees}[i].subordinates$  are valid IDs.
-

```

/*
// Definition for Employee.
class Employee {
 public int id;
 public int importance;
 public List<Integer> subordinates;
};

*/
//DFS
class Solution {
 public int getImportance(List<Employee> employees, int id) {

 int imp = 0;
 HashMap<Integer, Employee> emap = new HashMap<>();
 for(Employee e : employees) {
 emap.put(e.id, e);
 }

 Queue<Employee> queue = new LinkedList<>();
 queue.add(emap.get(id)); //add 1st

 while(queue.size()>0) {
 Employee emp = queue.poll();
 imp += emp.importance;
 List<Integer> sub = emp.subordinates;

 for (int i = 0; i < sub.size(); i++) {

 Employee subEmp = emap.get(sub.get(i));
 // imp += subEmp.importance; //jb process hoga tb add hog
 i... yaha pr double ho ja re
 queue.add(subEmp);
 }
 }
 return imp;
 }
}

```

## 692. Top K Frequent Words ↗

Given an array of strings `words` and an integer `k`, return *the k most frequent strings*.

Return the answer **sorted** by **the frequency** from highest to lowest. Sort the words with the same frequency by their **lexicographical order**.

### Example 1:

**Input:** words = ["i", "love", "leetcode", "i", "love", "coding"], k = 2

**Output:** ["i", "love"]

**Explanation:** "i" and "love" are the two most frequent words.

Note that "i" comes before "love" due to a lower alphabetical order.

### Example 2:

**Input:** words = ["the", "day", "is", "sunny", "the", "the", "the", "sunny", "is", "is"]

**Output:** ["the", "is", "sunny", "day"]

**Explanation:** "the", "is", "sunny" and "day" are the four most frequent words,

### Constraints:

- $1 \leq \text{words.length} \leq 500$
- $1 \leq \text{words}[i] \leq 10$
- $\text{words}[i]$  consists of lowercase English letters.
- k is in the range [1, The number of **unique**  $\text{words}[i]$ ]

**Follow-up:** Could you solve it in  $O(n \log(k))$  time and  $O(n)$  extra space?

```
class Solution {
 public List<String> topKFrequent(String[] words, int k) {

 List<String> list = new ArrayList<String>();
 HashMap<String, Integer> hm = new HashMap<>();

 for (int i = 0; i < words.length; i++)
 {
 if (hm.containsKey(words[i]))
 {
 hm.put(words[i], hm.get(words[i]) +1);
 }

 else
 {
 hm.put(words[i], 1);
 }
 }

 PriorityQueue<String> q = new PriorityQueue(new Comparator<String>()
() {

 public int compare(String s1, String s2)
 {
 if (hm.get(s1) == hm.get(s2)) {
 return s1.compareTo(s2);
 }
 return hm.get(s2) - hm.get(s1);
 }
 });

 for (Map.Entry<String, Integer> s : hm.entrySet())
 {
 q.add(s.getKey());
 }

 while (!q.isEmpty() && k > 0)
 {
 String s = q.poll();
 list.add(s);
 k--;
 }

}

return list;
}
```

```
}
```

## 695. Max Area of Island

You are given an  $m \times n$  binary matrix  $\text{grid}$ . An island is a group of 1's (representing land) connected **4-directionally** (horizontal or vertical.) You may assume all four edges of the grid are surrounded by water.

The **area** of an island is the number of cells with a value 1 in the island.

Return *the maximum area of an island in  $\text{grid}$* . If there is no island, return 0.

### Example 1:

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |

**Input:**  $\text{grid} = [[0,0,1,0,0,0,0,1,0,0,0,0,0],[0,0,0,0,0,0,0,1,1,1,0,0,0],[0,1,1$

**Output:** 6

**Explanation:** The answer is not 11, because the island must be connected 4-dir

### Example 2:

**Input:**  $\text{grid} = [[0,0,0,0,0,0,0,0]]$

**Output:** 0

## Constraints:

- $m == \text{grid.length}$
- $n == \text{grid}[i].length$
- $1 \leq m, n \leq 50$
- $\text{grid}[i][j]$  is either 0 or 1.

//dfs me confusion ho raha tha revise karo jyada

```
class Solution {
 public int maxAreaOfIsland(int[][] grid) {
 int count = 0;
 for (int i = 0; i < grid.length; i++) {
 for (int j = 0; j < grid[0].length; j++)
 {
 if (grid[i][j] == 1) {
 count = Math.max(count, dfs(i, j, grid));
 }
 }
 }
 return count;
 }

 int dfs(int i, int j, int[][] grid) {

 if (i < 0 || i >= grid.length || j < 0 || j >= grid[0].length || grid[i][j] == 0)
 return 0;

 grid[i][j] = 0;

 return (dfs(i+1, j, grid) + dfs(i-1, j, grid) + dfs(i, j+1, grid)+
 dfs(i, j-1, grid) +1); //was confused in adding one
 }
}
```

## 714. Best Time to Buy and Sell Stock with Transaction Fee ↗



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

Find the maximum profit you can achieve. You may complete as many transactions as you like, but you need to pay the transaction fee for each transaction.

**Note:** You may not engage in multiple transactions simultaneously (i.e., you must sell the stock before you buy again).

### Example 1:

**Input:** `prices = [1,3,2,8,4,9]`, `fee = 2`

**Output:** 8

**Explanation:** The maximum profit can be achieved by:

- Buying at `prices[0] = 1`
- Selling at `prices[3] = 8`
- Buying at `prices[4] = 4`
- Selling at `prices[5] = 9`

The total profit is  $((8 - 1) - 2) + ((9 - 4) - 2) = 8$ .

### Example 2:

**Input:** `prices = [1,3,7,5,10,3]`, `fee = 3`

**Output:** 6

### Constraints:

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

```

class Solution {
 public int maxProfit(int[] a, int fee) {
 if (a.length == 0) return 0;

 int obsp = -a[0]; //old bought state profit..1st we have to buy
 int ossp = 0; //old sold state profit

 for (int i = 1; i < a.length; i++)
 {
 int nbsp = 0; //new bought state profit
 int nssp = 0; //new sell state profit

 if (ossp - a[i] > obsp) //bought today-->old profit se kharedge
e
 {
 nbsp = ossp - a[i];
 }
 else
 {
 nbsp = obsp;
 }
 if (obsp + a[i] - fee > ossp)//sell today
 {
 nssp = obsp + a[i] - fee;
 }
 else
 {
 nssp = ossp;
 }

 obsp = nbsp;
 ossp = nssp;
 }
 return ossp;
 }
}

```

## Same Approach Diff way

```

class Solution {
 public int maxProfit(int[] P, int fee) {

 int len = P.length, buying = -P[0], selling = 0;

 for (int i = 1; i < len; i++) {

 buying = Math.max(buying, selling - P[i]);

 selling = Math.max(selling, (buying+P[i] - fee));

 }
 return selling;
 }
}

```

## 718. Maximum Length of Repeated Subarray ↗

Given two integer arrays `nums1` and `nums2`, return *the maximum length of a subarray that appears in both arrays*.

### Example 1:

**Input:** `nums1 = [1,2,3,2,1]`, `nums2 = [3,2,1,4,7]`

**Output:** 3

**Explanation:** The repeated subarray with maximum length is `[3,2,1]`.

### Example 2:

**Input:** `nums1 = [0,0,0,0,0]`, `nums2 = [0,0,0,0,0]`

**Output:** 5

### Constraints:

- $1 \leq \text{nums1.length}, \text{nums2.length} \leq 1000$
- $0 \leq \text{nums1}[i], \text{nums2}[i] \leq 100$

```

//same as longest common subsequence...uper ka dia dekho ya necha ka aat ek
he h
//uper me 0 he h row and column me ... qki empty number h(koi digit nahi h)
class Solution {
 public int findLength(int[] nums1, int[] nums2) {

 int max = 0;
 int dp[][] = new int[nums1.length+1][nums2.length+1];
 //start from 1st row and 1st col
 for (int i = 1; i < dp.length; i++) {
 for (int j = 1; j < dp[0].length; j++) {

 if (nums1[i-1] == nums2[j-1]) {
 dp[i][j] = 1+ dp[i-1][j-1]; //one + dia(abhi tk kitna c
omon tha)
 max = Math.max(max, dp[i][j]);
 }
 }
 }

 return max;
 }
}

```

## 724. Find Pivot Index ↗

Given an array of integers `nums` , calculate the **pivot index** of this array.

The **pivot index** is the index where the sum of all the numbers **strictly** to the left of the index is equal to the sum of all the numbers **strictly** to the index's right.

If the index is on the left edge of the array, then the left sum is `0` because there are no elements to the left. This also applies to the right edge of the array.

Return *the leftmost pivot index*. If no such index exists, return `-1`.

**Example 1:**

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

**Output:** 3

**Explanation:**

The pivot index is 3.

Left sum = nums[0] + nums[1] + nums[2] = 1 + 7 + 3 = 11

Right sum = nums[4] + nums[5] = 5 + 6 = 11

### Example 2:

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

**Output:** -1

**Explanation:**

There is no index that satisfies the conditions in the problem statement.

### Example 3:

**Input:** nums = [2,1,-1]

**Output:** 0

**Explanation:**

The pivot index is 0.

Left sum = 0 (no elements to the left of index 0)

Right sum = nums[1] + nums[2] = 1 + -1 = 0

### Constraints:

- $1 \leq \text{nums.length} \leq 10^4$
- $-1000 \leq \text{nums}[i] \leq 1000$

**Note:** This question is the same as 1991: <https://leetcode.com/problems/find-the-middle-index-in-array/> (<https://leetcode.com/problems/find-the-middle-index-in-array/>)

```

class Solution {
 public int pivotIndex(int[] nums) {

 int totSum = 0;
 int leftSum = 0;

 for (int i = 0; i < nums.length; i++) {
 totSum += nums[i];
 }

 for (int i = 0; i < nums.length; i++) {
 int rightSum = totSum - leftSum - nums[i]; //qki current bhi index exclude karna h

 if (rightSum == leftSum) return i;

 leftSum += nums[i];
 }

 return -1;
 }
}

```

## 729. My Calendar I ↗

You are implementing a program to use as your calendar. We can add a new event if adding the event will not cause a **double booking**.

A **double booking** happens when two events have some non-empty intersection (i.e., some moment is common to both events.).

The event can be represented as a pair of integers `start` and `end` that represents a booking on the half-open interval `[start, end)`, the range of real numbers `x` such that `start <= x < end`.

Implement the `MyCalendar` class:

- `MyCalendar()` Initializes the calendar object.
- `boolean book(int start, int end)` Returns `true` if the event can be added to the calendar successfully without causing a **double booking**. Otherwise, return `false` and do not add the event to the calendar.

## Example 1:

### Input

```
["MyCalendar", "book", "book", "book"]
[[], [10, 20], [15, 25], [20, 30]]
```

### Output

```
[null, true, false, true]
```

### Explanation

```
MyCalendar myCalendar = new MyCalendar();
myCalendar.book(10, 20); // return True
myCalendar.book(15, 25); // return False, It can not be booked because time 1
myCalendar.book(20, 30); // return True, The event can be booked, as the firs
```

### Constraints:

- $0 \leq \text{start} < \text{end} \leq 10^9$
- At most 1000 calls will be made to book .

```

/*
cal.floorKey(start) --> logn
cal.ceilingKey(start) --> logn
*/

```

```

class MyCalendar {

 TreeMap<Integer, Integer> cal;
 public MyCalendar() {
 cal = new TreeMap<>(); //<start, end>
 }

 public boolean book(int start, int end) {

 Integer floorKey = cal.floorKey(start); //this can be null

 if (floorKey != null && start < cal.get(floorKey)) return false;

 Integer ceilKey = cal.ceilingKey(start); //after this new meeting,,,
which meeting will start
 if (ceilKey != null && end > ceilKey) return false; //if new meet
ing end time > already existing meeting start time

 cal.put(start, end);
 return true;
 }
}

/**
 * Your MyCalendar object will be instantiated and called as such:
 * MyCalendar obj = new MyCalendar();
 * boolean param_1 = obj.book(start,end);
 */

```

## 735. Asteroid Collision ↗

We are given an array `asteroids` of integers representing asteroids in a row.

For each asteroid, the absolute value represents its size, and the sign represents its direction (positive meaning right, negative meaning left). Each asteroid moves at the same speed.

Find out the state of the asteroids after all collisions. If two asteroids meet, the smaller one will explode. If both are the same size, both will explode. Two asteroids moving in the same direction will never meet.

### Example 1:

**Input:** asteroids = [5,10,-5]

**Output:** [5,10]

**Explanation:** The 10 and -5 collide resulting in 10. The 5 and 10 never collide.

### Example 2:

**Input:** asteroids = [8,-8]

**Output:** []

**Explanation:** The 8 and -8 collide exploding each other.

### Example 3:

**Input:** asteroids = [10,2,-5]

**Output:** [10]

**Explanation:** The 2 and -5 collide resulting in -5. The 10 and -5 collide resulting in 10.

### Example 4:

**Input:** asteroids = [-2,-1,1,2]

**Output:** [-2,-1,1,2]

**Explanation:** The -2 and -1 are moving left, while the 1 and 2 are moving right.

### Constraints:

- $2 \leq \text{asteroids.length} \leq 10^4$
- $-1000 \leq \text{asteroids}[i] \leq 1000$
- $\text{asteroids}[i] \neq 0$

```

/*
Say a row of asteroids is stable. What happens when a new asteroid is added
on the right?
*/
class Solution {
 public int[] asteroidCollision(int[] asteroids) {

 Stack<Integer> positive = new Stack<>();
 Stack<Integer> negative = new Stack<>();
 boolean negativeDestroyed = false;

 for (int i = 0; i < asteroids.length; i++)
 {
 int num = asteroids[i];
 negatveDestroyed = false; //new negative aaya h
 if (num < 0)
 {
 num = num*-1; //make negative positive for comparision
 while (!positive.isEmpty() && negativeDestroyed == false) //jb tk +ve me hai ya -ve distroy ni hua h
 {
 int po = positive.peek();

 if (num == po) //both destroyed
 {
 positive.pop();
 negativeDestroyed = true;
 }
 else if (num > po) //negative > positive
 {
 positive.pop();
 }
 else
 {
 negativeDestroyed = true;
 }
 }
 if (positive.isEmpty() && !negativeDestroyed) negative.push
(num*-1);
 }
 else //number +ve tha to direct +ve stack me rakh dia
 {
 positive.push(num);
 }
 }
 }
}

```

```

int totAsteroid = positive.size() + negative.size();
int[] ans = new int[totAsteroid];

while (!positive.isEmpty())
{
 ans[--totAsteroid] = positive.pop();

}

while (!negative.isEmpty())
{
 ans[--totAsteroid] = negative.pop();

}

return ans;
}
}

```

## 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<sup>th</sup> 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:

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

```
class Solution {
 public int[] dailyTemperatures(int[] T) {

 int len = T.length;

 int ans[] = new int[len];
 Stack<Integer> st = new Stack<Integer>();

 for (int i = len-1; i >= 0; i--)
 {
 while(!st.isEmpty() && T[i] >= T[st.peek()])
 {
 st.pop();
 }
 if (!st.isEmpty() && T[i] < T[st.peek()])
 {
 ans[i] = st.peek() - i;
 st.push(i);
 }
 //if stack is empty then we have to push element and put 0 in answer
 else
 {
 ans[i] = 0;
 st.push(i);
 }
 }

 return ans;
 }
}
```



You are given an integer array `cost` where `cost[i]` is the cost of  $i^{\text{th}}$  step on a staircase. Once you pay the cost, you can either climb one or two steps.

You can either start from the step with index `0`, or the step with index `1`.

Return *the minimum cost to reach the top of the floor*.

### Example 1:

**Input:** `cost = [10, 15, 20]`

**Output:** 15

**Explanation:** You will start at index 1.

- Pay 15 and climb two steps to reach the top.

The total cost is 15.

### Example 2:

**Input:** `cost = [1, 100, 1, 1, 1, 100, 1, 1, 100, 1]`

**Output:** 6

**Explanation:** You will start at index 0.

- Pay 1 and climb two steps to reach index 2.  
- Pay 1 and climb two steps to reach index 4.  
- Pay 1 and climb two steps to reach index 6.  
- Pay 1 and climb one step to reach index 7.  
- Pay 1 and climb two steps to reach index 9.  
- Pay 1 and climb one step to reach the top.

The total cost is 6.

### Constraints:

- $2 \leq \text{cost.length} \leq 1000$
- $0 \leq \text{cost}[i] \leq 999$

```

class Solution {
 public int minCostClimbingStairs(int[] cost)
 {
 int l = cost.length-1;
 int dp[] = new int[cost.length];
 // if (cost.length == 1) return cost[0];

 dp[1] = cost[1];
 dp[l-1] = cost[l-1];//last 2 fixed qki 2nd last se bhi baher jump p
 ossible h

 for (int i = l-2; i >= 0; i--)
 {
 dp[i] = cost[i] + Math.min(dp[i+1],dp[i+2]);
 }

 return dp[0] < dp[1] ? dp[0] : dp[1]; //1 step can be at 0 or 1
 }
}

```

## 752. Open the Lock ↗

You have a lock in front of you with 4 circular wheels. Each wheel has 10 slots: '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'. The wheels can rotate freely and wrap around: for example we can turn '9' to be '0', or '0' to be '9'. Each move consists of turning one wheel one slot.

The lock initially starts at '0000', a string representing the state of the 4 wheels.

You are given a list of `deadends` dead ends, meaning if the lock displays any of these codes, the wheels of the lock will stop turning and you will be unable to open it.

Given a `target` representing the value of the wheels that will unlock the lock, return the minimum total number of turns required to open the lock, or -1 if it is impossible.

**Example 1:**

**Input:** deadends = ["0201", "0101", "0102", "1212", "2002"], target = "0202"

**Output:** 6

**Explanation:**

A sequence of valid moves would be "0000" -> "1000" -> "1100" -> "1200" -> "1212". Note that a sequence like "0000" -> "0001" -> "0002" -> "0102" -> "0202" would not work because the wheels of the lock become stuck after the display becomes the deadend "0102".

## Example 2:

**Input:** deadends = ["8888"], target = "0009"

**Output:** 1

**Explanation:**

We can turn the last wheel in reverse to move from "0000" -> "0009".

## Example 3:

**Input:** deadends = ["8887", "8889", "8878", "8898", "8788", "8988", "7888", "9888"], target = "8888"

**Output:** -1

**Explanation:**

We can't reach the target without getting stuck.

## Example 4:

**Input:** deadends = ["0000"], target = "8888"

**Output:** -1

## Constraints:

- $1 \leq \text{deadends.length} \leq 500$
- $\text{deadends}[i].length == 4$
- $\text{target.length} == 4$
- $\text{target}$  will not be in the list `deadends`.
- $\text{target}$  and  $\text{deadends}[i]$  consist of digits only.

```

//basically her digit me +1 and -1 karna hai, deadends ko handle krna hai..
bfs approch se use karna h
class Solution {
 public int openLock(String[] deadends, String target) {

 String start = "0000";
 HashSet<String> visited = new HashSet<>();

 for (String deadend : deadends) {
 visited.add(deadend); //deadvidited bhi aage jane se rokega he same as visited to ek me he kar dia
 }
 int level = -1;
 Queue<String> queue = new LinkedList<>();
 queue.add(start);

 while (!queue.isEmpty()) {
 level++;
 int size = queue.size(); //is level k locks
 for (int i = 0; i < size; i++) { //level nikaal rahai ha isliye for bhulna nahi

 String number = queue.poll();

 if (visited.contains(number)) continue;

 if (number.equals(target)) return level; //target mil gaya

 //neighbour ko dekho..isme hum generate karege
 for (String nextCom : generateNext(number)) {

 if (!visited.contains(nextCom)) queue.add(nextCom);
 }
 visited.add(number); //ab iske lia process nahi hogaye mai bhul gae t add karna*****
 }
 }
 return -1; //target tk nahi pahuche and queue khali
 }

 public Set<String> generateNext(String lock) {

 Set<String> newLocks = new HashSet<>(); /**
 char[] lockChar = lock.toCharArray();
 for (int i = 0; i < 4; i++) {

```

```

 char c = lockChar[i];
 // moving clockwise
 lockChar[i] = (c == '9') ? '0' : (char) (c + 1);
 newLocks.add(new String(lockChar));
 // moving anticlockwise
 lockChar[i] = (c == '0') ? '9' : (char) (c -1);
 newLocks.add(new String(lockChar));
 lockChar[i] = c;//make to its previous state
 }
 return newLocks;
}
}

```

## 763. Partition Labels ↗

You are given a string  $s$ . We want to partition the string into as many parts as possible so that each letter appears in at most one part.

Return *a list of integers representing the size of these parts.*

### Example 1:

**Input:**  $s = \text{"ababcbacadefegdehijhklij"}$

**Output:** [9,7,8]

**Explanation:**

The partition is "ababcaca", "defegde", "hijhklij".

This is a partition so that each letter appears in at most one part.

A partition like "ababcbacadefegde", "hijhklij" is incorrect, because it spli

### Example 2:

**Input:**  $s = \text{"eccbbbbdec"}$

**Output:** [10]

### Constraints:

- $1 \leq s.length \leq 500$
- $s$  consists of lowercase English letters.

```

class Solution {
 public List<Integer> partitionLabels(String S) // "ababcbacadefegdehijk
lij"
 {
 List<Integer> list = new ArrayList<>();

 HashMap<Character, Integer> map = new HashMap<>(); // will store las
t occurrence of each character

 for (int i = 0; i < S.length(); i++)
 {
 char ch = S.charAt(i);
 map.put(ch, i);
 }

 int j = 0;
 int lastIndex = 0;
 int firstIndex = 0;
 while (firstIndex < S.length()) // start kia 0 se
 {
 lastIndex = map.get(S.charAt(firstIndex)); // "ababcbacadefegde
hijklij" stores last index od 'a'
 j = firstIndex +1;
 while (j < lastIndex) // jb tk last index pr na aa jae..matlab p
artition khatam ni hua h
 {
 //last index ko update kar rai if new character comes in be
tween
 lastIndex = Math.max(lastIndex, map.get(S.charAt(j)));
 j++;
 }
 list.add(lastIndex - firstIndex +1); //calculating length of p
artition

 firstIndex = lastIndex+1; // first index or new partition star
t
 }
 return list;
 }
}

```



You're given strings `jewels` representing the types of stones that are jewels, and `stones` representing the stones you have. Each character in `stones` is a type of stone you have. You want to know how many of the stones you have are also jewels.

Letters are case sensitive, so "a" is considered a different type of stone from "A".

### Example 1:

**Input:** `jewels = "aA"`, `stones = "aAAAbbbb"`

**Output:** 3

### Example 2:

**Input:** `jewels = "z"`, `stones = "ZZ"`

**Output:** 0

### Constraints:

- $1 \leq \text{jewels.length}, \text{stones.length} \leq 50$
- `jewels` and `stones` consist of only English letters.
- All the characters of `jewels` are **unique**.

```

class Solution {
 public int numJewelsInStones(String J, String S) {

 Set<Character> s = new HashSet<Character>();

 int i,c = 0;
 for (i = 0; i < J.length(); i++)
 {
 s.add(J.charAt(i));
 }

 for (i = 0; i < S.length(); i++)
 {
 if (s.contains(S.charAt(i)))
 {
 c++;
 }
 }

 return c;
 }
}

```

## **Second Approach**

```

class Solution {
 public int numJewelsInStones(String J, String S) {

 int c = 0, i;
 for (i = 0; i < S.length(); i++)
 {
 if (J.indexOf(S.charAt(i)) != -1)
 {
 c++;
 }
 }

 return c;
 }
}

```

# 792. Number of Matching Subsequences



Given a string `s` and an array of strings `words`, return *the number of words[i] that is a subsequence of s*.

A **subsequence** of a string is a new string generated from the original string with some characters (can be none) deleted without changing the relative order of the remaining characters.

- For example, "ace" is a subsequence of "abcde".

## Example 1:

**Input:** `s = "abcde", words = ["a", "bb", "acd", "ace"]`

**Output:** 3

**Explanation:** There are three strings in words that are a subsequence of `s`: "a", "acd", and "ace".

## Example 2:

**Input:** `s = "dsahjpjauf", words = ["ahjpjau", "ja", "ahbwzgqnu", "tnmlanowax"]`

**Output:** 2

## Constraints:

- $1 \leq s.length \leq 5 * 10^4$
- $1 \leq \text{words.length} \leq 5000$
- $1 \leq \text{words}[i].length \leq 50$
- `s` and `words[i]` consist of only lowercase English letters.

---

taking time and TC nahi pata

```

class Solution {
 public int numMatchingSubseq(String s, String[] words) {
 int count = 0;
 HashMap<Character, ArrayList<String>> map = new HashMap<>();

 for (int i = 0; i < s.length(); i++) { //s k her character ko map me
 as a key rakho
 if (!map.containsKey(s.charAt(i))){
 map.put(s.charAt(i), new ArrayList<>());
 }
 }
 for (String word : words) { //value me vo word hai jo key se start
 ho rai

 char firstChar = word.charAt(0);
 if (map.containsKey(firstChar)) {
 ArrayList<String> existingList = map.get(firstChar);
 existingList.add(word);
 map.put(firstChar, existingList);
 }
 }

 for (int j = 0; j < s.length(); j++) { //s me traverse kar raoi
 char toCheck = s.charAt(j); //char in s

 ArrayList<String> al = map.get(toCheck); //us character k corre
 sponding arrayList
 map.put(toCheck, new ArrayList<>()); //map k us key ko empty ka
 ro qki hume list le lia

 for (int i = 0; i < al.size(); i++) { //ab us list me ghoom rai

 String inAl = al.get(i); //list k ander k word
 if (inAl.length() == 1){ //agr 1 length ka hai to ab gayab
 matlb pura tha 's' me islia count++
 count++;
 continue;
 }

 String newString = inAl.substring(1); //first letter hata k
 baki ki string ab map me apni jagah dhudhge
 char fchar = newString.charAt(0); //first character he key
 banega

 if (map.containsKey(fchar)) { //map me hai to vo word daal
 do varna vo word baher
 }
 }
 }
}
```

```

 ArrayList<String> listAtNew = map.get(fchar); //us key
k corresponding ArrayList lia and usme abhi ka word add kar dia
 listAtNew.add(newString);
 map.put(fchar, listAtNew);
 }
}
return count;
}

}

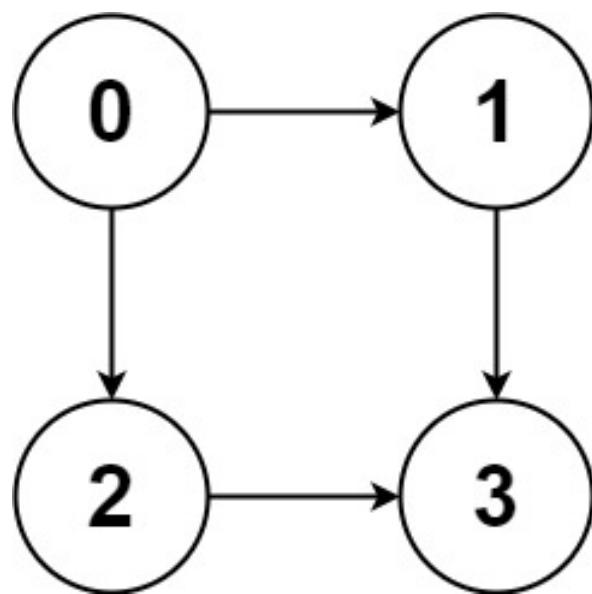
```

## 797. All Paths From Source to Target ↗ ▾

Given a directed acyclic graph (**DAG**) of  $n$  nodes labeled from  $0$  to  $n - 1$ , find all possible paths from node  $0$  to node  $n - 1$  and return them in **any order**.

The graph is given as follows: `graph[i]` is a list of all nodes you can visit from node  $i$  (i.e., there is a directed edge from node  $i$  to node `graph[i][j]`).

### Example 1:

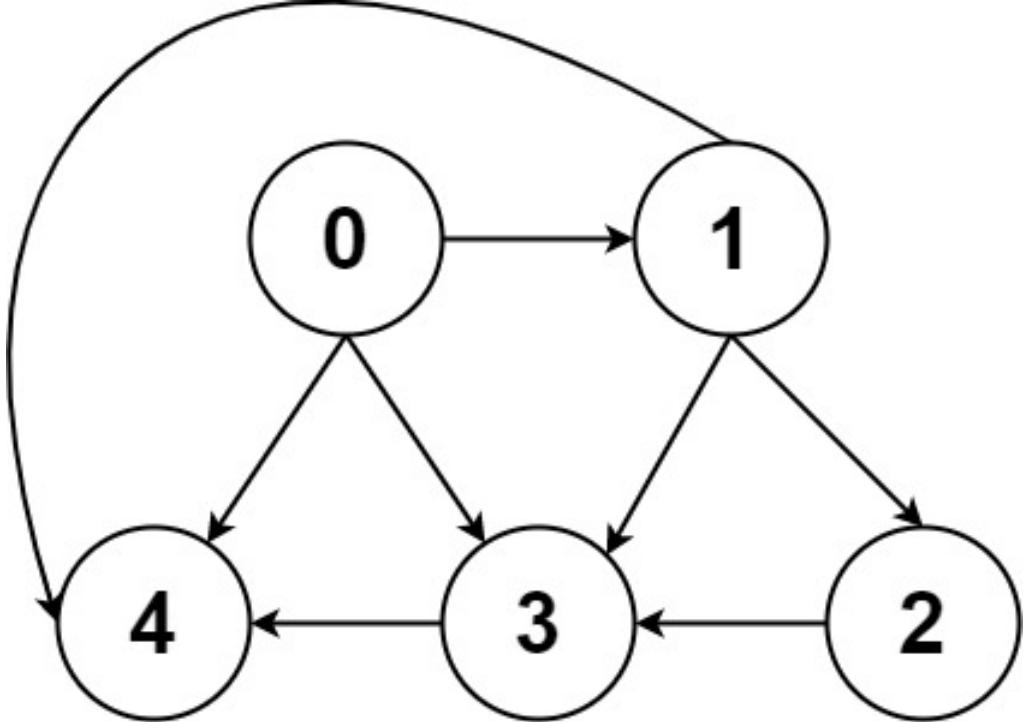


**Input:** `graph = [[1,2],[3],[3],[]]`

**Output:** `[[0,1,3],[0,2,3]]`

**Explanation:** There are two paths:  $0 \rightarrow 1 \rightarrow 3$  and  $0 \rightarrow 2 \rightarrow 3$ .

### Example 2:



**Input:** graph = [[4,3,1],[3,2,4],[3],[4],[]]

**Output:** [[0,4],[0,3,4],[0,1,3,4],[0,1,2,3,4],[0,1,4]]

### Example 3:

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

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

### Example 4:

**Input:** graph = [[1,2,3],[2],[3],[]]

**Output:** [[0,1,2,3],[0,2,3],[0,3]]

### Example 5:

**Input:** graph = [[1,3],[2],[3],[]]

**Output:** [[0,1,2,3],[0,3]]

### Constraints:

- $n == \text{graph.length}$
- $2 \leq n \leq 15$
- $0 \leq \text{graph}[i][j] < n$
- $\text{graph}[i][j] \neq i$  (i.e., there will be no self-loops).
- All the elements of  $\text{graph}[i]$  are **unique**.
- The input graph is **guaranteed** to be a **DAG**.

```

//0 hamesha source hai last element hamesha target
// jitni rows utni nodes..nodes bhi 0 se start ho re
//isme graph ko list of list me badalne ki jarurat nahi
//is graph me dia h ki cyclic nahi h and directed h to visited ki jarurat nahi
//dont forget to add starting point
class Solution {
 public List<List<Integer>> allPathsSourceTarget(int[][] graph) {

 System.out.println(Arrays.deepToString(graph));
 List<List<Integer>> ans = new ArrayList<>();
 // boolean[] visited = new boolean[graph.length];

 ArrayList<Integer> currList = new ArrayList<>();
 currList.add(0); //dont forget to add starting point

 allPath(graph, 0, graph.length-1, currList, ans);
 return ans;
 }

 public void allPath(int[][] adj, int start, int target, ArrayList<Integer> currList, List<List<Integer>> ans)
 {

 if (start == target) {
 ans.add(new ArrayList<>(currList));
 }

 // visited[start] = true;
 for (int neighbour : adj[start]) {
 // if (!visited[neighbour]) {
 // visited[neighbour] = true;
 currList.add(neighbour);
 allPath(adj, neighbour, target, currList, ans);
 // visited[neighbour] = false;
 currList.remove(currList.size()-1);
 // }
 }
 }
}

```



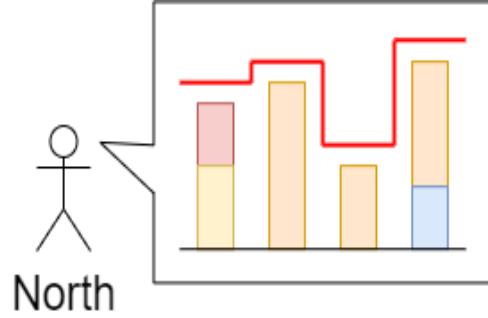
There is a city composed of  $n \times n$  blocks, where each block contains a single building shaped like a vertical square prism. You are given a **0-indexed**  $n \times n$  integer matrix `grid` where `grid[r][c]` represents the **height** of the building located in the block at row `r` and column `c`.

A city's **skyline** is the the outer contour formed by all the building when viewing the side of the city from a distance. The **skyline** from each cardinal direction north, east, south, and west may be different.

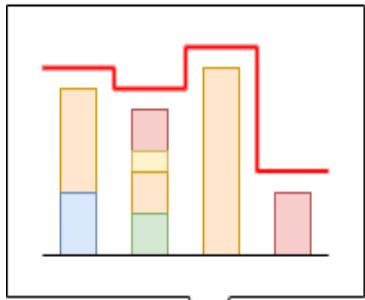
We are allowed to increase the height of **any number of buildings by any amount** (the amount can be different per building). The height of a `0`-height building can also be increased. However, increasing the height of a building should **not** affect the city's **skyline** from any cardinal direction.

Return *the maximum total sum that the height of the buildings can be increased by without changing the city's skyline from any cardinal direction.*

**Example 1:**



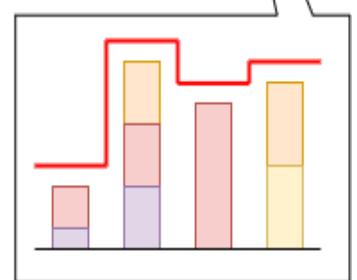
North



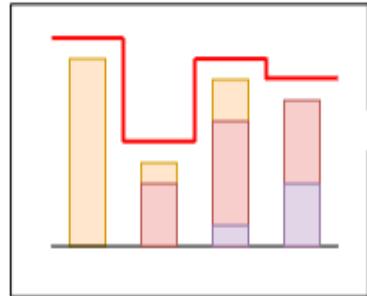
West

|   |   |   |   |
|---|---|---|---|
| 3 | 0 | 8 | 4 |
| 2 | 4 | 5 | 7 |
| 9 | 2 | 6 | 3 |
| 0 | 3 | 1 | 0 |

East



South



**Input:** grid = [[3,0,8,4],[2,4,5,7],[9,2,6,3],[0,3,1,0]]

**Output:** 35

**Explanation:** The building heights are shown in the center of the above image.

The skylines when viewed from each cardinal direction are drawn in red.

The grid after increasing the height of buildings without affecting skylines

```
gridNew = [[8, 4, 8, 7],
 [7, 4, 7, 7],
 [9, 4, 8, 7],
 [3, 3, 3, 3]]
```

**Example 2:**

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

**Output:** 0

**Explanation:** Increasing the height of any building will result in the skyline

### Constraints:

- n == grid.length
  - n == grid[r].length
  - 2 <= n <= 50
  - 0 <= grid[r][c] <= 100
-

```

//we can increase build but cannot make them mor than max height in its row
and col
class Solution {
 public int maxIncreaseKeepingSkyline(int[][] grid) {

 int sum = 0;
 int n = grid.length;
 int[] maxRow = new int[n];
 int[] maxCol = new int[n];

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

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

 maxRow[i] = Math.max(maxRow[i], grid[i][j]); //storing max
for ith row
 maxCol[j] = Math.max(maxCol[j], grid[i][j]); //storing max
for jth column
 }
 }

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

 int maxRowValue = maxRow[i];
 int maxColValue = maxCol[j];

 int minVal = Math.min(maxRowValue, maxColValue); //this wil
l be mac=ximum that this row can have
 int diff = minVal-grid[i][j]; //value by which we can incre
ase the height
 sum += diff;
 }
 }
 return sum;
 }
}

```

## 820. Short Encoding of Words ↗

A **valid encoding** of an array of words is any reference string `s` and array of indices `indices` such that:

- `words.length == indices.length`
- The reference string `s` ends with the '#' character.
- For each index `indices[i]`, the **substring** of `s` starting from `indices[i]` and up to (but not including) the next '#' character is equal to `words[i]`.

Given an array of `words`, return the **length of the shortest reference string s possible of any valid encoding** of `words`.

### Example 1:

**Input:** `words = ["time", "me", "bell"]`

**Output:** 10

**Explanation:** A valid encoding would be `s = "time#bell#"` and `indices = [0, 2, 5]`.

`words[0] = "time"`, the substring of `s` starting from `indices[0] = 0` to the next

`words[1] = "me"`, the substring of `s` starting from `indices[1] = 2` to the next

`words[2] = "bell"`, the substring of `s` starting from `indices[2] = 5` to the next

### Example 2:

**Input:** `words = ["t"]`

**Output:** 2

**Explanation:** A valid encoding would be `s = "t#"` and `indices = [0]`.

### Constraints:

- `1 <= words.length <= 2000`
- `1 <= words[i].length <= 7`
- `words[i]` consists of only lowercase letters.

```

class Solution {
 public int minimumLengthEncoding(String[] words)
 {
 Set<String> set = new HashSet<>();
 Set<String> set2 = new HashSet<>(); //taken 2 set to avoid concurrent modification exception
 StringBuilder sb = new StringBuilder();

 for (int i = 0; i < words.length; i++)
 {
 set.add(words[i]);
 set2.add(words[i]);
 }
 for (String s : set)
 {
 for (int i = 1; i < s.length(); i++)
 {
 String word = s.substring(i, s.length()); //koi bhi substring ho to hata do
 if(set2.contains(word))
 {
 set2.remove(word); //yaha pr agar hum set1 se he remove karege tb concurrent modification exception aaega
 }
 }
 }
 for (String s : set2)
 {
 sb.append(s);
 sb.append("#");
 }
 return sb.length();
 }
}

```

## 823. Binary Trees With Factors ↗

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

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

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

### Example 1:

**Input:** arr = [2,4]

**Output:** 3

**Explanation:** We can make these trees: [2], [4], [4, 2, 2]

### Example 2:

**Input:** arr = [2,4,5,10]

**Output:** 7

**Explanation:** We can make these trees: [2], [4], [5], [10], [4, 2, 2], [10, 2, 5], [10, 5, 2].

### Constraints:

- $1 \leq \text{arr.length} \leq 1000$
  - $2 \leq \text{arr}[i] \leq 10^9$
  - All the values of arr are **unique**.
-

```

class Solution {
 public int numFactoredBinaryTrees(int[] arr) {

 // int[] a = arr.clone();

 Arrays.sort(arr);

 HashMap<Integer, Long> map = new HashMap<>();

 for (int i = 0; i < arr.length; i++)
 {
 long val = 1L;
 for (int j = 0; j <=i; j++)
 {
 if(arr[i] % arr[j] == 0 && map.containsKey(arr[i]/arr[j]))
// factor h, and quotient map me bhi h
 {
 val = val + (map.get(arr[i]/arr[j]) * map.get(arr[j]));
//factors k factors ko bhi include karna h islia * kia
 }
 }
 map.put(arr[i], val);
 }

 long tot = 0L;
 for(Long v : map.values())
 {
 tot += (long)v;
 }
 return (int)(tot%(1000000000+7));

 }
}

```

## 833. Find And Replace in String ↗

You are given a **0-indexed** string  $s$  that you must perform  $k$  replacement operations on. The replacement operations are given as three **0-indexed** parallel arrays,  $\text{indices}$ ,  $\text{sources}$ , and  $\text{targets}$ , all of length  $k$ .

To complete the  $i^{\text{th}}$  replacement operation:

1. Check if the **substring** `sources[i]` occurs at index `indices[i]` in the **original string** `s`.
2. If it does not occur, **do nothing**.
3. Otherwise if it does occur, **replace** that substring with `targets[i]`.

For example, if `s = "abcd"`, `indices[0] = 0`, `sources[0] = "ab"`, and `targets[0] = "eee"`, then the result of this replacement will be `"eeecd"`.

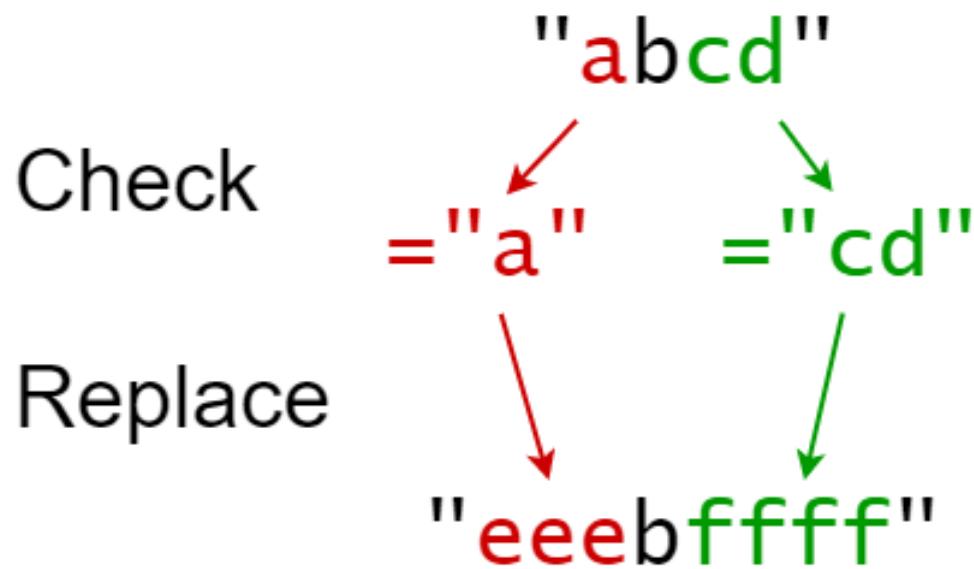
All replacement operations must occur **simultaneously**, meaning the replacement operations should not affect the indexing of each other. The testcases will be generated such that the replacements will **not overlap**.

- For example, a testcase with `s = "abc"`, `indices = [0, 1]`, and `sources = ["ab", "bc"]` will not be generated because the "ab" and "bc" replacements overlap.

Return *the resulting string* after performing all replacement operations on `s`.

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

### Example 1:

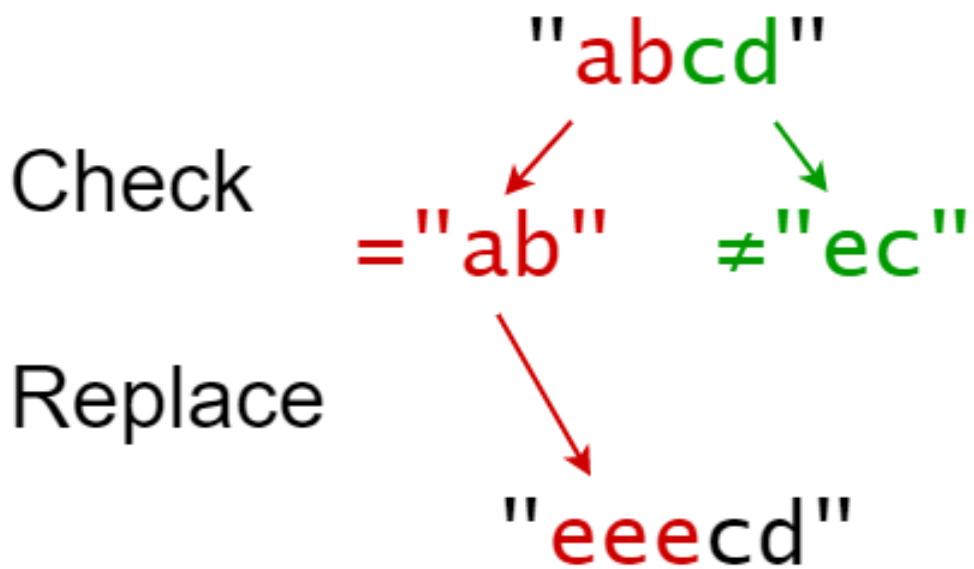


**Input:** `s = "abcd"`, `indices = [0, 2]`, `sources = ["a", "cd"]`, `targets = ["eee", "ffff"]`  
**Output:** "eeebffff"

#### Explanation:

"a" occurs at index 0 in `s`, so we replace it with "eee".  
"cd" occurs at index 2 in `s`, so we replace it with "ffff".

### Example 2:



**Input:** s = "abcd", indices = [0, 2], sources = ["ab", "ec"], targets = ["eee", ]

**Output:** "eeecd"

**Explanation:**

"ab" occurs at index 0 in s, so we replace it with "eee".

"ec" does not occur at index 2 in s, so we do nothing.

### Constraints:

- $1 \leq s.length \leq 1000$
- $k == \text{indices.length} == \text{sources.length} == \text{targets.length}$
- $1 \leq k \leq 100$
- $0 \leq \text{indexes}[i] < s.length$
- $1 \leq \text{sources}[i].length, \text{targets}[i].length \leq 50$
- s consists of only lowercase English letters.
- sources[i] and targets[i] consist of only lowercase English letters.

```
/*
Mistakes that happened:
I assumed indices array to be sorted
I used index of but we are not searching for 1st occurance so it went wrong
..then i switched to use substring
*/

class Solution {
 public String findReplaceString(String s, int[] indices, String[] sources, String[] targets) {

 StringBuffer sb = new StringBuffer();
 HashMap<Integer, List<String>> map = new HashMap<>();

 for (int i = 0; i < indices.length; i++) {
 List<String> temp = new ArrayList<>();
 temp.add(sources[i]);
 temp.add(targets[i]);
 map.put(indices[i], temp);
 }
 // System.out.println(Collections.singletonList(map));
 // System.out.println(s.indexOf("vo"));
 for (int i = 0; i < s.length(); i++) {
 if (map.containsKey(i)) {
 String toReplace = map.get(i).get(0);
 String replaceWith = map.get(i).get(1);

 String subString = s.substring(i,i+toReplace.length());

 if (subString.equals(toReplace)) {
 sb.append(replaceWith);
 i = i+toReplace.length()-1;
 // System.out.println(toReplace + " "+ replaceWith);
 }
 else {
 sb.append(s.charAt(i));
 }
 }
 else {
 sb.append(s.charAt(i));
 }
 }
 return sb.toString();
 }
}
```

```
}
```

## 841. Keys and Rooms ↗

There are  $n$  rooms labeled from  $0$  to  $n - 1$  and all the rooms are locked except for room  $0$ . Your goal is to visit all the rooms. However, you cannot enter a locked room without having its key.

When you visit a room, you may find a set of **distinct keys** in it. Each key has a number on it, denoting which room it unlocks, and you can take all of them with you to unlock the other rooms.

Given an array `rooms` where `rooms[i]` is the set of keys that you can obtain if you visited room  $i$ , return `true` if you can visit **all** the rooms, or `false` otherwise.

### Example 1:

**Input:** `rooms = [[1],[2],[3],[]]`

**Output:** `true`

**Explanation:**

We visit room  $0$  and pick up key  $1$ .

We then visit room  $1$  and pick up key  $2$ .

We then visit room  $2$  and pick up key  $3$ .

We then visit room  $3$ .

Since we were able to visit every room, we return `true`.

### Example 2:

**Input:** `rooms = [[1,3],[3,0,1],[2],[0]]`

**Output:** `false`

**Explanation:** We can not enter room number  $2$  since the only key that unlocks it is  $1$ , but we do not have key  $1$ .

### Constraints:

- $n == \text{rooms.length}$
- $2 \leq n \leq 1000$

- $0 \leq \text{rooms}[i].\text{length} \leq 1000$
  - $1 \leq \sum(\text{rooms}[i].\text{length}) \leq 3000$
  - $0 \leq \text{rooms}[i][j] < n$
  - All the values of `rooms[i]` are **unique**.
-

```

class Solution {
 public boolean canVisitAllRooms(List<List<Integer>> rooms) {

 int totalRooms = rooms.size();

 boolean[] visited = new boolean[totalRooms];
 visited[0] = true;

 dfs(rooms, 0, visited); //sending current room****confusion rehta
h list(0) ya 0 beju

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

 void dfs(List<List<Integer>> rooms, int currRoom, boolean[] visited) {

 //get keys of other room
 for (int neighbour : rooms.get(currRoom)) { //currRoom k ander ki key
s ek ek kar k aa re

 if (!visited[neighbour]) {
 visited[neighbour] = true; //jo jo room visit kia use mark
karo
 dfs(rooms, neighbour, visited);
 }
 }
 }

/*
index ---> list (0 room beja, fir jb get(0) kis to hume 1 mila:::0 child li
st ka part nahi h)
0 -> 1
1 -> 2
2 -> 3
3 -> []
*/

```

This is an **interactive problem**.

You are given an array of **unique** strings `wordlist` where `wordlist[i]` is 6 letters long, and one word in this list is chosen as `secret`.

You may call `Master.guess(word)` to guess a word. The guessed word should have type `string` and must be from the original list with 6 lowercase letters.

This function returns an `integer` type, representing the number of exact matches (value and position) of your guess to the `secret` word. Also, if your guess is not in the given `wordlist`, it will return -1 instead.

For each test case, you have exactly 10 guesses to guess the word. At the end of any number of calls, if you have made 10 or fewer calls to `Master.guess` and at least one of these guesses was `secret`, then you pass the test case.

### Example 1:

**Input:** secret = "acckzz", wordlist = ["acckzz", "ccbazz", "eiowzz", "abcczz"], numguesses = 10  
**Output:** You guessed the secret word correctly.

#### Explanation:

`master.guess("aaaaaa")` returns -1, because "aaaaaa" is not in `wordlist`.

`master.guess("acckzz")` returns 6, because "acckzz" is `secret` and has all 6 matches.

`master.guess("ccbazz")` returns 3, because "ccbazz" has 3 matches.

`master.guess("eiowzz")` returns 2, because "eiowzz" has 2 matches.

`master.guess("abcczz")` returns 4, because "abcczz" has 4 matches.

We made 5 calls to `master.guess` and one of them was the `secret`, so we pass the test case.

### Example 2:

**Input:** secret = "hamada", wordlist = ["hamada", "khaled"], numguesses = 10  
**Output:** You guessed the secret word correctly.

### Constraints:

- $1 \leq \text{wordlist.length} \leq 100$
- $\text{wordlist}[i].length == 6$
- $\text{wordlist}[i]$  consist of lowercase English letters.
- All the strings of `wordlist` are **unique**.

- secret exists in wordlist .
  - numguesses == 10
- 

Passing 6/8

```

/**
 * // This is the Master's API interface.
 * // You should not implement it, or speculate about its implementation
 * interface Master {
 * public int guess(String word) {}
 * }
 */
class Solution {
 public void findSecretWord(String[] wordlist, Master master) {

 int start = 0;
 for (int j = 0; j < 10 && start < wordlist.length; j++) {
 String currGuess = wordlist[start];
 while (start < wordlist.length-2 && currGuess.equals("")) {
 start++;
 currGuess = wordlist[start];
 }start++;
 int correctGuess = master.guess(currGuess);
 if (correctGuess > 0) {

 for (int i = wordlist.length-1; i >= 0; i--) {
 int matchingChar = matchInArray(currGuess, wordlist
[i]);
 if (matchingChar < correctGuess) //this is not matching
at all
 {
 wordlist[i] = "";
 }
 }
 }
 }
 }

 private int matchInArray(String guess, String match) {
 int count = 0;
 for (int i = 0; i < match.length(); i++) {
 if (guess.charAt(i) == match.charAt(i)) {
 count++;
 }
 }
 return count;
 }
}

```

# 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 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 and 8 become a fleet, meeting each other at 12.

The car starting at 0 doesn't catch up to any other car, so it is a fleet by itself.

The cars starting at 5 and 3 become a fleet, meeting each other at 6.

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

## 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**.
- $0 < \text{speed}[i] \leq 10^6$

```

class Solution {
 public int carFleet(int target, int[] position, int[] speed) {
 int fleet= 1;
 Car[] cars = new Car[position.length];
 for (int i = 0; i < position.length; i++) {

 Car cr = new Car();
 cr.speed = speed[i];
 cr.pos = target-position[i]; //dist left to cover
 cr.time = (double)(target-position[i])/speed[i];
 cars[i] = cr;
 }
 Arrays.sort(cars, (a,b) -> -(a.pos - b.pos));
 // System.out.print(Arrays.toString(cars));

 double lastCarTime = cars[cars.length-1].time;
 for (int i = cars.length-2; i >= 0 ; i--) {

 double currCarTime = cars[i].time;
 if (currCarTime <= lastCarTime){
 currCarTime = lastCarTime;
 }
 else fleet++;
 lastCarTime = currCarTime; //for next iteration
 }
 return fleet;
 }
 private class Car {
 int speed;
 int pos;
 double time;
 }
}

```

A car travels from a starting position to a destination which is `target` miles east of the starting position.

There are gas stations along the way. The gas stations are represented as an array `stations` where `stations[i] = [positioni, fueli]` indicates that the  $i^{\text{th}}$  gas station is `positioni` miles east of the starting position and has `fueli` liters of gas.

The car starts with an infinite tank of gas, which initially has `startFuel` liters of fuel in it. It uses one liter of gas per one mile that it drives. When the car reaches a gas station, it may stop and refuel, transferring all the gas from the station into the car.

Return *the minimum number of refueling stops the car must make in order to reach its destination*. If it cannot reach the destination, return `-1`.

Note that if the car reaches a gas station with `0` fuel left, the car can still refuel there. If the car reaches the destination with `0` fuel left, it is still considered to have arrived.

### Example 1:

**Input:** `target = 1, startFuel = 1, stations = []`

**Output:** `0`

**Explanation:** We can reach the target without refueling.

### Example 2:

**Input:** `target = 100, startFuel = 1, stations = [[10,100]]`

**Output:** `-1`

**Explanation:** We can not reach the target (or even the first gas station).

### Example 3:

**Input:** `target = 100, startFuel = 10, stations = [[10,60],[20,30],[30,30],[60,`

**Output:** `2`

**Explanation:** We start with `10` liters of fuel.

We drive to position `10`, expending `10` liters of fuel. We refuel from `0` liter. Then, we drive from position `10` to position `60` (expending `50` liters of fuel), and refuel from `10` liters to `50` liters of gas. We then drive to and reach the destination. We made `2` refueling stops along the way, so we return `2`.

### Constraints:

- $1 \leq \text{target}, \text{startFuel} \leq 10^9$
  - $0 \leq \text{stations.length} \leq 500$
  - $0 \leq \text{position}_i \leq \text{position}_{i+1} < \text{target}$
  - $1 \leq \text{fuel}_i < 10^9$
- 

```
/* Greedy yaha pr ab tk tk max gas lenge(40) --->10-----70(10(prev
stop)+60(fuel se pnd)) ----->(target cross ho gaya) 40(ab lia 60 gas se) 0---10(60)--
-20(30)---30(30)---60(40)-----100(target) */
```

baad me realise kia ki heap me position ki jarurat nahi naya solution:

```

//chalte raho jb tk fuel hai and raste k fuel tank queue me dalo
//bahut he corner case the OMG.....
class Solution {
 public int minRefuelStops(int target, int startFuel, int[][] stations)
 {

 if (startFuel >= target) return 0;

 int start = startFuel;
 PriorityQueue<Integer> pq = new PriorityQueue<>((a,b) -> b - a); //decending order
 int i = 0;
 int refillCount = 0;

 while (start < target) {

 while (i < stations.length && stations[i][0] <= start) { //gas station ko store karo aage kaam aa skte h
 pq.add(stations[i][1]);
 i++;
 }

 if(pq.isEmpty()) return -1; //gas station nahi mila and hum raste me he atak gae
 start = start + pq.remove(); //ab ka max dist, sbse jyada waka gas lia
 refillCount++;
 if (start >= target) return refillCount;

 }

 if (start >= target) return refillCount; //last check to verify kya humpahuch gae?
 return -1;
 }
}

```

## 875. Koko Eating Bananas ↗



Koko loves to eat bananas. There are  $n$  piles of bananas, the  $i^{\text{th}}$  pile has  $\text{piles}[i]$  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:

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

//Mistakes: Blunder I did ...always remember min hr will be 0 and not minimum in array and do ceit on double

```
class Solution {
 public int minEatingSpeed(int[] piles, int h) {

 Arrays.sort(piles);
 int minSpeed = 0; //minimum speed
 int maxSpeed = piles[piles.length-1]; // max speed 1 hr me sbse jya
da banana kha jao
 int possibleSpeed = 0;
 while (minSpeed <= maxSpeed) {
 int midSpeed = minSpeed + (maxSpeed - minSpeed)/2; //middle
speed

 if (isPossible(piles, h, midSpeed)) { //agr possible hai tb tho
re kam speed se bhi trykarege qki koko ko dhore khana pasand hai
 possibleSpeed = midSpeed;
 maxSpeed = midSpeed -1;

 } else {
 minSpeed = midSpeed +1; //agr possible nahi tha tb to spee
d badhani parege..qki sare banana kane h
 }
 }
 return possibleSpeed;
 }

 private boolean isPossible(int[] piles, int h, int bananaPerHour) {

 int hourReq = 0;
 for (int i = 0; i < piles.length; i++) {

 hourReq += Math.ceil((double)piles[i]/bananaPerHour); //dekh r
ai hai ki kha pae ya nahi

 if (hourReq > h) {
 return false;
 }
 }
 return true;
 }
}
```

# 900. RLE Iterator ↗

We can use run-length encoding (i.e., **RLE**) to encode a sequence of integers. In a run-length encoded array of even length `encoding (0-indexed)`, for all even `i`, `encoding[i]` tells us the number of times that the non-negative integer value `encoding[i + 1]` is repeated in the sequence.

- For example, the sequence `arr = [8,8,8,5,5]` can be encoded to be `encoding = [3,8,2,5]`. `encoding = [3,8,0,9,2,5]` and `encoding = [2,8,1,8,2,5]` are also valid **RLE** of `arr`.

Given a run-length encoded array, design an iterator that iterates through it.

Implement the `RLEIterator` class:

- `RLEIterator(int[] encoded)` Initializes the object with the encoded array `encoded`.
- `int next(int n)` Exhausts the next `n` elements and returns the last element exhausted in this way. If there is no element left to exhaust, return `-1` instead.

## Example 1:

### Input

```
["RLEIterator", "next", "next", "next", "next"]
```

```
[[[3, 8, 0, 9, 2, 5]], [2], [1], [1], [2]]
```

### Output

```
[null, 8, 8, 5, -1]
```

### Explanation

```
RLEIterator rLEIterator = new RLEIterator([3, 8, 0, 9, 2, 5]); // This maps to the first term of the sequence, returning 3.
rLEIterator.next(2); // exhausts 2 terms of the sequence, returning 8. The result is now [0, 9, 2, 5].
rLEIterator.next(1); // exhausts 1 term of the sequence, returning 8. The result is now [2, 5].
rLEIterator.next(1); // exhausts 1 term of the sequence, returning 5. The result is now [5].
rLEIterator.next(2); // exhausts 2 terms, returning -1. This is because the final term was exhausted but the second term did not exist. Since the last term exhausted does not exist, we return -1.
```

## Constraints:

- `2 <= encoding.length <= 1000`
- `encoding.length` is even.

- $0 \leq \text{encoding}[i] \leq 10^9$
- $1 \leq n \leq 10^9$
- At most 1000 calls will be made to `next`.

```
//Pointer will move by 2
class RLEIterator {

 int[] enc;
 int pointer;
 public RLEIterator(int[] encoding) {
 this.enc = encoding;
 this.pointer = 0;
 }

 public int next(int n) {
 int lastIndex = -1;
 while (n > 0 && pointer < enc.length) {

 if (n <= enc[pointer]){ //3 8
 enc[pointer] = enc[pointer] - n;// 3--> 2 8
 n = 0;
 lastIndex = enc[pointer+1];
 }
 else {
 n = n - enc[pointer]; //8 is exhausted
 enc[pointer] = 0;
 pointer = pointer +2;
 }
 }
 return lastIndex;
 }

}

/**
 * Your RLEIterator object will be instantiated and called as such:
 * RLEIterator obj = new RLEIterator(encoding);
 * int param_1 = obj.next(n);
 */
}
```



You are visiting a farm that has a single row of fruit trees arranged from left to right. The trees are represented by an integer array `fruits` where `fruits[i]` is the **type** of fruit the  $i^{\text{th}}$  tree produces.

You want to collect as much fruit as possible. However, the owner has some strict rules that you must follow:

- You only have **two** baskets, and each basket can only hold a **single type** of fruit. There is no limit on the amount of fruit each basket can hold.
- Starting from any tree of your choice, you must pick **exactly one fruit** from **every** tree (including the start tree) while moving to the right. The picked fruits must fit in one of your baskets.
- Once you reach a tree with fruit that cannot fit in your baskets, you must stop.

Given the integer array `fruits`, return the **maximum** number of fruits you can pick.

### Example 1:

**Input:** fruits = [1,2,1]

**Output:** 3

**Explanation:** We can pick from all 3 trees.

### Example 2:

**Input:** fruits = [0,1,2,2]

**Output:** 3

**Explanation:** We can pick from trees [1,2,2].

If we had started at the first tree, we would only pick from trees [0,1].

### Example 3:

**Input:** fruits = [1,2,3,2,2]

**Output:** 4

**Explanation:** We can pick from trees [2,3,2,2].

If we had started at the first tree, we would only pick from trees [1,2].

### Example 4:

**Input:** fruits = [3,3,3,1,2,1,1,2,3,3,4]

**Output:** 5

**Explanation:** We can pick from trees [1,2,1,1,2].

### **Constraints:**

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

```
//variable size sliding window (Aditya's video for Sliding Window)
class Solution {
 public int totalFruit(int[] fruits) {

 if (fruits.length == 0) return 0;

 int start = 0;
 int end = 0;
 int count = 0;
 int maxCount = Integer.MIN_VALUE;
 HashMap<Integer, Integer> map = new HashMap<>();

 while (end < fruits.length) {

 map.put(fruits[end], map.getOrDefault(fruits[end], 0)+1);
 count++;

 if (map.size() < 2) {
 end++;
 }
 else if (map.size() == 2){ //we got our window
 maxCount = Math.max(count, maxCount);
 end++;
 }
 else if (map.size() > 2) { //number of unique increased

 while (map.size() > 2) {
 int firstFruit = fruits[start];

 if (map.get(firstFruit) == 1) {
 map.remove(firstFruit);
 count--;
 }
 else {
 map.put(firstFruit, map.get(firstFruit)-1);
 count--;
 }
 start++;
 }
 end++;
 }
 }
 maxCount = Math.max(count, maxCount);
 return maxCount;
 }
}
```

# 905. Sort Array By Parity ↗

Given an integer array `nums`, move all the even integers at the beginning of the array followed by all the odd integers.

Return **any array** that satisfies this condition.

## Example 1:

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

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

**Explanation:** The outputs `[4,2,3,1]`, `[2,4,1,3]`, and `[4,2,1,3]` would also be ac

## Example 2:

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

**Output:** `[0]`

## Constraints:

- `1 <= nums.length <= 5000`
- `0 <= nums[i] <= 5000`

```
class Solution { public int[] sortArrayByParity(int[] A) {
 int i = 0, temp, l = A.length - 1; //if even then skip, move to last when odd while(i < A.length && i < l) { if (A[i] % 2 == 0) { i++; } else { temp = A[i]; A[i] = A[l]; A[l] = temp; l--; }
}
return A;
}
```

# 925. Long Pressed Name ↗

Your friend is typing his name into a keyboard. Sometimes, when typing a character c , the key might get *long pressed*, and the character will be typed 1 or more times.

You examine the typed characters of the keyboard. Return True if it is possible that it was your friends name, with some characters (possibly none) being long pressed.

## Example 1:

**Input:** name = "alex", typed = "aaleex"

**Output:** true

**Explanation:** 'a' and 'e' in 'alex' were long pressed.

## Example 2:

**Input:** name = "saeed", typed = "ssaaedd"

**Output:** false

**Explanation:** 'e' must have been pressed twice, but it wasn't in the typed out

## Example 3:

**Input:** name = "leelee", typed = "lleeelée"

**Output:** true

## Example 4:

**Input:** name = "laiden", typed = "laiden"

**Output:** true

**Explanation:** It's not necessary to long press any character.

## Constraints:

- $1 \leq \text{name.length} \leq 1000$
- $1 \leq \text{typed.length} \leq 1000$
- name and typed contain only lowercase English letters.

```

//pep
class Solution {
 public boolean isLongPressedName(String name, String typed) {

 if (name.length() > typed.length()) return false;
 int i = 0;
 int j = 0;

 while (i < name.length() && j < typed.length()) {

 if (name.charAt(i) == typed.charAt(j)) {
 i++;
 j++;
 }
 else {
 if (i > 0 && typed.charAt(j) == name.charAt(i-1)) { //typed
me purana h
 j++;
 }
 else return false;
 }
 }
 if (i < name.length()) return false; //i bacha h j over ho gaya "py
plrz" "ppyypllr"
 while (j < typed.length()) {//j bacha h
 if (typed.charAt(j) != name.charAt(i-1)) { //j last char of nam
e k baraber nahi
 return false;
 }
 j++;
 }
 return true;
 }
}

```

## 926. Flip String to Monotone Increasing ↗

A binary string is monotone increasing if it consists of some number of 0's (possibly none), followed by some number of 1's (also possibly none).

You are given a binary string  $s$ . You can flip  $s[i]$  changing it from 0 to 1 or from 1 to 0.

Return the minimum number of flips to make  $s$  monotone increasing.

### **Example 1:**

**Input:** s = "00110"

**Output:** 1

**Explanation:** We flip the last digit to get 00111.

### **Example 2:**

**Input:** s = "010110"

**Output:** 2

**Explanation:** We flip to get 011111, or alternatively 000111.

### **Example 3:**

**Input:** s = "00011000"

**Output:** 2

**Explanation:** We flip to get 00000000.

### **Constraints:**

- $1 \leq s.length \leq 10^5$
- $s[i]$  is either '0' or '1' .

```

class Solution {
 public int minFlipsMonoIncr(String s) {

 int zeroCount = 0;
 int oneCount = 0;
 int flip = 0;

 //find first occurrence of 1
 int i = 0;
 for (i = 0; i < s.length(); i++) {

 if (s.charAt(i) == '1') break;
 }
 //now we have first occurrence of 1, now count begin
 while (i < s.length()) {

 if (s.charAt(i) == '1') {
 oneCount++;
 }
 else zeroCount++;

 if (zeroCount > oneCount) { //110000000011
 flip = flip + oneCount; //qki 1 kam hai to better ise ko fl
ip karo abhi
 while (i < s.length() && s.charAt(i) != '1'){ //jb tk next
1 na mile
 i++;
 }
 oneCount = 0; //reset
 zeroCount = 0;
 i--; //1000000101010 qki jb bhi aisa ho raha while k karan
vo 1 k index pr rehta h..nexhe jake i++ ho jata h to ek 1 miss ho ja raha t
ha
 }
 i++;
 }

 if (zeroCount > oneCount) flip = flip + oneCount;
 else flip = flip + zeroCount;
 return flip;
 }
}

```

# 929. Unique Email Addresses

Every **valid email** consists of a **local name** and a **domain name**, separated by the '@' sign. Besides lowercase letters, the email may contain one or more '.' or '+'.

- For example, in "alice@leetcode.com", "alice" is the **local name**, and "leetcode.com" is the **domain name**.

If you add periods '.' between some characters in the **local name** part of an email address, mail sent there will be forwarded to the same address without dots in the local name. Note that this rule **does not apply to domain names**.

- For example, "alice.z@leetcode.com" and "alicez@leetcode.com" forward to the same email address.

If you add a plus '+' in the **local name**, everything after the first plus sign **will be ignored**. This allows certain emails to be filtered. Note that this rule **does not apply to domain names**.

- For example, "m.y+name@email.com" will be forwarded to "my@email.com".

It is possible to use both of these rules at the same time.

Given an array of strings `emails` where we send one email to each `email[i]`, return *the number of different addresses that actually receive mails*.

## Example 1:

**Input:** emails = ["test.email+alex@leetcode.com", "test.e.mail+bob.cathy@leetcc

**Output:** 2

**Explanation:** "testemail@leetcode.com" and "testemail@lee.tcode.com" actually

## Example 2:

**Input:** emails = ["a@leetcode.com", "b@leetcode.com", "c@leetcode.com"]

**Output:** 3

## Constraints:

- `1 <= emails.length <= 100`
- `1 <= emails[i].length <= 100`

- `email[i]` consist of lowercase English letters, `'+'`, `'.'` and `'@'`.
  - Each `emails[i]` contains exactly one `'@'` character.
  - All local and domain names are non-empty.
  - Local names do not start with a `'+'` character.
-

```
//domain name ko to kch bhi karne ki jarurat nahi h... bs local me he chane
g karna h
//O(N*M)
class Solution {
 public int numUniqueEmails(String[] emails) {

 HashSet<String> mails = new HashSet<>();

 for (String mail : emails) {

 StringBuffer modifiedMail = new StringBuffer();
 int start = 0;

 while (start < mail.length() && mail.charAt(start) != '@') {
 modifiedMail.append(mail.charAt(start));
 start++;
 }

 modifiedMail = buildLocalName(modifiedMail); //localName modifi
ed

 while (start < mail.length()) {
 modifiedMail.append(mail.charAt(start)); //join domainName
 start++;
 }

 mails.add(modifiedMail.toString());
 }
 // System.out.print(mails);
 return mails.size();
 }

 private StringBuffer buildLocalName(StringBuffer localName) {

 StringBuffer local = new StringBuffer();

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

 if (localName.charAt(i) == '.') continue;
 if (localName.charAt(i) == '+') break;

 else local.append(localName.charAt(i));
 }
 return local;
 }
}
```

```
}
```

## 931. Minimum Falling Path Sum

Given an  $n \times n$  array of integers `matrix`, return *the minimum sum of any falling path through matrix*.

A **falling path** starts at any element in the first row and chooses the element in the next row that is either directly below or diagonally left/right. Specifically, the next element from position `(row, col)` will be `(row + 1, col - 1)`, `(row + 1, col)`, or `(row + 1, col + 1)`.

### Example 1:

**Input:** `matrix = [[2,1,3],[6,5,4],[7,8,9]]`

**Output:** 13

**Explanation:** There are two falling paths with a minimum sum underlined below:

`[[2,1,3],        [[2,1,3],  
[6,5,4],        [6,5,4],  
[7,8,9]]        [7,8,9]]`

### Example 2:

**Input:** `matrix = [[-19,57],[-40,-5]]`

**Output:** -59

**Explanation:** The falling path with a minimum sum is underlined below:

`[[-19,57],  
[-40,-5]]`

### Example 3:

**Input:** `matrix = [[-48]]`

**Output:** -48

### Constraints:

- `n == matrix.length`

- $n == \text{matrix}[i].length$
  - $1 \leq n \leq 100$
  - $-100 \leq \text{matrix}[i][j] \leq 100$
- 

```

/*
Mistakes: forgot to add matrix value and else if condition(wrote only if, so it went to all condotion)
*/
class Solution {
 public int minFallingPathSum(int[][] matrix) {
 int row = matrix.length;
 int col = matrix[0].length;
 int[][] dp = new int[row][col];

 //fill last row 3rd row
 for (int i = 0; i < col; i++) {
 dp[row-1][i] = matrix[row-1][i];
 }
 //start from 2nd last row and last col
 for (int i = row-2; i >= 0; i--) {
 for (int j = col-1; j >= 0; j--) {

 if (j == col-1) dp[i][j] = matrix[i][j] + Math.min(dp[i+1]
[j-1], dp[i+1][j]);
 else if (j == 0) dp[i][j] = matrix[i][j] + Math.min(dp[i+1]
[j], dp[i+1][j+1]);
 else if(j > 0 && j < col) dp[i][j] = matrix[i][j] + Math.mi
n(dp[i+1][j-1], Math.min(dp[i+1][j], dp[i+1][j+1]));
 }
 }

 int min = Integer.MAX_VALUE;

 //find min in top row
 for (int i = 0; i < col; i++) {
 min = Math.min(min, dp[0][i]);
 }
 return min;
 }
}

```

# 937. Reorder Data in Log Files

You are given an array of `logs`. Each log is a space-delimited string of words, where the first word is the **identifier**.

There are two types of logs:

- **Letter-logs**: All words (except the identifier) consist of lowercase English letters.
- **Digit-logs**: All words (except the identifier) consist of digits.

Reorder these logs so that:

1. The **letter-logs** come before all **digit-logs**.
2. The **letter-logs** are sorted lexicographically by their contents. If their contents are the same, then sort them lexicographically by their identifiers.
3. The **digit-logs** maintain their relative ordering.

Return *the final order of the logs*.

## Example 1:

```
Input: logs = ["dig1 8 1 5 1","let1 art can","dig2 3 6","let2 own kit dig","l
Output: ["let1 art can","let3 art zero","let2 own kit dig","dig1 8 1 5 1","di
Explanation:
```

The letter-log contents are all different, so their ordering is "art can", "a  
The digit-logs have a relative order of "dig1 8 1 5 1", "dig2 3 6".

## Example 2:

```
Input: logs = ["a1 9 2 3 1","g1 act car","zo4 4 7","ab1 off key dog","a8 act
Output: ["g1 act car","a8 act zoo","ab1 off key dog","a1 9 2 3 1","zo4 4 7"]
```

## Constraints:

- $1 \leq \text{logs.length} \leq 100$
- $3 \leq \text{logs[i].length} \leq 100$
- All the tokens of `logs[i]` are separated by a **single** space.
- `logs[i]` is guaranteed to have an identifier and at least one word after the identifier.

```

class Solution {
 public String[] reorderLogFiles(String[] logs) {
 Arrays.sort(logs, (s1, s2) -> {
 String[] split1 = s1.split(" ", 2);
 String[] split2 = s2.split(" ", 2);

 boolean isDigit1 = Character.isDigit(split1[1].charAt(0));
 boolean isDigit2 = Character.isDigit(split2[1].charAt(0));

 if(!isDigit1 && !isDigit2) {
 // both letter-logs.
 int comp = split1[1].compareTo(split2[1]);
 if (comp == 0) return split1[0].compareTo(split2[0]);
 else return comp;
 } else if (isDigit1 && isDigit2) {
 // both digit-logs. So keep them in original order
 return 0;
 } else if (isDigit1 && !isDigit2) {
 // first is digit, second is letter. bring letter to forward.
 return 1;
 } else {
 // first is letter, second is digit. keep them in this order.
 return -1;
 }
 });
 return logs;
 }
}

```

## 940. Distinct Subsequences II ↗

Given a string  $s$ , return the number of ***distinct non-empty subsequences*** of  $s$ . Since the answer may be very large, return it **modulo  $10^9 + 7$** .

A **subsequence** of a string is a new string that is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (i.e., "ace" is a subsequence of "abcde" while "aec" is not.

**Example 1:**

**Input:** s = "abc"

**Output:** 7

**Explanation:** The 7 distinct subsequences are "a", "b", "c", "ab", "ac", "bc",

### Example 2:

**Input:** s = "aba"

**Output:** 6

**Explanation:** The 6 distinct subsequences are "a", "b", "ab", "aa", "ba", and

### Example 3:

**Input:** s = "aaa"

**Output:** 3

**Explanation:** The 3 distinct subsequences are "a", "aa" and "aaa".

### Constraints:

- $1 \leq s.length \leq 2000$
- s consists of lowercase English letters.

//very tricky: code --> PEP and MOD extra

```
class Solution {
 public int distinctSubseqII(String s) {

 HashMap<Character, Integer> map = new HashMap<>();
 long[] dp = new long[s.length()+1];
 int n=s.length();
 dp[0] = 1; //"" blank ka
 int MOD = 1_000_000_007;
 for (int i = 1; i < dp.length; i++) {

 dp[i] = 2*dp[i-1]%MOD;//ek baar aane ka count en baar na aane k
a
 char currentCharacter = s.charAt(i-1);//qki dp me blank tha is
ia string ka count peche chal raha h
 if (map.containsKey(currentCharacter)) { //pahle aaya hai matla
b repetition hua hoga islia minus karo
 int prevIndex = map.get(currentCharacter);
 dp[i] = (dp[i] - dp[prevIndex-1])%MOD;//ye bana hoga us tim
e bhi islia hata do

 }
 map.put(currentCharacter, i);

 }
 if(dp[n]<0) dp[n]=dp[n]+MOD;
 return (int)dp[s.length()]-1;

 }
}
```

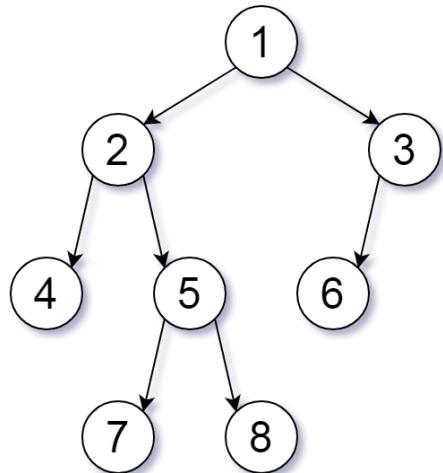
## 951. Flip Equivalent Binary Trees ↗

For a binary tree **T**, we can define a **flip operation** as follows: choose any node, and swap the left and right child subtrees.

A binary tree **X** is *flip equivalent* to a binary tree **Y** if and only if we can make **X** equal to **Y** after some number of flip operations.

Given the roots of two binary trees `root1` and `root2`, return `true` if the two trees are flip equivalent or `false` otherwise.

### Example 1:



**Input:** root1 = [1,2,3,4,5,6,null,null,null,7,8], root2 = [1,3,2,null,6,4,5,null,8,7]

**Output:** true

**Explanation:** We flipped at nodes with values 1, 3, and 5.

### Example 2:

**Input:** root1 = [], root2 = []

**Output:** true

### Example 3:

**Input:** root1 = [], root2 = [1]

**Output:** false

### Example 4:

**Input:** root1 = [0,null,1], root2 = []

**Output:** false

### Example 5:

**Input:** root1 = [0,null,1], root2 = [0,1]

**Output:** true

### Constraints:

- The number of nodes in each tree is in the range [0, 100].
- Each tree will have **unique node values** in the range [0, 99].

```

//first I thought to store all nodes in HM and put its child in list as value in map...then while traversing other tree compare childs of each node from map
///BUT BETTER SOLUTION IS BELOW
//ya to flip k sth equal ho ya vaise he ho... flip matlab left right k barabar and riht left k baraber....
class Solution {
 public boolean flipEquiv(TreeNode root1, TreeNode root2) {

 if (root1 == null && root2 == null) return true;
 if (root1 == null || root2 == null) return false;

 if (root1.val != root2.val) return false;
 //either it should be same or fliped version should be same
 return ((flipEquiv(root1.left, root2.left) && flipEquiv(root1.right, root2.right)) ||
 (flipEquiv(root1.left, root2.right) && flipEquiv(root1.right, root2.left)));
 }
}

```

## 953. Verifying an Alien Dictionary ↗

In an alien language, surprisingly, they also use English lowercase letters, but possibly in a different order. The order of the alphabet is some permutation of lowercase letters.

Given a sequence of words written in the alien language, and the order of the alphabet, return true if and only if the given words are sorted lexicographically in this alien language.

### Example 1:

**Input:** words = ["hello", "leetcode"], order = "hlabcdegijklnopqrstuvwxyz"

**Output:** true

**Explanation:** As 'h' comes before 'l' in this language, then the sequence is s

## **Example 2:**

**Input:** words = ["word", "world", "row"], order = "worldabcefghijklmnpqrstuvwxyz"

**Output:** false

**Explanation:** As 'd' comes after 'l' in this language, then words[0] > words[1]

## **Example 3:**

**Input:** words = ["apple", "app"], order = "abcdefghijklmnopqrstuvwxyz"

**Output:** false

**Explanation:** The first three characters "app" match, and the second string is

## **Constraints:**

- $1 \leq \text{words.length} \leq 100$
  - $1 \leq \text{words}[i].length \leq 20$
  - $\text{order.length} == 26$
  - All characters in  $\text{words}[i]$  and  $\text{order}$  are English lowercase letters.
-

```

class Solution {

 HashMap<Character, Integer> map = new HashMap<>();
 public boolean isAlienSorted(String[] words, String order) {
 //map k pass uska order rahega kis order me letter hone chaea
 for (int i = 0; i < order.length(); i++) {
 map.put(order.charAt(i), i);
 }

 for (int i = 1; i < words.length; i++) {
 if (!(isSmaller(words[i-1], words[i]))) return false;
 }
 return true;
 }

 private boolean isSmaller(String a, String b) {

 int minLength = Math.min(a.length(), b.length());

 for (int i = 0; i < minLength; i++) {
 char aChar = a.charAt(i);
 char bChar = b.charAt(i);

 if (map.get(aChar) < map.get(bChar)) return true; //chota pahle
 aaya to shi
 if (map.get(aChar) > map.get(bChar)) return false; //bada pahle
 aaya to galat
 //equal aaya to check karte jao
 }

 if (a.length() > b.length()) return false; //abhi tk same hai to ch
 ota word aage hona chaea

 return true;
 }
}

```

## 954. Array of Doubled Pairs ↗

Given an integer array of even length `arr`, return `true` if it is possible to reorder `arr` such that  $\text{arr}[2 * i + 1] = 2 * \text{arr}[2 * i]$  for every  $0 \leq i < \text{len}(\text{arr}) / 2$ , or `false` otherwise.

### **Example 1:**

**Input:** arr = [3,1,3,6]

**Output:** false

### **Example 2:**

**Input:** arr = [2,1,2,6]

**Output:** false

### **Example 3:**

**Input:** arr = [4,-2,2,-4]

**Output:** true

**Explanation:** We can take two groups, [-2,-4] and [2,4] to form [-2,-4,2,4] or

### **Example 4:**

**Input:** arr = [1,2,4,16,8,4]

**Output:** false

### **Constraints:**

- $2 \leq \text{arr.length} \leq 3 * 10^4$
- arr.length is even.
- $-10^5 \leq \text{arr}[i] \leq 10^5$

```

/*
Interating in map keys instead of array
Corner case : -ve odd number will not have its exact double,,,,,used getOr
Default to avoide checking contains everytime
Many corner case...took 1 day
*/

class Solution {
 public boolean canReorderDoubled(int[] arr) {

 Map<Integer, Integer> map = new TreeMap(); //sorted freq map

 for (int i : arr) {
 map.put(i, map.getOrDefault(i, 0)+1);
 }

 for (int el : map.keySet()) {

 if (map.get(el) > 0) { //> 0 check in case it is consumed earlier(el consume an hua ho)

 // if (map.get(el) == 0) continue; //matlb exhaust ho chuka h..iske half tha pahle

 if (el < 0 && el%2 != 0) return false; //-ve odd cannot be devided into half

 int target = (el < 0)? el/2 : el*2;

 if (map.get(el) > map.getOrDefault(target, 0)) return false; //qki ye source tha (source ki freq target se jyada h)

 int freqTarget = map.get(target);
 int freqEl = map.get(el); //this will be exhausted...qki source hai and freq target se kam hai

 map.put(target, freqTarget-freqEl);

 }
 }
 return true;
 }
}

```

# 977. Squares of a Sorted Array ↗

Given an integer array `nums` sorted in **non-decreasing** order, return *an array of **the squares of each number** sorted in non-decreasing order.*

## Example 1:

**Input:** `nums = [-4, -1, 0, 3, 10]`

**Output:** `[0, 1, 9, 16, 100]`

**Explanation:** After squaring, the array becomes `[16, 1, 0, 9, 100]`.

After sorting, it becomes `[0, 1, 9, 16, 100]`.

## Example 2:

**Input:** `nums = [-7, -3, 2, 3, 11]`

**Output:** `[4, 9, 9, 49, 121]`

## Constraints:

- $1 \leq \text{nums.length} \leq 10^4$
- $-10^4 \leq \text{nums}[i] \leq 10^4$
- `nums` is sorted in **non-decreasing** order.

**Follow up:** Squaring each element and sorting the new array is very trivial, could you find an  $O(n)$  solution using a different approach?

```

class Solution {
 public int[] sortedSquares(int[] nums) {

 Arrays.sort(nums);

 int[] ans = new int[nums.length];

 int l = 0, r= nums.length-1;
 int k = r;

 while (l <= r) {

 int leftsq = nums[l] * nums[l];
 int rightsq = nums[r] * nums[r];

 if (leftsq >= rightsq) {
 ans[k] = leftsq;
 l++;
 }
 else {
 ans[k] = rightsq;
 r--;
 }
 k--;
 }
 return ans;
 }
}

```

## 994. Rotting Oranges

You are given an  $m \times n$  grid where each cell can have one of three values:

- 0 representing an empty cell,
- 1 representing a fresh orange, or
- 2 representing a rotten orange.

Every minute, any fresh orange that is **4-directionally adjacent** to a rotten orange becomes rotten.

Return the minimum number of minutes that must elapse until no cell has a fresh orange. If this is impossible, return -1 .

### Example 1:

| Minute 0 | Minute 1 | Minute 2 | Minute 3 | Minute 4 |
|----------|----------|----------|----------|----------|
|          |          |          |          |          |
|          |          |          |          |          |
|          |          |          |          |          |

**Input:** grid = [[2,1,1],[1,1,0],[0,1,1]]

**Output:** 4

### Example 2:

**Input:** grid = [[2,1,1],[0,1,1],[1,0,1]]

**Output:** -1

**Explanation:** The orange in the bottom left corner (row 2, column 0) is never

### Example 3:

**Input:** grid = [[0,2]]

**Output:** 0

**Explanation:** Since there are already no fresh oranges at minute 0, the answer

### Constraints:

- $m == \text{grid.length}$
- $n == \text{grid}[i].length$
- $1 \leq m, n \leq 10$
- $\text{grid}[i][j]$  is 0, 1, or 2.

```

class Solution {
 public int orangesRotting(int[][][] grid) {

 int fresh = 0;
 Queue<int[]> queue = new LinkedList<>();
 for (int i = 0; i < grid.length; i++) {
 for (int j = 0; j < grid[0].length; j++) {

 if (grid[i][j] == 2) {

 queue.add(new int[]{i,j});
 }
 if (grid[i][j] == 1) {

 fresh++;
 }
 }
 }
 if (fresh == 0) return 0; //no fresh
 if (queue.size() == 0) return -1; //no rotten

 int time = -1; //coz we are starting with one which is already rott
en

 //BFS
 while (!queue.isEmpty()) {
 time++;

 int size = queue.size();
 for (int i = 0; i < size; i++) {

 int[] block = queue.remove();
 int row = block[0];
 int col = block[1];

 //check in all 4 direction and make tomato rotten
 if (row +1 < grid.length) { //down

 if (grid[row+1][col] == 1){ // resh
 grid[row+1][col] = 2;
 queue.add(new int[]{row+1, col}); //will rot its ne
ighbour in next min
 }
 }
 if (row -1 >= 0) { //up

```

```

 if (grid[row-1][col] == 1){ //fresh
 grid[row-1][col] = 2;
 queue.add(new int[]{row-1, col}); //will rot its ne
ighbour in next min
 }
 }
 if (col +1 < grid[0].length) { //right

 if (grid[row][col+1] == 1){ //unvisited and fresh
 grid[row][col+1] = 2;
 queue.add(new int[]{row, col+1}); //will rot its ne
ighbour in next min
 }
 }
 if (col -1 >= 0) { //left

 if (grid[row][col-1] == 1){ //unvisited and fresh
 grid[row][col-1] = 2;
 queue.add(new int[]{row, col-1}); //will rot its ne
ighbour in next min
 }
 }
}

for (int i = 0; i < grid.length; i++) {
 for (int j = 0; j < grid[0].length; j++) {

 if (grid[i][j] == 1) return -1;
 }
}
return time;
}

/*
2 1 1
0 1 1
1 0 1
*/

```

# 1010. Pairs of Songs With Total Durations Divisible by 60 ↴

You are given a list of songs where the  $i^{\text{th}}$  song has a duration of `time[i]` seconds.

Return *the number of pairs of songs for which their total duration in seconds is divisible by 60*.

Formally, we want the number of indices  $i, j$  such that  $i < j$  with  $(\text{time}[i] + \text{time}[j]) \% 60 == 0$ .

## Example 1:

**Input:** `time = [30, 20, 150, 100, 40]`

**Output:** 3

**Explanation:** Three pairs have a total duration divisible by 60:

$(\text{time}[0] = 30, \text{time}[2] = 150)$ : total duration 180

$(\text{time}[1] = 20, \text{time}[3] = 100)$ : total duration 120

$(\text{time}[1] = 20, \text{time}[4] = 40)$ : total duration 60

## Example 2:

**Input:** `time = [60, 60, 60]`

**Output:** 3

**Explanation:** All three pairs have a total duration of 120, which is divisible

## Constraints:

- $1 \leq \text{time.length} \leq 6 * 10^4$
- $1 \leq \text{time}[i] \leq 500$

---

\*\* if (`remainder1 == 0`) the `remainder2` can be either 0 or 60 if(`remainder1 != 0`) then `remainder1+ remainder2 == 60`\*\*

```

class Solution {
 public int numPairsDivisibleBy60(int[] time) {

 HashMap<Integer, Integer> map = new HashMap<>();
 int count = 0;
 //store all remainder ki freq : remainder sum of both number should
 be 60 or 0
 for (int i = 0; i < time.length; i++){

 int rem = time[i] %60;
 int reqRem = 60-rem;

 //corner Case: if rem is 0 then req rem can be (60-0)=60 or 0 i
 tseft
 if (rem == 0) {
 if (map.containsKey(rem)){
 count += map.get(rem);
 map.put(rem, map.get(rem));
 }
 }
 if (map.containsKey(reqRem)){
 count += map.get(reqRem);
 }

 if (map.containsKey(rem)) map.put(rem, map.get(rem)+1);
 else map.put(rem, 1);

 }
 return count;
 }
}

```

## 1021. Remove Outermost Parentheses ↗

A valid parentheses string is either empty "", "(" + A + ")", or A + B, where A and B are valid parentheses strings, and + represents string concatenation.

- For example, "", "()", "((()))", and "(()())()" are all valid parentheses strings.

A valid parentheses string  $s$  is primitive if it is nonempty, and there does not exist a way to split it into  $s = A + B$ , with  $A$  and  $B$  nonempty valid parentheses strings.

Given a valid parentheses string  $s$ , consider its primitive decomposition:  $s = P_1 + P_2 + \dots + P_k$ , where  $P_i$  are primitive valid parentheses strings.

Return  $s$  after removing the outermost parentheses of every primitive string in the primitive decomposition of  $s$ .

### Example 1:

**Input:**  $s = "((())())()$ "

**Output:**  $"()()()$ "

**Explanation:**

The input string is  $"((())())()$ ", with primitive decomposition  $"((()))" + "(($   
After removing outer parentheses of each part, this is  $"()()" + "()" = "()()()$

### Example 2:

**Input:**  $s = "((())())(())(())()$ "

**Output:**  $"()()()()()()$ "

**Explanation:**

The input string is  $"((())())(())(())()$ ", with primitive decomposition  $"((())$   
After removing outer parentheses of each part, this is  $"()()" + "()" + "()()()$

### Example 3:

**Input:**  $s = "()()$ "

**Output:**  $"$ "

**Explanation:**

The input string is  $"()()$ , with primitive decomposition  $"()" + "()"$ .  
After removing outer parentheses of each part, this is  $"" + "" = ""$ .

### Constraints:

- $1 \leq s.length \leq 10^5$
- $s[i]$  is either ' $'$  or ' $'$ '.
- $s$  is a valid parentheses string.

```

class Solution {
 public String removeOuterParentheses(String S) {
 //((()())((())((())))
 StringBuilder s = new StringBuilder();
 int count = 0;

 for (char ch : S.toCharArray())
 {
 if (ch == '(')
 {
 if (count >0)
 s.append(ch);

 count++;
 }
 else if (ch == ')')
 {
 count--;

 if(count > 0)
 s.append(ch);
 }
 }
 return s.toString();
 }
}

```

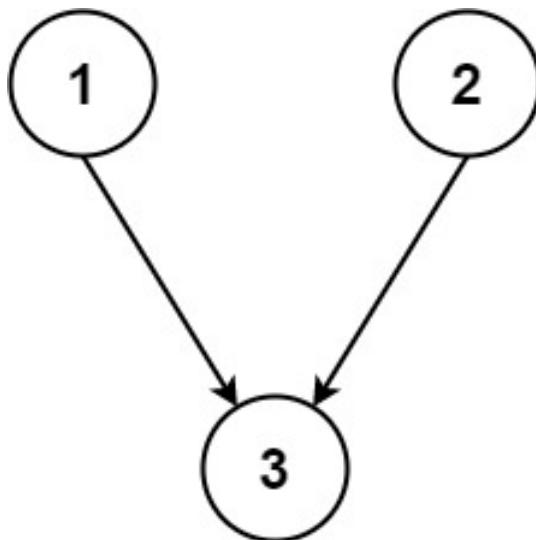
## 1136. Parallel Courses ↗

You are given an integer  $n$ , which indicates that there are  $n$  courses labeled from 1 to  $n$ . You are also given an array `relations` where `relations[i] = [prevCoursei, nextCoursei]`, representing a prerequisite relationship between course `prevCoursei` and course `nextCoursei`: course `prevCoursei` has to be taken before course `nextCoursei`.

In one semester, you can take **any number** of courses as long as you have taken all the prerequisites in the **previous** semester for the courses you are taking.

Return *the minimum number of semesters needed to take all courses*. If there is no way to take all the courses, return -1 .

### **Example 1:**



**Input:** n = 3, relations = [[1,3],[2,3]]

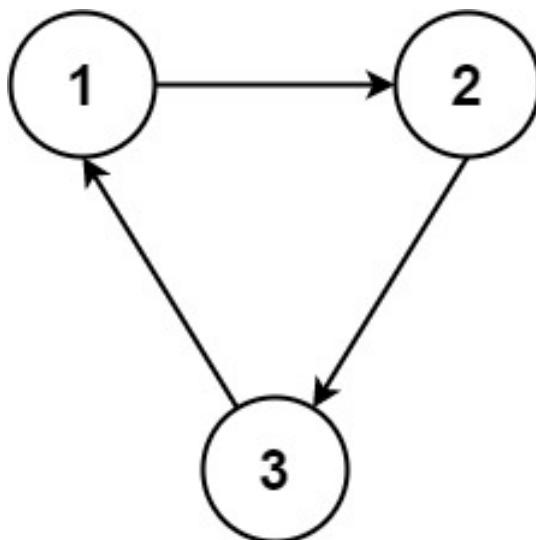
**Output:** 2

**Explanation:** The figure above represents the given graph.

In the first semester, you can take courses 1 and 2.

In the second semester, you can take course 3.

### **Example 2:**



**Input:** n = 3, relations = [[1,2],[2,3],[3,1]]

**Output:** -1

**Explanation:** No course can be studied because they are prerequisites of each

### **Constraints:**

- $1 \leq n \leq 5000$
- $1 \leq \text{relations.length} \leq 5000$
- $\text{relations}[i].length == 2$
- $1 \leq \text{prevCourse}_i, \text{nextCourse}_i \leq n$

- $\text{prevCourse}_i \neq \text{nextCourse}_i$
  - All the pairs  $[\text{prevCourse}_i, \text{nextCourse}_i]$  are **unique**.
-

```

//ye graph me 0 nahi hai dhyaan do...blunders I did in making of graph
//indegree me index dalna indegree[index] nahi
//kon sa main list banana h and kon sa sub ye bhi socha hota h
class Solution {
 public int minimumSemesters(int n, int[][] relations) {

 List<List<Integer>> graph = new ArrayList<>();
 int[] indegree = new int[n+1]; //indegree for all nodes

 for (int i = 0; i <= n; i++){
 graph.add(new ArrayList<>());
 } //now graph is ready
 System.out.print(graph);
 for (int[] r: relations) {

 int course = r[1]; //3
 int pre = r[0]; //1

 graph.get(pre).add(course); //ye pre pr kon kon course depende
nt h
 indegree[course]++;
 } //indegree for that course

 Queue<Integer> queue = new LinkedList<>();

 for (int i = 1; i < indegree.length; i++) {
 if (indegree[i] == 0){ //course in independent we can start fro
m here
 queue.add(i); //i add karna h index
 }
 }
 if (queue.isEmpty()) return -1; //all courses were dependent on eac
h other
 int sem = 0; //zero sem
 while (!queue.isEmpty()) {

 sem++;
 int size = queue.size(); //current course which has to be compl
eted

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

 int currCourse = queue.poll(); // ab ispr jo jo course dep
endent hogा uska indegree kam ho jaega

 for (int dependentCourse : graph.get(currCourse)) { //get a

```

```

 11 course which was dependent on this
 indegree[dependentCourse]--;
 if (indegree[dependentCourse] == 0) { //now we can do
this course also in next sem
 queue.add(dependentCourse);
 }

 }
}

for (int i = 0; i < indegree.length; i++) {
 if (indegree[i] != 0) return -1;
}

return sem;
}
}

```

## 1034. Coloring A Border ↗

You are given an `m x n` integer matrix `grid`, and three integers `row`, `col`, and `color`. Each value in the grid represents the color of the grid square at that location.

Two squares belong to the same **connected component** if they have the same color and are next to each other in any of the 4 directions.

The **border of a connected component** is all the squares in the connected component that are either **4-directionally** adjacent to a square not in the component, or on the boundary of the grid (the first or last row or column).

You should color the **border** of the **connected component** that contains the square `grid[row][col]` with `color`.

Return *the final grid*.

### Example 1:

**Input:** `grid = [[1,1],[1,2]]`, `row = 0`, `col = 0`, `color = 3`  
**Output:** `[[3,3],[3,2]]`

## **Example 2:**

**Input:** grid = [[1,2,2],[2,3,2]], row = 0, col = 1, color = 3

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

## **Example 3:**

**Input:** grid = [[1,1,1],[1,1,1],[1,1,1]], row = 1, col = 1, color = 2

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

## **Constraints:**

- $m == \text{grid.length}$
  - $n == \text{grid}[i].length$
  - $1 \leq m, n \leq 50$
  - $1 \leq \text{grid}[i][j], \text{color} \leq 1000$
  - $0 \leq \text{row} < m$
  - $0 \leq \text{col} < n$
-

```

//ve for visited, count rakh rahe..ye bataega neighbour me kitne log mere
tarah hai
class Solution {
 public int[][] colorBorder(int[][] grid, int row, int col, int color) {

 int rows = grid.length;
 int cols = grid[0].length;
 boolean[][] marked = new boolean[rows][cols];
 int currentColor = grid[row][col];

 dfs(currentColor, row, col, marked, grid);

 for (int i = 0; i < rows; i++) {
 for (int j = 0; j < cols; j++) {

 if (marked[i][j]) { //possible hai ki ise color karna hai..
if boundary hai

 //pakka color karna hai qki board k border pe hai!! to
pakka middle nahi ho skta
 if (i+1 >= grid.length || j+1 >= grid[0].length || i-1
< 0 || j-1 < 0){
 grid[i][j] = color;
 }
 //agar is element k charo taraf marked hai matlab middl
e hai to na mark karo...maine ulti condition likha hai.. ! laga k(sb mark n
ahi hai to color karo)
 else if (!(marked[i+1][j] && marked[i-1][j] &&
marked[i][j+1] && marked[i][j-1]))
{
 grid[i][j] = color;
 }
 }
 }
 }
 return grid;
 }

 private void dfs(int currentColor, int row, int col, boolean[][] count,
int[][] grid){

 //corner cases--> board se baher ya visited ya same color nahi jaha
failna hai
 }
}

```

```

 if (row < 0 || row >= grid.length || col < 0 || col >= grid[0].length || grid[row][col] != currentColor || count[row][col]) return;

 count[row][col] = true; //visited to hamehsa he karna hai
 dfs(currentColor, row+1, col, count, grid); //down
 dfs(currentColor, row-1, col, count, grid); //up
 dfs(currentColor, row, col+1, count, grid); //right
 dfs(currentColor, row, col-1, count, grid); //left
 }

}

```

## 1166. Design File System ↗

You are asked to design a file system that allows you to create new paths and associate them with different values.

The format of a path is one or more concatenated strings of the form: / followed by one or more lowercase English letters. For example, "/leetcode" and "/leetcode/problems" are valid paths while an empty string "" and "/" are not.

Implement the `FileSystem` class:

- `bool createPath(string path, int value)` Creates a new path and associates a value to it if possible and returns `true`. Returns `false` if the path **already exists** or its parent path **doesn't exist**.
- `int get(string path)` Returns the value associated with `path` or returns -1 if the path doesn't exist.

**Example 1:**

**Input:**

```
["FileSystem","createPath","get"]
[[],["/a",1],["/a"]]
```

**Output:**

```
[null,true,1]
```

**Explanation:**

```
FileSystem fileSystem = new FileSystem();
```

```
fileSystem.createPath("/a", 1); // return true
fileSystem.get("/a"); // return 1
```

**Example 2:****Input:**

```
["FileSystem","createPath","createPath","get","createPath","get"]
[[],["/leet",1],["/leet/code",2],["/leet/code"],["/c/d",1],["/c"]]
```

**Output:**

```
[null,true,true,2,false,-1]
```

**Explanation:**

```
FileSystem fileSystem = new FileSystem();
```

```
fileSystem.createPath("/leet", 1); // return true
fileSystem.createPath("/leet/code", 2); // return true
fileSystem.get("/leet/code"); // return 2
fileSystem.createPath("/c/d", 1); // return false because the parent path "/c"
fileSystem.get("/c"); // return -1 because this path doesn't exist.
```

**Constraints:**

- The number of calls to the two functions is less than or equal to  $10^4$  in total.
- $2 \leq \text{path.length} \leq 100$
- $1 \leq \text{value} \leq 10^9$

```
class FileSystem {

 HashMap<String, Integer> paths;
 public FileSystem() {
 paths = new HashMap<>();
 }

 public boolean createPath(String path, int value) {

 if (path == null || path.length() == 0) return false;

 if (paths.containsKey(path)) {//path already exists
 return false;
 }

 int i = 0;
 for (i = path.length() - 1 ; i >= 0; i--) {
 if (path.charAt(i) == '/') break; //yahe pr parent tk ka path h
ai jo hashmap me hoga
 }
 String parentPath = path.substring(0, i); //i tk qki hume '/' inclu
de nahi karna
 //parent path ki length agr 0 hue to matlab ye he 1st parent hai is
lia add karna h..islia yaha > 0 rakha h..varna 1st parent he add nahi kar r
aha tha
 if (parentPath.length() > 0 && !paths.containsKey(parentPath)){ //p
arent path nahi h
 return false;
 }
 else {
 paths.put(path, value);
 }
 return true;
 }

 public int get(String path) {

 if (path == null || path.length() == 0) return -1;

 return paths.getOrDefault(path,-1);
 }
}

/**
```

```
* Your FileSystem object will be instantiated and called as such:
* FileSystem obj = new FileSystem();
* boolean param_1 = obj.createPath(path,value);
* int param_2 = obj.get(path);
*/
```

## 1048. Longest String Chain ↗

You are given an array of words where each word consists of lowercase English letters.

word<sub>A</sub> is a **predecessor** of word<sub>B</sub> if and only if we can insert **exactly one** letter anywhere in word<sub>A</sub> **without changing the order of the other characters** to make it equal to word<sub>B</sub>.

- For example, "abc" is a **predecessor** of "abac" , while "cba" is not a **predecessor** of "bcad" .

A **word chain** is a sequence of words [word<sub>1</sub>, word<sub>2</sub>, ..., word<sub>k</sub>] with k >= 1 , where word<sub>1</sub> is a **predecessor** of word<sub>2</sub> , word<sub>2</sub> is a **predecessor** of word<sub>3</sub> , and so on. A single word is trivially a **word chain** with k == 1 .

Return *the length of the longest possible word chain* with words chosen from the given list of words .

### Example 1:

**Input:** words = ["a", "b", "ba", "bca", "bda", "bdca"]

**Output:** 4

**Explanation:** One of the longest word chains is ["a", "ba", "bda", "bdca"].

### Example 2:

**Input:** words = ["xbc", "pcxbcf", "xb", "cxbc", "pcxbc"]

**Output:** 5

**Explanation:** All the words can be put in a word chain ["xb", "xbc", "cxbc", "

### Example 3:

**Input:** words = ["abcd", "dbqca"]

**Output:** 1

**Explanation:** The trivial word chain ["abcd"] is one of the longest word chains. ["abcd", "dbqca"] is not a valid word chain because the ordering of the letters is invalid.

### Constraints:

- `1 <= words.length <= 1000`
- `1 <= words[i].length <= 16`
- `words[i]` only consists of lowercase English letters.

---

Substring not giving issues for 1 character also Sort array based on length : **Arrays.sort(words, (a,b)-> a.length()-b.length());**

```

class Solution {
 public int longestStrChain(String[] words) {

 Arrays.sort(words, (a,b)-> a.length()-b.length());
 System.out.println(Arrays.toString(words));

 HashMap<String, Integer> map = new HashMap<>();
 int maxValue = Integer.MIN_VALUE;

 for (String word : words) {

 int value = 0;
 int size = word.length();
 for (int i = 0; i < size; i++) {

 String sWord = word.substring(0,i)+word.substring(i+1); //skipping ith character

 if (map.containsKey(sWord)) {
 value = Math.max(value, map.get(sWord));
 }
 map.put(word, value+1);
 maxValue = Math.max(maxValue, value+1);
 }
 return maxValue;
 }
 }
}

```

## 1051. Height Checker ↗

A school is trying to take an annual photo of all the students. The students are asked to stand in a single file line in **non-decreasing order** by height. Let this ordering be represented by the integer array `expected` where `expected[i]` is the expected height of the  $i^{\text{th}}$  student in line.

You are given an integer array `heights` representing the **current order** that the students are standing in. Each `heights[i]` is the height of the  $i^{\text{th}}$  student in line (**0-indexed**).

Return *the number of indices* where `heights[i] != expected[i]`.

**Example 1:**

**Input:** heights = [1,1,4,2,1,3]

**Output:** 3

**Explanation:**

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

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

Indices 2, 4, and 5 do not match.

### Example 2:

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

**Output:** 5

**Explanation:**

heights: [5,1,2,3,4]

expected: [1,2,3,4,5]

All indices do not match.

### Example 3:

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

**Output:** 0

**Explanation:**

heights: [1,2,3,4,5]

expected: [1,2,3,4,5]

All indices match.

### Constraints:

- $1 \leq \text{heights.length} \leq 100$
- $1 \leq \text{heights}[i] \leq 100$

---

```
class Solution { public int heightChecker(int[] heights) {
 int c = 0; int[] copyArray = heights.clone(); Arrays.sort(copyArray);
```

```

 for(int i = 0; i < heights.length; i++)
 {
 if (heights[i] != copyArray[i])
 {
 c++;
 }
 }
 return c;
}

}

```

## 1094. Car Pooling ↗

There is a car with `capacity` empty seats. The vehicle only drives east (i.e., it cannot turn around and drive west).

You are given the integer `capacity` and an array `trips` where `trip[i] = [numPassengersi, fromi, toi]` indicates that the  $i^{\text{th}}$  trip has `numPassengersi` passengers and the locations to pick them up and drop them off are `fromi` and `toi` respectively. The locations are given as the number of kilometers due east from the car's initial location.

Return `true` if it is possible to pick up and drop off all passengers for all the given trips, or `false` otherwise.

### Example 1:

**Input:** `trips = [[2,1,5],[3,3,7]]`, `capacity = 4`  
**Output:** `false`

### Example 2:

**Input:** `trips = [[2,1,5],[3,3,7]]`, `capacity = 5`  
**Output:** `true`

### Example 3:

**Input:** `trips = [[2,1,5],[3,5,7]]`, `capacity = 3`  
**Output:** `true`

#### Example 4:

**Input:** trips = [[3,2,7],[3,7,9],[8,3,9]], capacity = 11

**Output:** true

#### Constraints:

- $1 \leq \text{trips.length} \leq 1000$
- $\text{trips}[i].length == 3$
- $1 \leq \text{numPassengers}_i \leq 100$
- $0 \leq \text{from}_i < \text{to}_i \leq 1000$
- $1 \leq \text{capacity} \leq 10^5$

---

faster than 60.01% video dekhna hai koi acha

```

/*
-----5
 3-----7

-----5
 3-----7
*/

class Solution {
 public boolean carPooling(int[][] trips, int capacity) {

 int currFilled = 0;
 Arrays.sort(trips, (a,b) -> a[1] - b[1]); //start first
 PriorityQueue<int[]> pq = new PriorityQueue<>((a,b) -> a[2] - b[2]); //trip which will end first

 int starting = trips[0][1]; //first trip ka start location
 int end = trips[trips.length-1][2]; //last trip ka end location
 int indexTrip = 0;
 for (int i = starting; i <= end && indexTrip < trips.length; i++){

 while (!pq.isEmpty() && pq.peek()[2] <= i) {
 //this trip is completed, uterne ka location aa gaya

 int[] completedTrip = pq.poll();
 currFilled = currFilled - completedTrip[0];
 }

 while (indexTrip < trips.length && trips[indexTrip][1] <= i) {
 //start karne ka location

 int capReq = trips[indexTrip][0]; //kitne log
 if (currFilled + capReq > capacity) return false;
 else {
 pq.add(trips[indexTrip]);
 currFilled = currFilled + capReq;
 indexTrip++;
 }
 }
 }
 return true;
 }
}

```

# 1110. Delete Nodes And Return Forest ↗

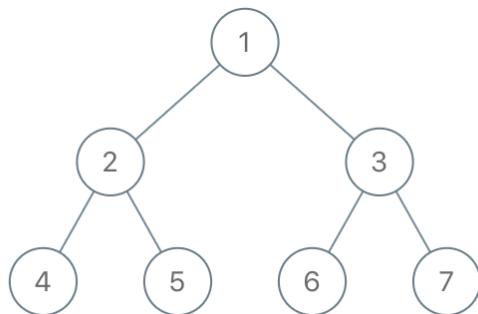


Given the `root` of a binary tree, each node in the tree has a distinct value.

After deleting all nodes with a value in `to_delete`, we are left with a forest (a disjoint union of trees).

Return the roots of the trees in the remaining forest. You may return the result in any order.

## Example 1:



**Input:** `root = [1,2,3,4,5,6,7]`, `to_delete = [3,5]`

**Output:** `[[1,2,null,4],[6],[7]]`

## Example 2:

**Input:** `root = [1,2,4,null,3]`, `to_delete = [3]`

**Output:** `[[1,2,4]]`

## Constraints:

- The number of nodes in the given tree is at most `1000`.
- Each node has a distinct value between `1` and `1000`.
- `to_delete.length <= 1000`
- `to_delete` contains distinct values between `1` and `1000`.

```

/**
 * 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;
 * }
 * }
 */
//Pura traverse karo leaf se value rakhna start karo... agr koi node delete hni h to uska left right bacha lo(if they are not null)
class Solution {
 public List<TreeNode> delNodes(TreeNode root, int[] to_delete) {
 List<TreeNode> ans = new ArrayList<>();

 Set<Integer> toDelete = new HashSet<>();
 for (int i : to_delete) toDelete.add(i);

 helper(root, toDelete, ans);
 if (!toDelete.contains(root.val)) ans.add(root); //suppose only last node got deletes then there will be nothing in list
 return ans;
 }

 private TreeNode helper(TreeNode root, Set<Integer> toDelete, List<TreeNode> ans) {
 if (root == null) return null;
 //work in post
 root.left = helper(root.left, toDelete, ans);
 root.right = helper(root.right, toDelete, ans);

 if (toDelete.contains(root.val))
 {
 //saving child put in list
 if (root.left != null) ans.add(root.left);
 if (root.right != null) ans.add(root.right);
 return null;
 }
 return root;
 }
}

```

```
}
```

# 1299. Replace Elements with Greatest Element on Right Side



Given an array `arr`, replace every element in that array with the greatest element among the elements to its right, and replace the last element with `-1`.

After doing so, return the array.

## Example 1:

**Input:** arr = [17,18,5,4,6,1]

**Output:** [18,6,6,6,1,-1]

**Explanation:**

- index 0 --> the greatest element to the right of index 0 is index 1 (18).
- index 1 --> the greatest element to the right of index 1 is index 4 (6).
- index 2 --> the greatest element to the right of index 2 is index 4 (6).
- index 3 --> the greatest element to the right of index 3 is index 4 (6).
- index 4 --> the greatest element to the right of index 4 is index 5 (1).
- index 5 --> there are no elements to the right of index 5, so we put -1.

## Example 2:

**Input:** arr = [400]

**Output:** [-1]

**Explanation:** There are no elements to the right of index 0.

## Constraints:

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

## Array:

```

class Solution { public int[] replaceElements(int[] arr) {
int max = -1; int result[] = new int [arr.length];

 for (int i = arr.length -1; i >= 0; i--)
 {
 result[i] = max;
 max = Math.max(max, arr[i]);
 }
 return result;
}

```

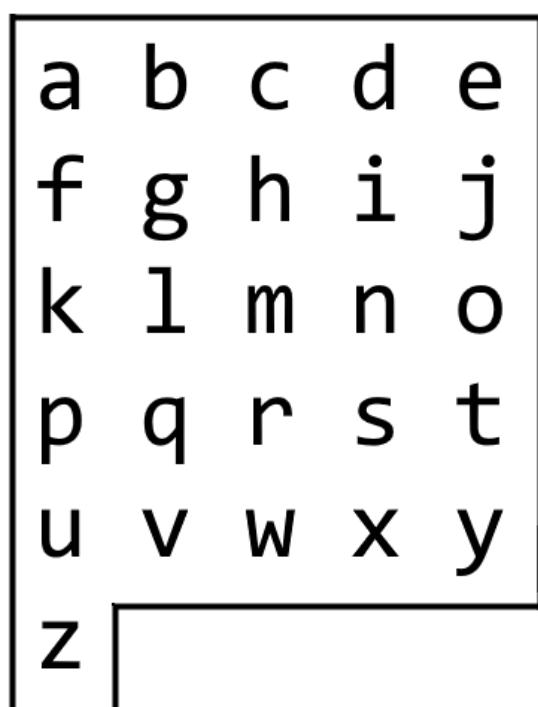
**\*\* This can be done using heap as well\*\***

}

## 1138. Alphabet Board Path ↗

On an alphabet board, we start at position  $(0, 0)$ , corresponding to character  $\text{board}[0][0]$ .

Here,  $\text{board} = ["abcde", "fghij", "klmno", "pqrst", "uvwxy", "z"]$ , as shown in the diagram below.



We may make the following moves:

- 'U' moves our position up one row, if the position exists on the board;
- 'D' moves our position down one row, if the position exists on the board;
- 'L' moves our position left one column, if the position exists on the board;

- 'R' moves our position right one column, if the position exists on the board;
- '!' adds the character `board[r][c]` at our current position (`r, c`) to the answer.

(Here, the only positions that exist on the board are positions with letters on them.)

Return a sequence of moves that makes our answer equal to `target` in the minimum number of moves. You may return any path that does so.

### **Example 1:**

```
Input: target = "leet"
Output: "DDR!UURRR!!DDD!"
```

### **Example 2:**

```
Input: target = "code"
Output: "RR!DDRR!UUL!R!"
```

### **Constraints:**

- `1 <= target.length <= 100`
- `target` consists only of English lowercase letters.

```

//row jyada h col kam hai..
//suppose a = 0 then new position of n 13:::: row = 13/5 = 2 && col = 13%5
= 3
//// U must before R, L must before D
class Solution {
 public String alphabetBoardPath(String target) {

 int currRow = 0;
 int currcol = 0;
 StringBuffer ans = new StringBuffer();
 for (int i = 0; i < target.length(); i++) {

 char charToReach = target.charAt(i);
 int position = (int)charToReach - 'a';
 int nextrow = position/5;
 int nextcol = position%5;

 while (nextrow < currRow) { //go up
 ans.append('U');
 currRow--;//*****
 }
 while (nextcol < currcol) { //go left
 ans.append('L');
 currcol--;
 }
 while (nextrow > currRow) { // go down
 ans.append('D'); //arram se ja skte h qki agr z hai to hum
z ki he column me hai..left kar k
 currRow++;
 }
 while (nextcol > currcol) {
 ans.append('R'); //go right(araam se ja skte h z se bhi qki
uper aa chuke)
 currcol++;
 }
 currRow = nextrow;//hum nextRow and next col ko assign kar rai
to inhe change na karna
 currcol = nextcol;
 ans.append('!');
 }
 return ans.toString();
 }
}

```

# 1146. Snapshot Array ↗

Implement a SnapshotArray that supports the following interface:

- `SnapshotArray(int length)` initializes an array-like data structure with the given length. **Initially, each element equals 0.**
- `void set(index, val)` sets the element at the given `index` to be equal to `val`.
- `int snap()` takes a snapshot of the array and returns the `snap_id` : the total number of times we called `snap()` minus 1 .
- `int get(index, snap_id)` returns the value at the given `index` , at the time we took the snapshot with the given `snap_id`

## Example 1:

**Input:** ["SnapshotArray","set","snap","set","get"]

[[3],[0,5],[],[0,6],[0,0]]

**Output:** [null,null,0,null,5]

### Explanation:

```
SnapshotArray snapshotArr = new SnapshotArray(3); // set the length to be 3
snapshotArr.set(0,5); // Set array[0] = 5
snapshotArr.snap(); // Take a snapshot, return snap_id = 0
snapshotArr.set(0,6);
snapshotArr.get(0,0); // Get the value of array[0] with snap_id = 0, return
```

## Constraints:

- `1 <= length <= 50000`
- At most `50000` calls will be made to `set` , `snap` , and `get` .
- `0 <= index < length`
- `0 <= snap_id < (the total number of times we call snap())`
- `0 <= val <= 10^9`

```
//Took 2 hrs..approach I was able to get but implementation was tricky got
TLE
//her snap pr array update nahi karna h.... her index ki alag history he h
ogi
//agr kise index pr get wali snap id nahi h to uske just pahle wali bej d
o... qki elemnet change to hua nahi--> this thing I cound not think and got
TLE
```

```
class SnapshotArray {

 HashMap<Integer, Integer> snapArr[]; //array of type hashMap
 int snapId;

 public SnapshotArray(int length) {
 snapArr = new HashMap[length];
 snapId = 0;
 for (int i = 0; i < length; i++) {
 snapArr[i] = new HashMap<Integer, Integer>();
 }
 }

 public void set(int index, int val) {
 HashMap<Integer, Integer> map = snapArr[index];
 map.put(this.snapId, val); //map ka req hai islia dubara snapArr
[index] = map karne ki jarurat nahi
 }

 public int snap() {
 this.snapId++;
 return snapId-1;
 }

 public int get(int index, int snap_id) {
 HashMap<Integer, Integer> map = snapArr[index];

 for (int i = snap_id; i >= 0; i--) {
 if(map.containsKey(i)) return map.get(i);
 }
 return 0;
 }
}

/**
 * Your SnapshotArray object will be instantiated and called as such:
 * SnapshotArray obj = new SnapshotArray(length);
 * obj.set(index,val);
```

```
* int param_2 = obj.snap();
* int param_3 = obj.get(index,snap_id);
*/
```

## 1143. Longest Common Subsequence ↗

Given two strings `text1` and `text2`, return *the length of their longest common subsequence*. If there is no **common subsequence**, return 0.

A **subsequence** of a string is a new string generated from the original string with some characters (can be none) deleted without changing the relative order of the remaining characters.

- For example, "ace" is a subsequence of "abcde".

A **common subsequence** of two strings is a subsequence that is common to both strings.

### Example 1:

**Input:** `text1 = "abcde"`, `text2 = "ace"`

**Output:** 3

**Explanation:** The longest common subsequence is "ace" and its length is 3.

### Example 2:

**Input:** `text1 = "abc"`, `text2 = "abc"`

**Output:** 3

**Explanation:** The longest common subsequence is "abc" and its length is 3.

### Example 3:

**Input:** `text1 = "abc"`, `text2 = "def"`

**Output:** 0

**Explanation:** There is no such common subsequence, so the result is 0.

### Constraints:

- $1 \leq \text{text1.length}, \text{text2.length} \leq 1000$
- `text1` and `text2` consist of only lowercase English characters.

```

class Solution {
 public int longestCommonSubsequence(String text1, String text2)
 {
 int text1_len = text1.length();
 int text2_len = text2.length();
 if (text1_len == 0 || text2_len == 0) return 0;

 //char array use karne se runtime kam ho gaya, charAt k comparision
 me
 char[] t1 = text1.toCharArray();
 char[] t2 = text2.toCharArray();

 int dp[][] = new int[text1_len+1][text2_len+1]; //last row and last
 col me 0 he hoga
 //qki.. blank ka kise k sth bhi common subsequence blank he hoga...
 blank ki length to 0 he h

 for (int i = dp.length-2; i >= 0; i--)
 {
 for (int j = dp[0].length-2; j >= 0; j--)
 {
 if (t1[i] == t2[j])
 {
 dp[i][j] = dp[i+1][j+1] +1; //down right dia +1
 }
 else if (t1[i] != t2[j])
 {
 dp[i][j] = Math.max(dp[i][j+1], dp[i+1][j]); //right an
 d down ka max
 }
 }
 }

 return dp[0][0];
 }
}

```

## 1331. Rank Transform of an Array ↗

Given an array of integers `arr`, replace each element with its rank.

The rank represents how large the element is. The rank has the following rules:

- Rank is an integer starting from 1.
- The larger the element, the larger the rank. If two elements are equal, their rank must be the same.
- Rank should be as small as possible.

### Example 1:

**Input:** arr = [40,10,20,30]

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

**Explanation:** 40 is the largest element. 10 is the smallest. 20 is the second

### Example 2:

**Input:** arr = [100,100,100]

**Output:** [1,1,1]

**Explanation:** Same elements share the same rank.

### Example 3:

**Input:** arr = [37,12,28,9,100,56,80,5,12]

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

### Constraints:

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

```

class Solution {
 public int[] arrayRankTransform(int[] arr)
 {
 int[] res = new int[arr.length];
 HashMap<Integer, Integer> map = new HashMap<>();
 int A[] = arr.clone();
 Arrays.sort(A);
 int rank = 0;

 for (int i = 0; i < A.length; i++)
 {
 if (!map.containsKey(A[i]))
 {
 map.put(A[i], ++rank);
 }
 }

 for (int i = 0; i < arr.length; i++)
 {
 res[i] = map.get(arr[i]);
 }
 return res;
 }
}

```

## 1176. Diet Plan Performance ↗

A dieter consumes `calories[i]` calories on the `i`-th day.

Given an integer `k`, for **every** consecutive sequence of `k` days (`calories[i]`, `calories[i+1]`, ..., `calories[i+k-1]` for all  $0 \leq i \leq n-k$ ), they look at  $T$ , the total calories consumed during that sequence of `k` days (`calories[i] + calories[i+1] + ... + calories[i+k-1]`):

- If  $T < \text{lower}$ , they performed poorly on their diet and lose 1 point;
- If  $T > \text{upper}$ , they performed well on their diet and gain 1 point;
- Otherwise, they performed normally and there is no change in points.

Initially, the dieter has zero points. Return the total number of points the dieter has after dieting for `calories.length` days.

Note that the total points can be negative.

### **Example 1:**

**Input:** calories = [1,2,3,4,5], k = 1, lower = 3, upper = 3

**Output:** 0

**Explanation:** Since k = 1, we consider each element of the array separately and calories[0] and calories[1] are less than lower so 2 points are lost. calories[3] and calories[4] are greater than upper so 2 points are gained.

### **Example 2:**

**Input:** calories = [3,2], k = 2, lower = 0, upper = 1

**Output:** 1

**Explanation:** Since k = 2, we consider subarrays of length 2. calories[0] + calories[1] > upper so 1 point is gained.

### **Example 3:**

**Input:** calories = [6,5,0,0], k = 2, lower = 1, upper = 5

**Output:** 0

**Explanation:**

calories[0] + calories[1] > upper so 1 point is gained.

lower <= calories[1] + calories[2] <= upper so no change in points.

calories[2] + calories[3] < lower so 1 point is lost.

### **Constraints:**

- $1 \leq k \leq \text{calories.length} \leq 10^5$
- $0 \leq \text{calories}[i] \leq 20000$
- $0 \leq \text{lower} \leq \text{upper}$

```

class Solution {
 public int dietPlanPerformance(int[] calories, int k, int lower, int upper) {

 int sumOfCal = 0;
 int start = 0;
 int end = 0;
 int totPoints = 0;

 while (end < calories.length){

 sumOfCal += calories[end]; //ye to hamesha he karna h

 if (end - start +1 < k) { //agr abhi window nahi pure hue to bas end badaho
 end++;
 }
 else if (end - start +1 == k) { //ab calculation sb isme he hogi
 if (sumOfCal < lower) {
 totPoints--;
 }
 if (sumOfCal > upper) {
 totPoints++;
 }
 sumOfCal -= calories[start];
 start++;
 end++;
 }
 }
 return totPoints;
 }
}

```

## 1706. Where Will the Ball Fall ↗

You have a 2-D grid of size  $m \times n$  representing a box, and you have  $n$  balls. The box is open on the top and bottom sides.

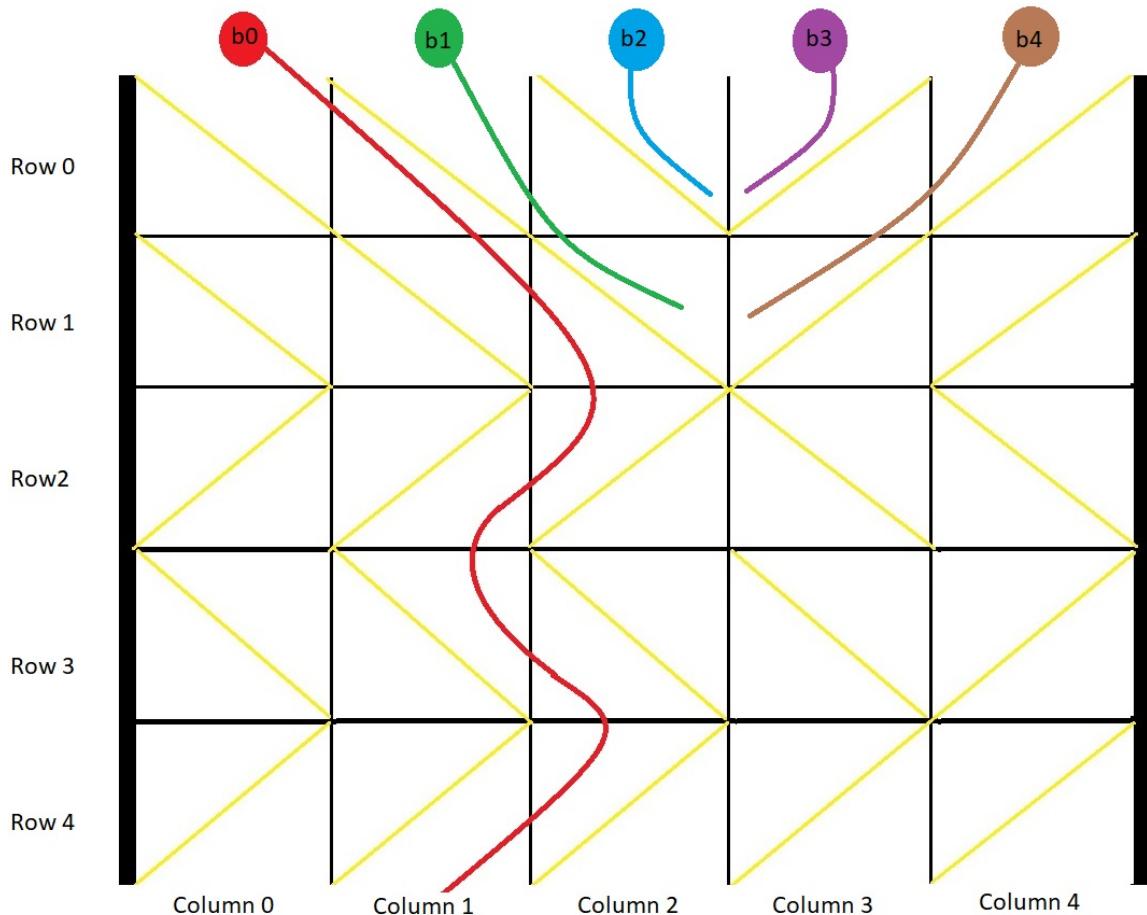
Each cell in the box has a diagonal board spanning two corners of the cell that can redirect a ball to the right or to the left.

- A board that redirects the ball to the right spans the top-left corner to the bottom-right corner and is represented in the grid as `1`.
- A board that redirects the ball to the left spans the top-right corner to the bottom-left corner and is represented in the grid as `-1`.

We drop one ball at the top of each column of the box. Each ball can get stuck in the box or fall out of the bottom. A ball gets stuck if it hits a "V" shaped pattern between two boards or if a board redirects the ball into either wall of the box.

Return an array `answer` of size `n` where `answer[i]` is the column that the ball falls out of at the bottom after dropping the ball from the  $i^{\text{th}}$  column at the top, or `-1` if the ball gets stuck in the box.

### Example 1:



**Input:** `grid = [[1,1,1,-1,-1],[1,1,1,-1,-1],[-1,-1,-1,1,1],[1,1,1,1,-1],[-1,-1]]`  
**Output:** `[1,-1,-1,-1,-1]`

**Explanation:** This example is shown in the photo.

Ball `b0` is dropped at column 0 and falls out of the box at column 1.

Ball `b1` is dropped at column 1 and will get stuck in the box between column 2 and column 3.

Ball `b2` is dropped at column 2 and will get stuck on the box between column 2 and column 3.

Ball `b3` is dropped at column 3 and will get stuck on the box between column 2 and column 3.

Ball `b4` is dropped at column 4 and will get stuck on the box between column 2 and column 3.

## **Example 2:**

**Input:** grid = [[-1]]

**Output:** [-1]

**Explanation:** The ball gets stuck against the left wall.

## **Example 3:**

**Input:** grid = [[1,1,1,1,1,1],[-1,-1,-1,-1,-1,-1],[1,1,1,1,1,1],[-1,-1,-1,-1,-1,-1],

**Output:** [0,1,2,3,4,-1]

## **Constraints:**

- $m == \text{grid.length}$
  - $n == \text{grid}[i].length$
  - $1 \leq m, n \leq 100$
  - $\text{grid}[i][j]$  is 1 or -1.
-

```
/*
```

```
very tricky... best filter based on negative cases
```

```
If 1 hai grid me that means right dia hai--> ball right me jaege(col++) ab
ye dekho neche ja skti hai ya nahi
```

```
Ball kab kab rukege:
```

```
jb next position pr same dia na ho... tb vo neche nahi uter paege
```

```
jb next position boundary k baher chala jae
```

```
*/
```

```
class Solution {
 public int[] findBall(int[][] grid) {
 int[] ans = new int[grid[0].length]; // jitne col utni balls
 Arrays.fill(ans, Integer.MAX_VALUE);

 // her column se ball girani h
 for (int j = 0; j < grid[0].length; j++) {

 int currCol = j; // first row se ball gira
 int i = 0; // starting row for each ball

 while (i < grid.length) {

 // check for all -ve cases 1--> move right....right jana tha
 // but last col hai(diwaar)
 if (currCol == grid[0].length - 1 && grid[i][currCol] == 1) {
 // this is and not or so neche bhi outOfBound check karna hoga
 ans[j] = -1;
 break;
 }
 // -1 hai but qki ye 1st col he hai to left nahi ja skta
 else if (currCol == 0 && grid[i][currCol] == -1) {
 ans[j] = -1;
 break;
 } // 1--> move right lekin right col me agr -1 hai to "V" me
 // block ho jaega
 else if (grid[i][currCol] == 1 && currCol + 1 < grid[0].length && grid[i][currCol + 1] == -1) {
 ans[j] = -1;
 break;
 } // -1--> move left lekin left col me agr 1 hai to "V" me bl
 // ock ho jaega
 else if (grid[i][currCol] == -1 && currCol - 1 >= 0 && grid[i][currCol - 1] == 1) {
 ans[j] = -1;
 break;
 }
 }
 }
 }
}
```

```

 ans[j] = -1;
 break;
 }//block nahi hua tb right dia ya left dia me ball jaege ba
sed on grif value
 else {
 //right
 if (grid[i][currCol] == 1) {
 i++;
 currCol = currCol+1;
 }
 //left
 else if (grid[i][currCol] == -1) {
 i++;
 currCol = currCol-1;
 }
 }
}

if (ans[j] != -1) ans[j] = currCol; //is column se baher aae
}
return ans;
}

}

```

## 1221. Split a String in Balanced Strings ↗

**Balanced** strings are those that have an equal quantity of 'L' and 'R' characters.

Given a **balanced** string  $s$ , split it in the maximum amount of balanced strings.

Return *the maximum amount of split **balanced** strings*.

### Example 1:

**Input:**  $s = "RLRRLLRLRL"$

**Output:** 4

**Explanation:**  $s$  can be split into "RL", "RRLL", "RL", "RL", each substring cor

### Example 2:

**Input:** s = "RLLLLRRRLR"

**Output:** 3

**Explanation:** s can be split into "RL", "LLLRRR", "LR", each substring contain

### Example 3:

**Input:** s = "LLLLRRRR"

**Output:** 1

**Explanation:** s can be split into "LLLLRRRR".

### Example 4:

**Input:** s = "RLRRRLLRLL"

**Output:** 2

**Explanation:** s can be split into "RL", "RRRLLRLL", since each substring conta

### Constraints:

- $1 \leq s.length \leq 1000$
- $s[i]$  is either 'L' or 'R'.
- s is a **balanced** string.

```
class Solution {
 public int balancedStringSplit(String s) {
 int ans = 0;

 int c = 0;

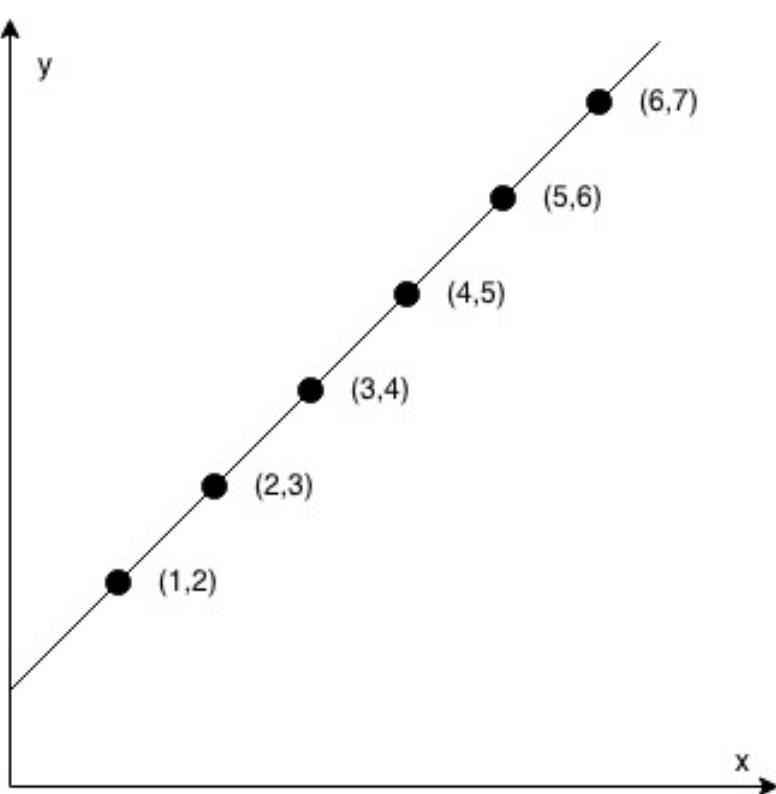
 for (int i = 0; i < s.length(); i++)
 {
 if (s.charAt(i) == 'L')
 {
 c++;
 }
 else if (s.charAt(i) == 'R')
 {
 c--;
 }

 if (c==0) ans++;
 }
 return ans;
 }
}
```

## 1232. Check If It Is a Straight Line ↗

You are given an array `coordinates` , `coordinates[i] = [x, y]` , where `[x, y]` represents the coordinate of a point. Check if these points make a straight line in the XY plane.

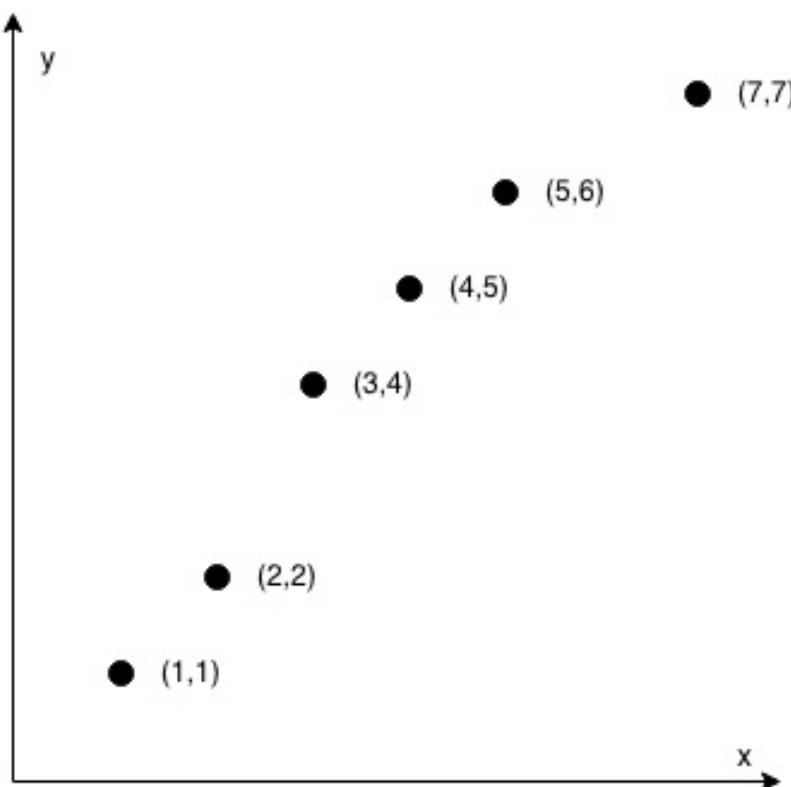
**Example 1:**



**Input:** coordinates = [[1,2],[2,3],[3,4],[4,5],[5,6],[6,7]]

**Output:** true

### Example 2:



**Input:** coordinates = [[1,1],[2,2],[3,4],[4,5],[5,6],[7,7]]

**Output:** false

### **Constraints:**

- $2 \leq \text{coordinates.length} \leq 1000$
  - $\text{coordinates}[i].length == 2$
  - $-10^4 \leq \text{coordinates}[i][0], \text{coordinates}[i][1] \leq 10^4$
  - $\text{coordinates}$  contains no duplicate point.
- 

**Note : Slope of Straight line is always equal.**

**Formula :  $(y_2 - y_1) / (x_2 - x_1)$**

**Type casting to (*double*) was required coz array is of int type and we were getting  $3/5 = 0$  instead of  $0.6$**

**Print 2D Array** *Array : int [][] arr = {{1,2},{2,3},{3,4},{4,5},{5,6},{6,7}};*

```
for(i = 1; i < coordinates.length; i++) { for(j = 0; j < coordinates[i].length; j++) {
 System.out.print(coordinates[i][j] + " "); } System.out.println(); }
```

**Output :**

2 3 3 4 4 5 5 6 6 7

```

class Solution {

 public boolean checkStraightLine(int[][] coordinates) {

 int i = 0, j=0; double slopeTest=0;

 double slope = (double)(coordinates[i+1][j+1] - coordinates[i][j+1])/(coordinates[i+1][j] - coordinates[i][j]);

 for(i = 1; i < coordinates.length-1; i++)
 {
 for(j = 0; j < coordinates[i].length-1; j++)
 {
 slopeTest = (double)(coordinates[i+1][j+1] - coordinates[i][j+1])/(coordinates[i+1][j] - coordinates[i][j]);
 }

 if(slope != slopeTest)
 return false;
 }

 return true;
 }
}

```

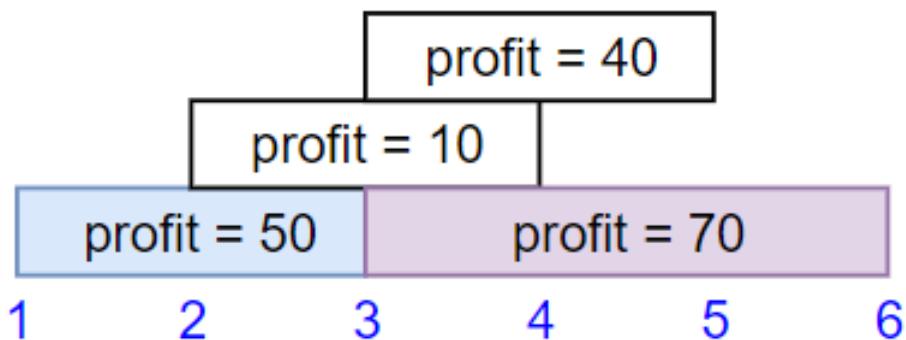
## 1235. Maximum Profit in Job Scheduling ↗ ▾

We have `n` jobs, where every job is scheduled to be done from `startTime[i]` to `endTime[i]`, obtaining a profit of `profit[i]`.

You're given the `startTime`, `endTime` and `profit` arrays, return the maximum profit you can take such that there are no two jobs in the subset with overlapping time range.

If you choose a job that ends at time `X` you will be able to start another job that starts at time `X`.

### Example 1:



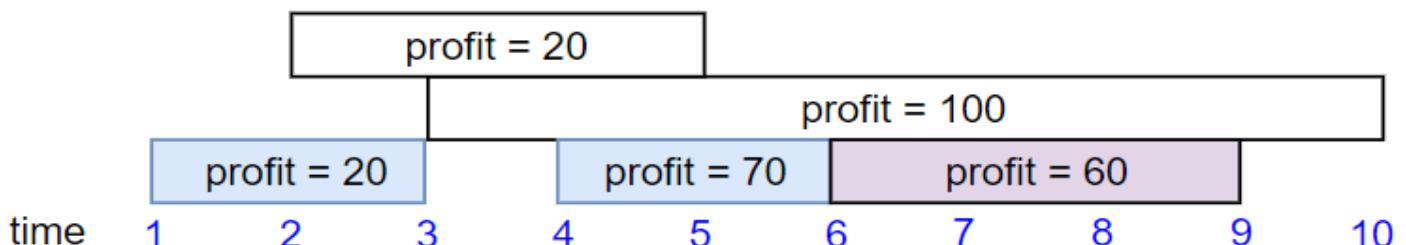
**Input:** startTime = [1,2,3,3], endTime = [3,4,5,6], profit = [50,10,40,70]

**Output:** 120

**Explanation:** The subset chosen is the first and fourth job.

Time range [1-3]+[3-6] , we get profit of 120 = 50 + 70.

### Example 2:



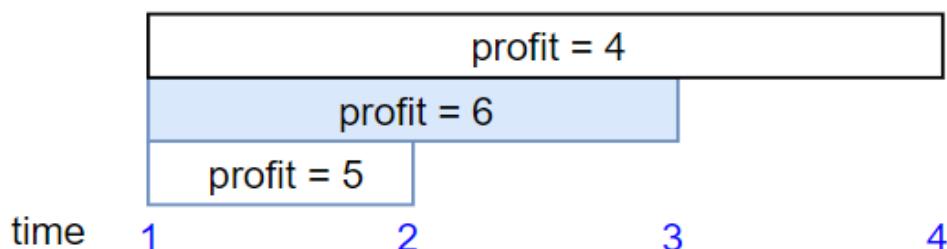
**Input:** startTime = [1,2,3,4,6], endTime = [3,5,10,6,9], profit = [20,20,100,70]

**Output:** 150

**Explanation:** The subset chosen is the first, fourth and fifth job.

Profit obtained 150 = 20 + 70 + 60.

### Example 3:



**Input:** startTime = [1,1,1], endTime = [2,3,4], profit = [5,6,4]

**Output:** 6

### Constraints:

- $1 \leq \text{startTime.length} == \text{endTime.length} == \text{profit.length} \leq 5 * 10^4$
- $1 \leq \text{startTime}[i] < \text{endTime}[i] \leq 10^9$

- $1 \leq \text{profit}[i] \leq 10^4$

```

//copied: floorKey and TreeMap
class Solution {
 private class Job{
 private int startTime;
 private int endTime;
 private int profit;

 public Job(int sT, int eT, int p){
 this.startTime = sT;
 this.endTime = eT;
 this.profit = p;
 }
 }

 public int jobScheduling(int[] startTime, int[] endTime, int[] profit)
 {
 List<Job> jobs = new ArrayList<>();
 for (int i=0; i<startTime.length; i++) {
 jobs.add(new Job(startTime[i],endTime[i], profit[i]));
 }

 Collections.sort(jobs, (a,b) -> (a.endTime - b.endTime));
 // Key => Time, Value => profitTillTime
 TreeMap<Integer, Integer> map = new TreeMap<>();
 int ans = 0;

 for (Job currJob : jobs) {
 Integer entryTillStartTime = map.floorKey(currJob.startTime);
 //The method call returns the greatest key less than or equal to key, or null if there is no such key
 int maxProfitTillStartTime = entryTillStartTime ==null?0:map.get(entryTillStartTime);
 ans = Math.max(ans, maxProfitTillStartTime+currJob.profit);
 map.put(currJob.endTime, ans);
 }
 return ans;
 }
}

```

# 1227. Airplane Seat Assignment Probability

$n$  passengers board an airplane with exactly  $n$  seats. The first passenger has lost the ticket and picks a seat randomly. But after that, the rest of passengers will:

- Take their own seat if it is still available,
- Pick other seats randomly when they find their seat occupied

What is the probability that the  $n$ -th person can get his own seat?

## Example 1:

**Input:**  $n = 1$

**Output:** 1.00000

**Explanation:** The first person can only get the first seat.

## Example 2:

**Input:**  $n = 2$

**Output:** 0.50000

**Explanation:** The second person has a probability of 0.5 to get the second seat.

## Constraints:

- $1 \leq n \leq 10^5$

```
class Solution {
 public double nthPersonGetsNthSeat(int n) {
 //There is always 50% probability
 if (n == 1) return 1;
 else return (double)1/2;
 }
}
```

# 1249. Minimum Remove to Make Valid Parentheses ↗

Given a string  $s$  of ' $($ ' , ' $)$ ' and lowercase English characters.

Your task is to remove the minimum number of parentheses ( $'($  or  $)'$ , in any positions) so that the resulting *parentheses string* is valid and return **any** valid string.

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

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

## Example 1:

**Input:**  $s = \text{"lee(t(c)o)de"}$

**Output:**  $\text{"lee(t(c)o)de"}$

**Explanation:**  $\text{"lee(t(co)de)"}$  ,  $\text{"lee(t(c)ode)"}$  would also be accepted.

## Example 2:

**Input:**  $s = \text{"a)b(c)d"}$

**Output:**  $\text{"ab(c)d"}$

## Example 3:

**Input:**  $s = \text{"})()("}$

**Output:**  $\text{"")"}$

**Explanation:** An empty string is also valid.

## Example 4:

**Input:**  $s = \text{"(a(b(c)d)"}$

**Output:**  $\text{"a(b(c)d)"}$

## Constraints:

- $1 \leq s.length \leq 10^5$
- $s[i]$  is either ' $($ ' , ' $)$ ' , or lowercase English letter .



```

class Solution {
 public String minRemoveToMakeValid(String s) {

 StringBuffer sb = new StringBuffer();
 int open = 0; // to maintain bracket balance
 Stack<Character> st = new Stack<>();

 for (char ch : s.toCharArray()) {

 if (ch == '(') {

 st.push(ch); //first open
 open++;

 }

 else if (ch == ')') {
 if(open > 0) { // matlab open ja chuka h
 st.push(ch);
 open--; // qki iska close bhi gaya
 }
 }

 else {
 st.push(ch);
 }

 }

 // ek extra open gaya h

 while (!st.isEmpty()) {

 char pop = st.pop();
 if (open > 0 && pop == '('){
 open--;
 continue; // dont include this (qki iska close nahi h t
bhi 0 nahi hua iska count

 }

 else {
 sb.append(pop);
 }

 }

 return sb.reverse().toString();
 }
}

```

# 1254. Number of Closed Islands ↗



Given a 2D grid consists of 0s (land) and 1s (water). An *island* is a maximal 4-directionally connected group of 0s and a *closed island* is an island **totally** (all left, top, right, bottom) surrounded by 1s.

Return the number of *closed islands*.

## Example 1:

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |

**Input:** grid = [[1,1,1,1,1,1,1,0],[1,0,0,0,0,1,1,0],[1,0,1,0,1,1,1,0],[1,0,0,0,1,1,1,0]]  
**Output:** 2

## Explanation:

Islands in gray are closed because they are completely surrounded by water (gray).

## Example 2:

|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 |

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

## Example 3:

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

**Output:** 2

**Constraints:**

- $1 \leq \text{grid.length}, \text{grid}[0].length \leq 100$
  - $0 \leq \text{grid}[i][j] \leq 1$
-

```

//island dekho jo corner se touch na kare...use 1 bacha le
class Solution {
 public int closedIsland(int[][] grid) {
 int row = grid.length;
 int col = grid[0].length;
 boolean[][] visited = new boolean[row][col];
 int count = 0;
 for (int i = 1; i < row-1; i++) {
 for (int j = 1; j < col-1; j++) {
 if (!visited[i][j] && grid[i][j] == 0) {
 boolean isClosed = dfs(i, j, grid, visited);
 if (isClosed) {
 count++;
 }
 }
 }
 }
 return count;
 }

 private boolean dfs(int row, int col, int[][] grid, boolean[][] visited){
 if (visited[row][col] || grid[row][col] == 1) return true; //ye condition pahle hogi..varna ans galat ho jaega..is condition ka matalb hai 1 ne bacha lia

 //is condition me hum dhyaan rakh rahai ki corner me jae he nahi is lia index ka error nahi aa raha "<= use kia hai instead of <" & ">= grid.length-1 instead of grid.length"
 if (row <= 0 || col <= 0 || row >= grid.length-1 || col >= grid[0].length-1)
 return false;

 visited[row][col] = true;
 boolean up = dfs(row +1, col, grid, visited);
 boolean down = dfs(row -1, col, grid, visited);
 boolean right = dfs(row, col+1, grid, visited);
 boolean left = dfs(row, col-1, grid, visited);

 if (up && down && right && left) return true;
 return false;
 }
}

```

```

//island dekho jo corner se touch na kare...use 1 bacha le
class Solution {
 public int closedIsland(int[][] grid) {
 int row = grid.length;
 int col = grid[0].length;
 int count = 0;
 for (int i = 1; i < row-1; i++) {
 for (int j = 1; j < col-1; j++) {
 if (grid[i][j] == 0) {
 boolean isClosed = dfs(i, j, grid);
 if (isClosed) {
 count++;
 }
 }
 }
 }
 return count;
 }

 private boolean dfs(int row, int col, int[][] grid){

 if (grid[row][col] == 1) return true; //ye condition pahle hogi..va
 rna ans galat ho jaega..is condition ka matalb hai 1 ne bacha lia

 //is condition me hum dhyaan rakh rahai ki corner me jae he nahi is
 lia index ka error nahi aa raha "<= use kia hai instead of <" & ">= grid.le
 ngth-1 instead of grid.length"
 if (row <= 0 || col <= 0 || row >= grid.length-1 || col >= grid[0].
 length-1)
 return false;

 grid[row][col] = 1;
 boolean up = dfs(row +1, col, grid);
 boolean down = dfs(row -1, col, grid);
 boolean right = dfs(row, col+1, grid);
 boolean left = dfs(row, col-1, grid);

 if (up && down && right && left) return true;
 return false;
 }
}

```

# 1277. Count Square Submatrices with All Ones



Given a  $m * n$  matrix of ones and zeros, return how many **square** submatrices have all ones.

## Example 1:

**Input:** matrix =

```
[
 [0,1,1,1],
 [1,1,1,1],
 [0,1,1,1]
```

]

**Output:** 15

**Explanation:**

There are 10 squares of side 1.

There are 4 squares of side 2.

There is 1 square of side 3.

Total number of squares =  $10 + 4 + 1 = 15$ .

## Example 2:

**Input:** matrix =

```
[
 [1,0,1],
 [1,1,0],
 [1,1,0]
```

]

**Output:** 7

**Explanation:**

There are 6 squares of side 1.

There is 1 square of side 2.

Total number of squares =  $6 + 1 = 7$ .

## Constraints:

- $1 \leq \text{arr.length} \leq 300$
- $1 \leq \text{arr}[0].length \leq 300$
- $0 \leq \text{arr}[i][j] \leq 1$

```

class Solution {
 public int countSquares(int[][][] matrix) {
 int count = 0;
 for (int i = 1; i < matrix.length; i++) {
 for (int j = 1; j < matrix[0].length; j++) {
 if (matrix[i][j] == 0) continue;

 int up = matrix[i-1][j];
 int left = matrix[i][j-1];
 int dia = matrix[i-1][j-1];

 int min = Math.min(up, Math.min(left, dia))+1;
 matrix[i][j] = min;

 }
 }
 // System.out.println(Arrays.deepToString(matrix));
 for (int i = 0; i < matrix.length; i++) {
 for (int j = 0; j < matrix[0].length; j++) {

 count += matrix[i][j];

 }
 }
 return count;
 }
}

```

## 1332. Remove Palindromic Subsequences ↗ ▾

You are given a string  $s$  consisting **only** of letters ' $a$ ' and ' $b$ '. In a single step you can remove one **palindromic subsequence** from  $s$ .

Return *the minimum number of steps to make the given string empty.*

A string is a **subsequence** of a given string if it is generated by deleting some characters of a given string without changing its order. Note that a subsequence does **not** necessarily need to be contiguous.

A string is called **palindrome** if it reads the same backward as well as forward.

### **Example 1:**

**Input:** s = "ababa"

**Output:** 1

**Explanation:** s is already a palindrome, so its entirety can be removed in a single step.

### **Example 2:**

**Input:** s = "abb"

**Output:** 2

**Explanation:** "abb" -> "bb" -> "".

Remove palindromic subsequence "a" then "bb".

### **Example 3:**

**Input:** s = "baabb"

**Output:** 2

**Explanation:** "baabb" -> "b" -> "".

Remove palindromic subsequence "baab" then "b".

### **Constraints:**

- $1 \leq s.length \leq 1000$
- $s[i]$  is either 'a' or 'b'.

```

class Solution {
 //String k pass a & b he hai
 //3 case e possible hai: empty pr 0.... pallindrome pr 1... else hum pa
 hle sb a hataege fir b : so and will be 2
 public int removePalindromeSub(String s) {

 if (s.isBlank()) return 0;

 else if (isPalindrome(s)) return 1;

 else return 2;
 }

 public boolean isPalindrome(String s)
 {
 int i = 0, j = s.length()-1;

 while (i < j)
 {
 if (s.charAt(i)!= s.charAt(j)) return false;

 i++;
 j--;
 }

 return true;
 }
}

```

## 1346. Check If N and Its Double Exist ↗

Given an array `arr` of integers, check if there exists two integers `N` and `M` such that `N` is the double of `M` (i.e.  $N = 2 * M$ ).

More formally check if there exists two indices `i` and `j` such that:

- $i \neq j$
- $0 \leq i, j < arr.length$
- $arr[i] == 2 * arr[j]$

**Example 1:**

**Input:** arr = [10,2,5,3]

**Output:** true

**Explanation:** N = 10 is the double of M = 5, that is,  $10 = 2 * 5$ .

### Example 2:

**Input:** arr = [7,1,14,11]

**Output:** true

**Explanation:** N = 14 is the double of M = 7, that is,  $14 = 2 * 7$ .

### Example 3:

**Input:** arr = [3,1,7,11]

**Output:** false

**Explanation:** In this case does not exist N and M, such that  $N = 2 * M$ .

### Constraints:

- $2 \leq \text{arr.length} \leq 500$
- $-10^3 \leq \text{arr}[i] \leq 10^3$

```
class Solution { public boolean checkIfExist(int[] arr) {
 Set set = new HashSet();
 for(int i = 0; i < arr.length; i++) { //imp condition if (set.contains(arr[i] * 2) || (arr[i] % 2 == 0 &&
 set.contains(arr[i]/2))) { return true; } else { set.add(arr[i]); } }
 return false;
}
```

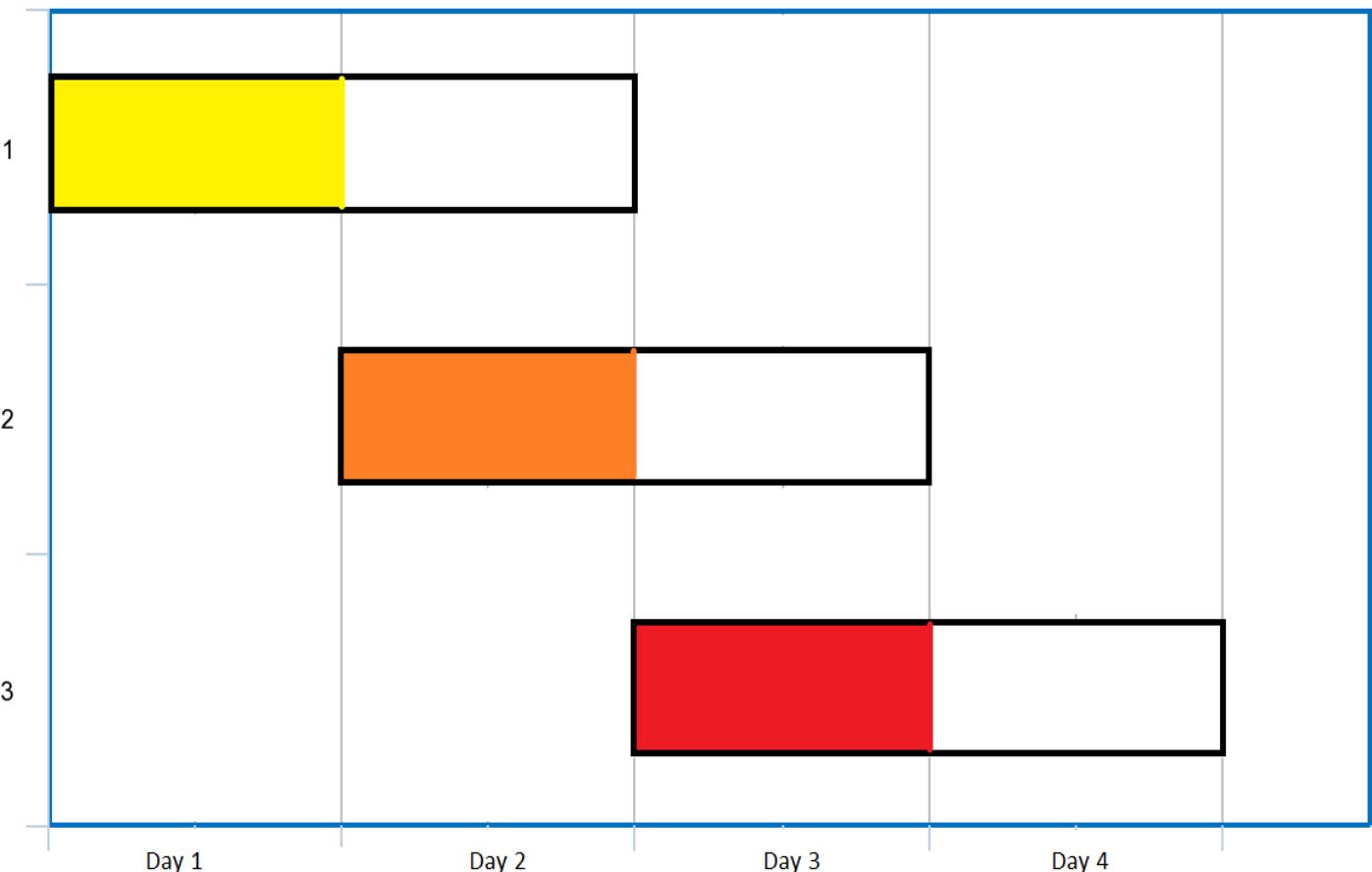
## 1353. Maximum Number of Events That Can Be Attended ↗

Given an array of events where  $\text{events}[i] = [\text{startDay}_i, \text{endDay}_i]$ . Every event  $i$  starts at  $\text{startDay}_i$  and ends at  $\text{endDay}_i$ .

You can attend an event  $i$  at any day  $d$  where  $\text{startTime}_i \leq d \leq \text{endTime}_i$ . Notice that you can only attend one event at any time  $d$ .

Return *the maximum number of events* you can attend.

### Example 1:



**Input:** events = [[1,2],[2,3],[3,4]]

**Output:** 3

**Explanation:** You can attend all the three events.

One way to attend them all is as shown.

Attend the first event on day 1.

Attend the second event on day 2.

Attend the third event on day 3.

### Example 2:

**Input:** events= [[1,2],[2,3],[3,4],[1,2]]

**Output:** 4

### Example 3:

**Input:** events = [[1,4],[4,4],[2,2],[3,4],[1,1]]

**Output:** 4

### Example 4:

**Input:** events = [[1,100000]]

**Output:** 1

### Example 5:

**Input:** events = [[1,1],[1,2],[1,3],[1,4],[1,5],[1,6],[1,7]]

**Output:** 7

### Constraints:

- $1 \leq \text{events.length} \leq 10^5$
  - $\text{events}[i].length == 2$
  - $1 \leq \text{startDay}_i \leq \text{endDay}_i \leq 10^5$
-

```

/*baate jo dhyaan rakha hai..ye normal merge ki jaisa nahi hai
1) loop chalao day wala(max last event day nikaal kar)
2) pahle unko nikalo jo ab nahi ho skte..matlab current day k pahle he unka
end day tha
3) fir unko add karo jo aaj attend kia ja skta h
4) sbse chota wala attend karo hume jyada se jyada attend karna h
*/

```

```

class Solution {
 public int maxEvents(int[][] events) {

 Arrays.sort(events, (a,b) -> a[0] - b[0]); //based on start day
 PriorityQueue<int[]> pq = new PriorityQueue<>((a,b) -> a[1]-b
[1]); //end day wala uper

 int lastEvent = 0;
 int totEvents = 0;
 for (int[] event : events) {
 int end = event[1];
 lastEvent = Math.max(lastEvent, end);
 }
 int currIndex = 0;
 for (int i = 1; i <= lastEvent; i++) { //day loop

 while (!pq.isEmpty()) { //pahle un sb event ko nikaal do jinhe
ab attend nahi kar skte
 int[] currEvent = pq.peek();
 int currEnd = currEvent[1]; //peche gaya ab to
 if (currEnd < i) { //matlab ab attend nahi kar skte
 pq.poll();
 }
 else {
 break; //or jb hume vo event mila jo attend kar skte ha
i then this loop should break
 }
 }

 while (currIndex < events.length && events[currIndex][0] == i)
{ //ab unko daal do jinko aaj attend kar ste h
 pq.add(events[currIndex]);
 currIndex++;
 }
 if (!pq.isEmpty()) {
 pq.poll(); //attended the even which is of smallest duratio
n
 totEvents++;
 }
 }
 }
}

```

```
 }
}

return totEvents;
}

}
```

## 1354. Construct Target Array With Multiple Sums



You are given an array `target` of  $n$  integers. From a starting array `arr` consisting of  $n$  1's, you may perform the following procedure :

- let  $x$  be the sum of all elements currently in your array.
- choose index  $i$ , such that  $0 \leq i < n$  and set the value of `arr` at index  $i$  to  $x$ .
- You may repeat this procedure as many times as needed.

Return `true` if it is possible to construct the `target` array from `arr`, otherwise, return `false`.

### Example 1:

**Input:** target = [9,3,5]

**Output:** true

**Explanation:** Start with arr = [1, 1, 1]

[1, 1, 1], sum = 3 choose index 1

[1, 3, 1], sum = 5 choose index 2

[1, 3, 5], sum = 9 choose index 0

[9, 3, 5] Done

### Example 2:

**Input:** target = [1,1,1,2]

**Output:** false

**Explanation:** Impossible to create target array from [1,1,1,1].

### Example 3:

**Input:** target = [8,5]

**Output:** true

## Constraints:

- $n == \text{target.length}$
- $1 \leq n \leq 5 * 10^4$
- $1 \leq \text{target}[i] \leq 10^9$

```
class Solution {
 public boolean isPossible(int[] target) {

 long totSum = 0l;
 PriorityQueue<Integer> pq = new PriorityQueue<>(Collections.reverse
Order()));

 for (int i : target)
 {
 pq.add(i);
 totSum += i;
 }

 while (!pq.isEmpty())
 {
 int maxEle = pq.poll(); //max element

 long rem = totSum - maxEle; //remaining
 if (maxEle == 1 || rem == 1) return true; //max element is 1 no

w

 if (maxEle < rem || rem == 0) return false;

 int updatedMax = (int)(maxEle%rem);

 if (updatedMax == 0) return false;

 pq.add(updatedMax);

 totSum = updatedMax+rem;
 }

 return false;
 }
}
```

# 1365. How Many Numbers Are Smaller Than the Current Number ↗



Given the array `nums`, for each `nums[i]` find out how many numbers in the array are smaller than it. That is, for each `nums[i]` you have to count the number of valid `j`'s such that `j != i` **and** `nums[j] < nums[i]`.

Return the answer in an array.

## Example 1:

**Input:** `nums = [8,1,2,2,3]`

**Output:** `[4,0,1,1,3]`

### Explanation:

For `nums[0]=8` there exist four smaller numbers than it (1, 2, 2 and 3).

For `nums[1]=1` does not exist any smaller number than it.

For `nums[2]=2` there exist one smaller number than it (1).

For `nums[3]=2` there exist one smaller number than it (1).

For `nums[4]=3` there exist three smaller numbers than it (1, 2 and 2).

## Example 2:

**Input:** `nums = [6,5,4,8]`

**Output:** `[2,1,0,3]`

## Example 3:

**Input:** `nums = [7,7,7,7]`

**Output:** `[0,0,0,0]`

## Constraints:

- $2 \leq \text{nums.length} \leq 500$
- $0 \leq \text{nums}[i] \leq 100$

```

class Solution {
 public int[] smallerNumbersThanCurrent(int[] nums) {

 int[] clone = nums.clone();
 Arrays.sort(clone);

 int[] ans = new int[nums.length];
 int count = 0;

 for (int i = 0; i < nums.length; i++)
 {
 for (int j = 0; j < nums.length; j++)
 {
 //System.out.print(" i " + i + " j " + j);
 if (clone[j] < nums[i]) count++;
 else break;
 }
 ans[i] = count;
 count = 0;
 }
 return ans;
 }
}

```

## 1376. Time Needed to Inform All Employees ↗

A company has  $n$  employees with a unique ID for each employee from  $0$  to  $n - 1$ . The head of the company is the one with `headID`.

Each employee has one direct manager given in the `manager` array where `manager[i]` is the direct manager of the  $i$ -th employee, `manager[headID] = -1`. Also, it is guaranteed that the subordination relationships have a tree structure.

The head of the company wants to inform all the company employees of an urgent piece of news. He will inform his direct subordinates, and they will inform their subordinates, and so on until all employees know about the urgent news.

The  $i$ -th employee needs `informTime[i]` minutes to inform all of his direct subordinates (i.e., After `informTime[i]` minutes, all his direct subordinates can start spreading the news).

Return *the number of minutes* needed to inform all the employees about the urgent news.

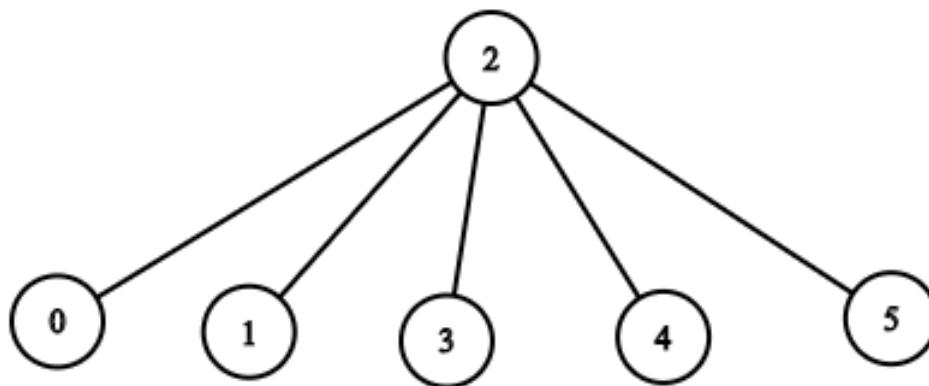
### Example 1:

**Input:** n = 1, headID = 0, manager = [-1], informTime = [0]

**Output:** 0

**Explanation:** The head of the company is the only employee in the company.

### Example 2:

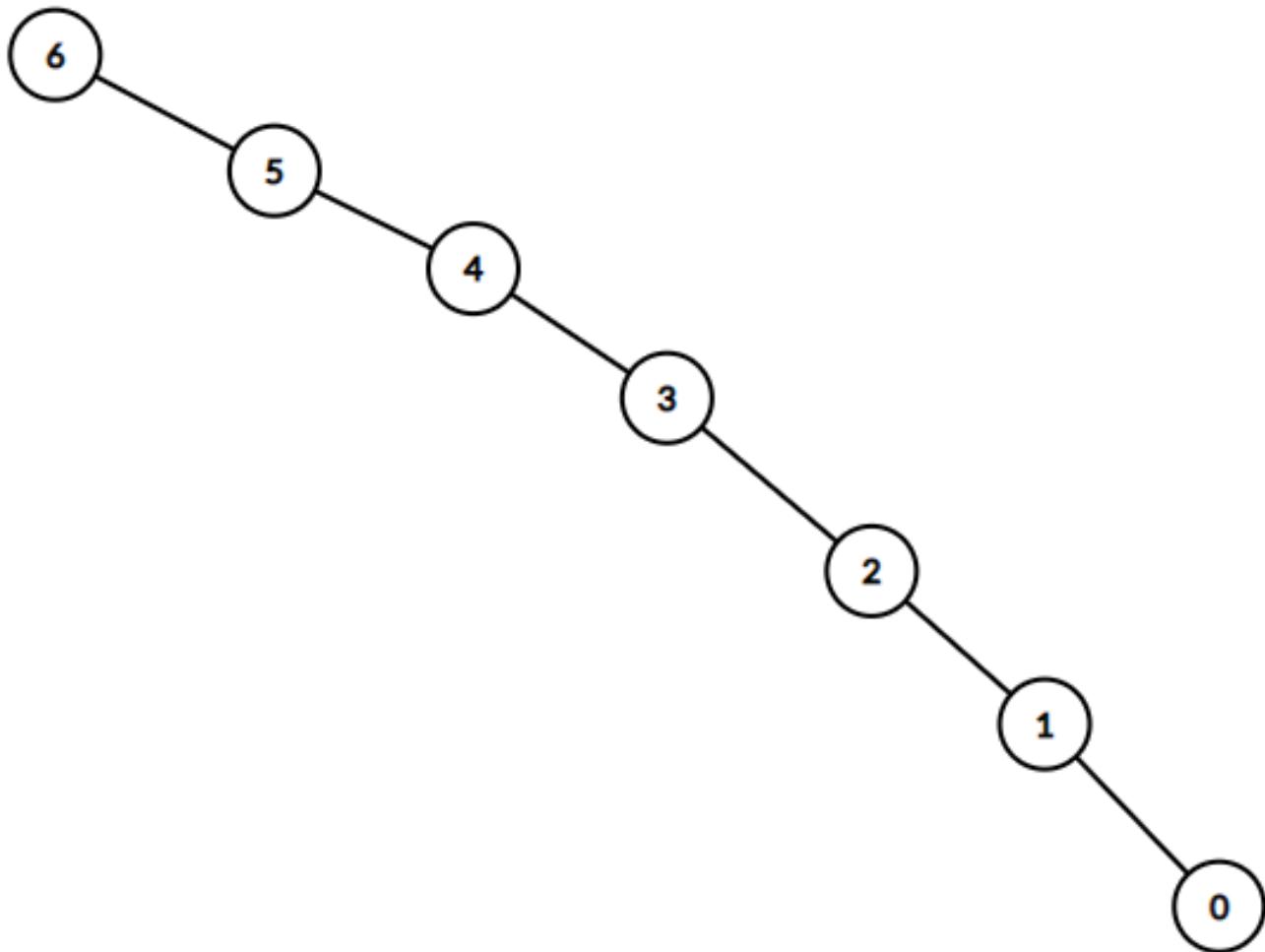


**Input:** n = 6, headID = 2, manager = [2,2,-1,2,2,2], informTime = [0,0,1,0,0,0]

**Output:** 1

**Explanation:** The head of the company with id = 2 is the direct manager of all. The tree structure of the employees in the company is shown.

### Example 3:



**Input:** n = 7, headID = 6, manager = [1,2,3,4,5,6,-1], informTime = [0,6,5,4,3,2,1]  
**Output:** 21

**Explanation:** The head has id = 6. He will inform employee with id = 5 in 1 minute. The employee with id = 5 will inform the employee with id = 4 in 2 minutes. The employee with id = 4 will inform the employee with id = 3 in 3 minutes. The employee with id = 3 will inform the employee with id = 2 in 4 minutes. The employee with id = 2 will inform the employee with id = 1 in 5 minutes. The employee with id = 1 will inform the employee with id = 0 in 6 minutes. Needed time = 1 + 2 + 3 + 4 + 5 + 6 = 21.

#### Example 4:

**Input:** n = 15, headID = 0, manager = [-1,0,0,1,1,2,2,3,3,4,4,5,5,6,6], informTime = [0,0,0,1,1,2,2,3,3,4,4,5,5,6,6]  
**Output:** 3

**Explanation:** The first minute the head will inform employees 1 and 2. The second minute they will inform employees 3, 4, 5 and 6. The third minute they will inform the rest of employees.

#### Example 5:

**Input:** n = 4, headID = 2, manager = [3,3,-1,2], informTime = [0,0,162,914]

**Output:** 1076

### Constraints:

- $1 \leq n \leq 10^5$
  - $0 \leq \text{headID} < n$
  - $\text{manager.length} == n$
  - $0 \leq \text{manager}[i] < n$
  - $\text{manager}[\text{headID}] == -1$
  - $\text{informTime.length} == n$
  - $0 \leq \text{informTime}[i] \leq 1000$
  - $\text{informTime}[i] == 0$  if employee i has no subordinates.
  - It is **guaranteed** that all the employees can be informed.
-

```

//time le raha h
//kafe time laga mujheise solve karne me ...BFS to lagana nahi DFS he shi h
class Solution {
 public int numOfMinutes(int n, int headID, int[] manager, int[] informTime) {

 // HashMap<Integer, Integer> inform = new HashMap<>(); //id and info
 rm time
 // System.out.print(informTime.length);
 // for (int i = 0; i < informTime.length; i++) {
 // inform.put(i, informTime[i]);
 // }
 // System.out.print(inform);
 List<List<Integer>> graph = new ArrayList<>();

 for (int i = 0; i < n; i++) {
 graph.add(new ArrayList<Integer>());
 }

 for (int i = 0; i < n; i++) {
 if (manager[i] != -1)
 graph.get(manager[i]).add(i); //is manager k under kon kon emp
 hai
 // inform.put(i, informTime[i]);
 } //graph ready

 int[] totalTime = new int[1];
 dfs(headID, graph, informTime, 0, totalTime);
 return totalTime[0];
 }

 void dfs(int headID, List<List<Integer>> graph, int[] informTime, int t
ime, int[] totalTime){

 int timeReq = informTime[headID];

 for (int i : graph.get(headID)) { //for each subord

 dfs(i, graph, informTime, time + timeReq, totalTime);

 }

 totalTime[0] = Math.max(time, totalTime[0]);
 }
}

```

# 1383. Maximum Performance of a Team



You are given two integers  $n$  and  $k$  and two integer arrays `speed` and `efficiency` both of length  $n$ . There are  $n$  engineers numbered from 1 to  $n$ . `speed[i]` and `efficiency[i]` represent the speed and efficiency of the  $i^{\text{th}}$  engineer respectively.

Choose **at most**  $k$  different engineers out of the  $n$  engineers to form a team with the maximum **performance**.

The performance of a team is the sum of their engineers' speeds multiplied by the minimum efficiency among their engineers.

Return *the maximum performance of this team*. Since the answer can be a huge number, return it **modulo**  $10^9 + 7$ .

## Example 1:

**Input:**  $n = 6$ , `speed` = [2,10,3,1,5,8], `efficiency` = [5,4,3,9,7,2],  $k = 2$

**Output:** 60

**Explanation:**

We have the maximum performance of the team by selecting engineer 2 (with spe

## Example 2:

**Input:**  $n = 6$ , `speed` = [2,10,3,1,5,8], `efficiency` = [5,4,3,9,7,2],  $k = 3$

**Output:** 68

**Explanation:**

This is the same example as the first but  $k = 3$ . We can select engineer 1, er

## Example 3:

**Input:**  $n = 6$ , `speed` = [2,10,3,1,5,8], `efficiency` = [5,4,3,9,7,2],  $k = 4$

**Output:** 72

### **Constraints:**

- $1 \leq k \leq n \leq 10^5$
  - $\text{speed.length} == n$
  - $\text{efficiency.length} == n$
  - $1 \leq \text{speed}[i] \leq 10^5$
  - $1 \leq \text{efficiency}[i] \leq 10^8$
-

```

//TC: O(nlogn)
//SC: O(n)
//question easy hai jyda combination vegera na sochne lagna bs
class Solution {

 class Engineer {
 int speed;
 int effeciency;

 Engineer(int speed, int effeciency) {
 this.speed = speed;
 this.effeciency = effeciency;
 }
 }

 public int maxPerformance(int n, int[] speed, int[] efficiency, int k)
 {

 Engineer[] engs = new Engineer[n];
 long maxPerformance = 0;
 long currPer = 0;

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

 Engineer eng = new Engineer(speed[i], efficiency[i]);
 engs[i] = eng;
 }
 Arrays.sort(engs, (a,b) -> b.effeciency - a.effeciency); //sbse jya
da jiski efficiency

 PriorityQueue<Engineer> pq = new PriorityQueue<>((a,b) -> a.speed-
b.speed); //jiski speed sbse kam hogi vo uper rahaega and use he baher kar
denge
 long currSpeed = 0;
 for (int i = 0; i < n; i++) {

 if (pq.size() == k) {
 Engineer removed = pq.remove(); //jiski speed kam thi..qki
speed add ho re
 currSpeed = currSpeed - removed.speed; //new speed
 }

 pq.add(engs[i]); //add karo
 currSpeed = currSpeed + (long)engs[i].speed; //qki speed ham
esha add hogi
 currPer = (long)engs[i].effeciency * currSpeed;
 maxPerformance = Math.max(maxPerformance, currPer);
 }
 }
}

```

```

 }
 return (int)(maxPerformance % (long)(1e9 + 7));
}

```

## 1431. Kids With the Greatest Number of Candies ↗

There are  $n$  kids with candies. You are given an integer array `candies`, where each `candies[i]` represents the number of candies the  $i^{\text{th}}$  kid has, and an integer `extraCandies`, denoting the number of extra candies that you have.

Return a boolean array `result` of length  $n$ , where `result[i]` is true if, after giving the  $i^{\text{th}}$  kid all the `extraCandies`, they will have the **greatest** number of candies among all the kids, or false otherwise.

Note that **multiple** kids can have the **greatest** number of candies.

### Example 1:

**Input:** `candies = [2,3,5,1,3]`, `extraCandies = 3`

**Output:** `[true,true,true,false,true]`

**Explanation:** If you give all `extraCandies` to:

- Kid 1, they will have  $2 + 3 = 5$  candies, which is the greatest among the kids.
- Kid 2, they will have  $3 + 3 = 6$  candies, which is the greatest among the kids.
- Kid 3, they will have  $5 + 3 = 8$  candies, which is the greatest among the kids.
  
- Kid 4, they will have  $1 + 3 = 4$  candies, which is not the greatest among the kids.
- Kid 5, they will have  $3 + 3 = 6$  candies, which is the greatest among the kids.

### Example 2:

**Input:** `candies = [4,2,1,1,2]`, `extraCandies = 1`

**Output:** `[true,false,false,false,false]`

**Explanation:** There is only 1 extra candy.

Kid 1 will always have the greatest number of candies, even if a different kid

### Example 3:

**Input:** candies = [12,1,12], extraCandies = 10

**Output:** [true,false,true]

### Constraints:

- $n == \text{candies.length}$
- $2 \leq n \leq 100$
- $1 \leq \text{candies}[i] \leq 100$
- $1 \leq \text{extraCandies} \leq 50$

```
class Solution
{
 public List<Boolean> kidsWithCandies(int[] candies, int extraCandies)
 {
 int max = maxCandies(candies);
 List<Boolean> rs = new ArrayList<>();

 for (int i = 0; i < candies.length; i++)
 {
 if (candies[i] + extraCandies >= max) rs.add(true); //agr sb, y
 a ek do extra candy dene se max k baraber ho ja ra to true
 else rs.add(false);
 }
 return rs;
 }

 //find max candies one child is having
 public int maxCandies(int[] candies)
 {
 int max = Integer.MIN_VALUE;

 for (int i = 0; i < candies.length; i++)
 {
 max = Math.max(max, candies[i]);
 }
 return max;
 }
}
```

# 1423. Maximum Points You Can Obtain from Cards ↗



There are several cards **arranged in a row**, and each card has an associated number of points. The points are given in the integer array `cardPoints`.

In one step, you can take one card from the beginning or from the end of the row. You have to take exactly `k` cards.

Your score is the sum of the points of the cards you have taken.

Given the integer array `cardPoints` and the integer `k`, return the *maximum score* you can obtain.

## Example 1:

**Input:** `cardPoints = [1,2,3,4,5,6,1]`, `k = 3`

**Output:** 12

**Explanation:** After the first step, your score will always be 1. However, choc

## Example 2:

**Input:** `cardPoints = [2,2,2]`, `k = 2`

**Output:** 4

**Explanation:** Regardless of which two cards you take, your score will always b

## Example 3:

**Input:** `cardPoints = [9,7,7,9,7,7,9]`, `k = 7`

**Output:** 55

**Explanation:** You have to take all the cards. Your score is the sum of points

## Example 4:

**Input:** `cardPoints = [1,1000,1]`, `k = 1`

**Output:** 1

**Explanation:** You cannot take the card in the middle. Your best score is 1.

### **Example 5:**

**Input:** cardPoints = [1,79,80,1,1,1,200,1], k = 3

**Output:** 202

### **Constraints:**

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

```

class Solution {
 public int maxScore(int[] cardPoints, int k) {

 int length = cardPoints.length;
 int remEle = length-k; //remainingElements

 //pink remainingElements with minium sum
 int minsum = Integer.MAX_VALUE;
 int totSum = 0;
 int currSum = 0;
 // int picked = 0;
 int start = 0;
 for (int i = 0; i < length; i++) {

 totSum += cardPoints[i];
 currSum += cardPoints[i];
 // picked++;

 if (i-start +1 == remEle) {
 minsum = Math.min(minsum, currSum);
 currSum -= cardPoints[start];
 start++;
 // picked--;
 }
 }
 if (k == cardPoints.length) return totSum;

 return totSum-minsum;
 }
}

```

## 1424. Diagonal Traverse II ↗

Given a list of lists of integers, `nums`, return all elements of `nums` in diagonal order as shown in the below images.

**Example 1:**

|   |   |   |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |

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

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

### Example 2:

|    |    |    |    |    |
|----|----|----|----|----|
| 1  | 2  | 3  | 4  | 5  |
| 6  | 7  |    |    |    |
| 8  |    |    |    |    |
| 9  | 10 | 11 |    |    |
| 12 | 13 | 14 | 15 | 16 |

**Input:** nums = [[1,2,3,4,5],[6,7],[8],[9,10,11],[12,13,14,15,16]]

**Output:** [1,6,2,8,7,3,9,4,12,10,5,13,11,14,15,16]

### Example 3:

**Input:** nums = [[1,2,3],[4],[5,6,7],[8],[9,10,11]]

**Output:** [1,4,2,5,3,8,6,9,7,10,11]

### Example 4:

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

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

### Constraints:

- $1 \leq \text{nums.length} \leq 10^5$
- $1 \leq \text{nums}[i].length \leq 10^5$
- $1 \leq \text{nums}[i][j] \leq 10^9$
- There at most  $10^5$  elements in nums .

```
class Solution {
 public int[] findDiagonalOrder(List<List<Integer>> nums) {

 Map<Integer, ArrayList<Integer>> map = new TreeMap<>();
 int row = 0;
 int sizeOfArray = 0;
 for (List<Integer> innerList : nums) { //get innerList one by one

 int size = innerList.size();

 for (int i = 0; i < size ; i++) {
 int key = row+i;
 ArrayList<Integer> al;
 if (map.containsKey(key)) {
 al = map.get(key);
 al.add(innerList.get(i));
 map.put(key, al);
 }
 else {
 al = new ArrayList<>();
 al.add(innerList.get(i));
 map.put(key, al);
 }
 sizeOfArray++;
 }
 row++;
 }

 // Set<Integer> s = map.keySet();
 int[] ans = new int[sizeOfArray];
 int index = 0;
 for (int key : map.keySet()) {
 ArrayList<Integer> al = map.get(key);
 for (int i = al.size()-1; i >= 0; i--){
 ans[index] = al.get(i);
 index++;
 }
 }
 return ans;
 }
}
```

You are given the array `paths`, where `paths[i] = [cityAi, cityBi]` means there exists a direct path going from `cityAi` to `cityBi`. *Return the destination city, that is, the city without any path outgoing to another city.*

It is guaranteed that the graph of paths forms a line without any loop, therefore, there will be exactly one destination city.

### Example 1:

**Input:** `paths = [["London", "New York"], ["New York", "Lima"], ["Lima", "Sao Paulo"]]`

**Output:** "Sao Paulo"

**Explanation:** Starting at "London" city you will reach "Sao Paulo" city which

### Example 2:

**Input:** `paths = [["B", "C"], ["D", "B"], ["C", "A"]]`

**Output:** "A"

**Explanation:** All possible trips are:

"D" -> "B" -> "C" -> "A".

"B" -> "C" -> "A".

"C" -> "A".

"A".

Clearly the destination city is "A".

### Example 3:

**Input:** `paths = [["A", "Z"]]`

**Output:** "Z"

### Constraints:

- $1 \leq \text{paths.length} \leq 100$
- $\text{paths}[i].length == 2$
- $1 \leq \text{cityA}_i.\text{length}, \text{cityB}_i.\text{length} \leq 10$
- $\text{cityA}_i \neq \text{cityB}_i$
- All strings consist of lowercase and uppercase English letters and the space character.

```

class Solution {
 public String destCity(List<List<String>> paths)
 {
 HashSet<String> set = new HashSet<>();

 for (List<String> oList : paths) //outer list se ek list lia
 {
 String start = oList.get(0); //sb he 'start'/'source' set me da
al do
 set.add(start);
 }

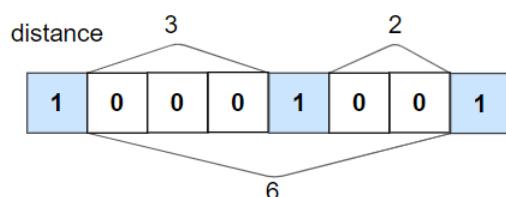
 for (List<String> oList : paths) //outer list se ek list lia
 {
 String destination = oList.get(1);
 //destination kise ka starting ni hoga islia set me nahi milega
 if (!set.contains(destination)) return destination;
 }
 return null;
 }
}

```

## 1437. Check If All 1's Are at Least Length K Places Away

Given an array `nums` of 0s and 1s and an integer `k`, return `True` if all 1's are at least `k` places away from each other, otherwise return `False`.

### Example 1:

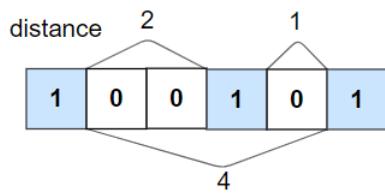


**Input:** `nums = [1,0,0,0,1,0,0,1]`, `k = 2`

**Output:** `true`

**Explanation:** Each of the 1s are at least 2 places away from each other.

### Example 2:



**Input:** nums = [1,0,0,1,0,1], k = 2

**Output:** false

**Explanation:** The second 1 and third 1 are only one apart from each other.

### Example 3:

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

**Output:** true

### Example 4:

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

**Output:** true

### Constraints:

- $1 \leq \text{nums.length} \leq 10^5$
- $0 \leq k \leq \text{nums.length}$
- $\text{nums}[i]$  is 0 or 1

```

class Solution {
 public boolean kLengthApart(int[] nums, int k)
 {
 int count = 0;
 boolean flag = false;
 for (int i = 0; i < nums.length; i++)
 {
 if (nums[i] == 0) count++;
 else
 {
 if (flag && count < k) return false;
 count = 0;
 flag = true; //first time wala 1 ko avoide karne k lia
 }
 }
 return true;
 }
}

```

## 1470. Shuffle the Array ↗

Given the array `nums` consisting of  $2n$  elements in the form  $[x_1, x_2, \dots, x_n, y_1, y_2, \dots, y_n]$ .

*Return the array in the form  $[x_1, y_1, x_2, y_2, \dots, x_n, y_n]$ .*

### Example 1:

**Input:** `nums = [2,5,1,3,4,7], n = 3`

**Output:** `[2,3,5,4,1,7]`

**Explanation:** Since  $x_1=2$ ,  $x_2=5$ ,  $x_3=1$ ,  $y_1=3$ ,  $y_2=4$ ,  $y_3=7$  then the answer is `[2,3,`

### Example 2:

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

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

### Example 3:

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

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

### Constraints:

- $1 \leq n \leq 500$
- $\text{nums.length} == 2n$
- $1 \leq \text{nums}[i] \leq 10^3$

```
class Solution {
 public int[] shuffle(int[] nums, int n) {

 int j = n;
 int k = 0;
 int result[] = new int[nums.length];
 for (int i = 0; i < n; i++)
 {
 result[k] = nums[i];
 result[k+1] = nums[j];
 j++;k= k+2;
 }
 return result;
 }
}
```

## 1480. Running Sum of 1d Array ↗

Given an array `nums`. We define a running sum of an array as `runningSum[i] = sum(nums[0]...nums[i])`.

Return the running sum of `nums`.

### Example 1:

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

**Output:** [1,3,6,10]

**Explanation:** Running sum is obtained as follows: [1, 1+2, 1+2+3, 1+2+3+4].

## Example 2:

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

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

**Explanation:** Running sum is obtained as follows: [1, 1+1, 1+1+1, 1+1+1+1, 1+1+1+1+1]

## Example 3:

**Input:** nums = [3,1,2,10,1]

**Output:** [3,4,6,16,17]

## Constraints:

- $1 \leq \text{nums.length} \leq 1000$
- $-10^6 \leq \text{nums}[i] \leq 10^6$

```
class Solution {
 public int[] runningSum(int[] nums) {

 int sum = nums[0];

 for (int i = 1; i < nums.length; i++) {
 nums[i] += sum;
 sum = nums[i];
 }
 return nums;

 }
}
```

```

class Solution {
 public int[] runningSum(int[] nums)
 {
 int sum = 0;
 int ans[] = new int[nums.length];
 for (int i = 0; i < nums.length; i++)
 {
 sum = sum + nums[i];
 ans[i] = sum;
 }
 return ans;
 }
}

```

## 1509. Minimum Difference Between Largest and Smallest Value in Three Moves ↗ ▾

Given an array `nums`, you are allowed to choose one element of `nums` and change it by any value in one move.

Return the minimum difference between the largest and smallest value of `nums` after performing at most 3 moves.

### Example 1:

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

**Output:** 0

**Explanation:** Change the array `[5,3,2,4]` to `[2,2,2,2]`.

The difference between the maximum and minimum is  $2-2 = 0$ .

### Example 2:

**Input:** `nums = [1,5,0,10,14]`

**Output:** 1

**Explanation:** Change the array `[1,5,0,10,14]` to `[1,1,0,1,1]`.

The difference between the maximum and minimum is  $1-0 = 1$ .

### Example 3:

**Input:** nums = [6,6,0,1,1,4,6]

**Output:** 2

#### Example 4:

**Input:** nums = [1,5,6,14,15]

**Output:** 1

#### Constraints:

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

```
class Solution {
 public int minDifference(int[] nums) {

 if (nums.length <= 4) return 0;
 int l = nums.length - 1;
 int min = Integer.MAX_VALUE;

 Arrays.sort(nums);

 //kill 3 big number
 min = Math.min(min, nums[l-3] - nums[0]);
 //kill 3 small number
 min = Math.min(min, nums[1] - nums[3]);
 //kill 1 small and 2 big
 min = Math.min(min, nums[l-2] - nums[1]);
 //kill 2 small and 1 big
 min = Math.min(min, nums[l-1] - nums[2]);

 return min;
 }
}
```

## 1525. Number of Good Ways to Split a String ↗

You are given a string `s`, a split is called *good* if you can split `s` into 2 non-empty strings `p` and `q` where its concatenation is equal to `s` and the number of distinct letters in `p` and `q` are the same.

Return the number of *good* splits you can make in `s`.

### Example 1:

**Input:** `s = "aacaba"`

**Output:** 2

**Explanation:** There are 5 ways to split "aacaba" and 2 of them are good.

(`"a"`, `"acaba"`) Left string and right string contains 1 and 3 different letter

(`"aa"`, `"caba"`) Left string and right string contains 1 and 3 different letter

(`"aac"`, `"aba"`) Left string and right string contains 2 and 2 different letter

(`"aaca"`, `"ba"`) Left string and right string contains 2 and 2 different letter

(`"aacab"`, `"a"`) Left string and right string contains 3 and 1 different letter

### Example 2:

**Input:** `s = "abcd"`

**Output:** 1

**Explanation:** Split the string as follows (`"ab"`, `"cd"`).

### Example 3:

**Input:** `s = "aaaaa"`

**Output:** 4

**Explanation:** All possible splits are good.

### Example 4:

**Input:** `s = "acbadbaada"`

**Output:** 2

### Constraints:

- `s` contains only lowercase English letters.
- $1 \leq s.length \leq 10^5$

```

class Solution {
 public int numSplits(String s) {

 int waysToSplit = 0;
 Set<Character> set = new HashSet<>();

 int[] prefix = new int[s.length()];//number of unique char left to
right
 int[] sufix = new int[s.length()];//number of unique char reight to
left
 int count = 0;
 for (int i = 0; i < s.length(); i++) {
 if (!set.contains(s.charAt(i))) {
 count++;
 prefix[i] = count;
 set.add(s.charAt(i));
 }
 else {
 prefix[i] = count;
 }
 }
 count = 0;
 set = new HashSet<>();
 for (int i = s.length()-1; i >= 0; i--) {
 if (!set.contains(s.charAt(i))) {
 count++;
 sufix[i] = count;
 set.add(s.charAt(i));
 }
 else {
 sufix[i] = count;
 }
 }

 for (int i = 0; i < prefix.length-1; i++) {
 if (prefix[i] == sufix[i+1]) waysToSplit++; //if unique char fr
om start to i and i+1 to end is same that means we can cut
 }
 return waysToSplit;
 }
}

```

# 1526. Minimum Number of Increments on Subarrays to Form a Target Array

Given an array of positive integers `target` and an array `initial` of same size with all zeros.

Return the minimum number of operations to form a `target` array from `initial` if you are allowed to do the following operation:

- Choose **any** subarray from `initial` and increment each value by one.

The answer is guaranteed to fit within the range of a 32-bit signed integer.

## Example 1:

**Input:** target = [1,2,3,2,1]

**Output:** 3

**Explanation:** We need at least 3 operations to form the target array from the [0,0,0,0,0] increment 1 from index 0 to 4 (inclusive).

[1,1,1,1,1] increment 1 from index 1 to 3 (inclusive).

[1,2,2,2,1] increment 1 at index 2.

[1,2,3,2,1] target array is formed.

## Example 2:

**Input:** target = [3,1,1,2]

**Output:** 4

**Explanation:** (initial)[0,0,0,0] -> [1,1,1,1] -> [1,1,1,2] -> [2,1,1,2] -> [3,

## Example 3:

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

**Output:** 7

**Explanation:** (initial)[0,0,0,0,0] -> [1,1,1,1,1] -> [2,1,1,1,1] -> [3,1,1,1,1] -> [3,1,2,2,2] -> [3,1,3,3,2] -> [3,1,4,4,2]

## Example 4:

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

**Output:** 1

### Constraints:

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

//Greedy :<https://www.youtube.com/watch?v=s6fytsG-UI>

```
class Solution {
 public int minNumberOperations(int[] target) {
 int maxMoves = target[0];
 for (int i = 1; i < target.length; i++) {
 //neche aate time ignoew ...but agr value jyaf=da hai to utna a
dd karege..qki sub array ko bhi dhyaan rakhna tha
 if (target[i] > target[i-1]) {
 int diff = target[i] - target[i-1];
 maxMoves += diff;
 }
 }
 return maxMoves;
 }
}
```

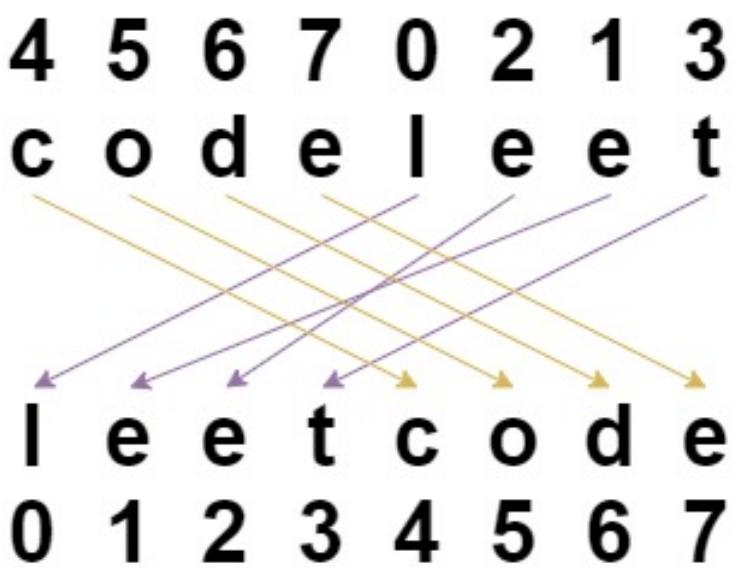
## 1528. Shuffle String ↗

Given a string `s` and an integer array `indices` of the **same length**.

The string `s` will be shuffled such that the character at the  $i^{th}$  position moves to `indices[i]` in the shuffled string.

Return *the shuffled string*.

### Example 1:



**Input:** s = "codeleet", indices = [4,5,6,7,0,2,1,3]

**Output:** "leetcode"

**Explanation:** As shown, "codeleet" becomes "leetcode" after shuffling.

### Example 2:

**Input:** s = "abc", indices = [0,1,2]

**Output:** "abc"

**Explanation:** After shuffling, each character remains in its position.

### Example 3:

**Input:** s = "aiohn", indices = [3,1,4,2,0]

**Output:** "nihao"

### Example 4:

**Input:** s = "aaiougrt", indices = [4,0,2,6,7,3,1,5]

**Output:** "arigatou"

### Example 5:

**Input:** s = "art", indices = [1,0,2]

**Output:** "rat"

### Constraints:

- s.length == indices.length == n
- 1 <= n <= 100
- s contains only lower-case English letters.

- $0 \leq \text{indices}[i] < n$
- All values of `indices` are unique (i.e. `indices` is a permutation of the integers from  $0$  to  $n - 1$ ).

```
class Solution {
 public String restoreString(String s, int[] indices) {

 char[] ch = s.toCharArray(); //convert string to char array
 char[] ans = new char[indices.length];

 for (int i = 0; i < indices.length; i++) {
 int indexToSend = indices[i];
 ans[indexToSend] = ch[i];
 }
 return String.valueOf(ans);
 }
}
```

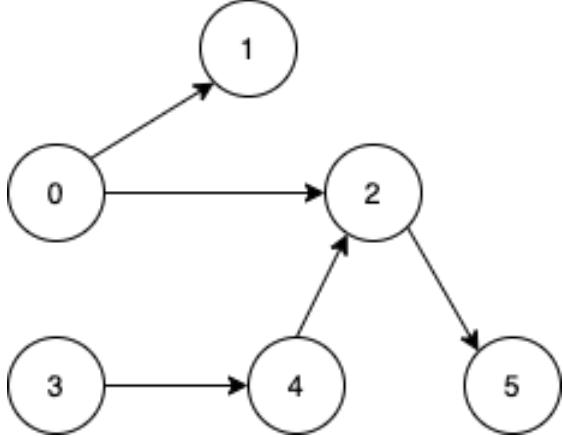
## 1557. Minimum Number of Vertices to Reach All Nodes ↴

Given a **directed acyclic graph**, with  $n$  vertices numbered from  $0$  to  $n-1$ , and an array `edges` where  $\text{edges}[i] = [\text{from}_i, \text{to}_i]$  represents a directed edge from node  $\text{from}_i$  to node  $\text{to}_i$ .

Find *the smallest set of vertices from which all nodes in the graph are reachable*. It's guaranteed that a unique solution exists.

Notice that you can return the vertices in any order.

**Example 1:**

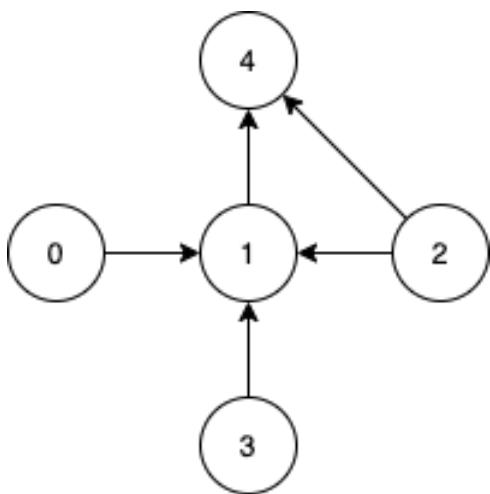


**Input:** n = 6, edges = [[0,1],[0,2],[2,5],[3,4],[4,2]]

**Output:** [0,3]

**Explanation:** It's not possible to reach all the nodes from a single vertex. F

### Example 2:



**Input:** n = 5, edges = [[0,1],[2,1],[3,1],[1,4],[2,4]]

**Output:** [0,2,3]

**Explanation:** Notice that vertices 0, 3 and 2 are not reachable from any other

### Constraints:

- $2 \leq n \leq 10^5$
- $1 \leq \text{edges.length} \leq \min(10^5, n * (n - 1) / 2)$
- $\text{edges}[i].length == 2$
- $0 \leq \text{from}_i, \text{to}_i < n$
- All pairs  $(\text{from}_i, \text{to}_i)$  are distinct.

```

//Find Indegree: node with 0 indegree is the ans
class Solution {
 public List<Integer> findSmallestSetOfVertices(int n, List<List<Integer>> edges) {

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

 int[] indegree = new int[n];

 for (List<Integer> edge : edges) { //inner AL (yaha pahucha ja skta h i se)

 int source = edge.get(0);
 int dest = edge.get(1);

 indegree[dest]++; //yaha kitne raste se pahucha ja skta h
 }
 // System.out.print(Arrays.toString(indegree));

 for (int i = 0; i < indegree.length; i++) {
 if (indegree[i] == 0) ans.add(i); //qki yaha se start karna he h..kahe se bhi yaha nahi pahucha ja skta
 }

 return ans;
 }
}

```

## 1551. Minimum Operations to Make Array Equal

You have an array `arr` of length `n` where  $arr[i] = (2 * i) + 1$  for all valid values of `i` (i.e.  $0 \leq i < n$ ).

In one operation, you can select two indices `x` and `y` where  $0 \leq x, y < n$  and subtract 1 from `arr[x]` and add 1 to `arr[y]` (i.e. perform `arr[x] -= 1` and `arr[y] += 1`). The goal is to make all the elements of the array **equal**. It is **guaranteed** that all the elements of the array can be made equal using some operations.

Given an integer `n`, the length of the array. Return *the minimum number of operations* needed to make all the elements of `arr` equal.

### **Example 1:**

**Input:** n = 3

**Output:** 2

**Explanation:** arr = [1, 3, 5]

First operation choose x = 2 and y = 0, this leads arr to be [2, 3, 4]

In the second operation choose x = 2 and y = 0 again, thus arr = [3, 3, 3].

### **Example 2:**

**Input:** n = 6

**Output:** 9

### **Constraints:**

- $1 \leq n \leq 10^4$

Basic Approach: if n is odd, suppose n=5. The array is : 1 3 5 7 9. Here, we will have the middle element as 5. We take 2 from 7 and add to 3 to make each one 5. We take 4 from 9 and add to 1 to make each one 5. Total steps:  $2+4=6$ . (sum of first  $n/2$  even numbers) Sum of first k EVEN numbers =  $\text{Sum}(i=1 \text{ to } k) \{2 * i\} = 2 * (\text{Sum}(i=1 \text{ to } k) \{i\}) = 2 * (k(k+1))/2 = k(k+1)$  if n is even, suppose n=6. The array is : 1 3 5 7 9 11. Here, we will have the middle element as  $(5+7)/2=6$ . We take 1 from 7 and add to 5 to make each one 6. We take 3 from 9 and add to 3 to make each one 6. We take 5 from 11 and add to 1 to make each one 6. Total steps:  $1+3+5=9$ . (sum of first  $n/2$  odd numbers) Sum of first k ODD numbers =  $k * k$ .

The focus is on the difference between the middle element and the first  $n/2$  numbers.

So for example:

1, 3, 5, 7, 9, 11 +5, +3, +1, -1, -3 , -5

As you can see, the sequence (+5, +3, +1) is the same as the sum of the first  $n/2$  numbers (1, 3, 5), just the opposite.

```
//Math
class Solution
{
 public int minOperations(int n)
 {
 // Take care of overflow if n is too large.
 if(n%2==1){
 n/=2;
 return (n*(n+1));
 }
 n/=2;
 return n*n;
 }
}
```

## 1631. Path With Minimum Effort ↗

You are a hiker preparing for an upcoming hike. You are given `heights`, a 2D array of size `rows`  $\times$  `columns`, where `heights[row][col]` represents the height of cell  $(row, col)$ . You are situated in the top-left cell,  $(0, 0)$ , and you hope to travel to the bottom-right cell,  $(rows-1, columns-1)$  (i.e., **0-indexed**). You can move **up**, **down**, **left**, or **right**, and you wish to find a route that requires the minimum **effort**.

A route's **effort** is the **maximum absolute difference** in heights between two consecutive cells of the route.

Return *the minimum **effort** required to travel from the top-left cell to the bottom-right cell.*

**Example 1:**

|   |   |   |
|---|---|---|
| 1 | 2 | 2 |
| 3 | 8 | 2 |
| 5 | 3 | 5 |

**Input:** heights = [[1,2,2],[3,8,2],[5,3,5]]

**Output:** 2

**Explanation:** The route of [1,3,5,3,5] has a maximum absolute difference of 2  
This is better than the route of [1,2,2,2,5], where the maximum absolute diff

### Example 2:

|   |   |   |
|---|---|---|
| 1 | 2 | 3 |
| 3 | 8 | 4 |
| 5 | 3 | 5 |

**Input:** heights = [[1,2,3],[3,8,4],[5,3,5]]

**Output:** 1

**Explanation:** The route of [1,2,3,4,5] has a maximum absolute difference of 1

### Example 3:

|   |   |   |   |   |
|---|---|---|---|---|
| 1 | 2 | 1 | 1 | 1 |
| 1 | 2 | 1 | 2 | 1 |
| 1 | 2 | 1 | 2 | 1 |
| 1 | 2 | 1 | 2 | 1 |
| 1 | 1 | 1 | 2 | 1 |

**Input:** heights = [[1,2,1,1,1],[1,2,1,2,1],[1,2,1,2,1],[1,2,1,2,1],[1,1,1,2,1]]

**Output:** 0

**Explanation:** This route does not require any effort.

### Constraints:

- rows == heights.length
- columns == heights[i].length
- 1 <= rows, columns <= 100
- 1 <= heights[i][j] <=  $10^6$

```

//shamjna hai Dijkstra
class Solution {
 public int minimumEffortPath(int[][][] heights) {

 int[][][] effort = new int[heights.length][heights[0].length];
 for (int[] cR: effort) {
 Arrays.fill(cR, Integer.MAX_VALUE);
 }

 PriorityQueue<int[]> pq = new PriorityQueue<>((a,b) -> (a[2] - b
[2]));
 effort[0][0] = 0;
 pq.add(new int[]{0,0,0}); //row, col, effort

 while (!pq.isEmpty()) {

 int[] currBox = pq.poll();
 int currRow = currBox[0];
 int currCol = currBox[1];
 int currWt = currBox[2];

 if (currRow == heights.length-1 && currCol == heights[0].length
-1){
 return currWt;
 }
 //down
 if (currRow +1 < heights.length) {
 int newWt = Math.max(currWt, Math.abs(heights[currRow][curr
Col] - heights[currRow+1][currCol]));
 if (effort[currRow+1][currCol] > newWt) {
 effort[currRow+1][currCol] = newWt;
 pq.add(new int[]{currRow+1, currCol, newWt});
 }
 }
 //left
 if (currRow -1 >= 0) {
 int newWt = Math.max(currWt, Math.abs(heights[currRow][curr
Col] - heights[currRow-1][currCol]));
 if (effort[currRow-1][currCol] > newWt) {
 effort[currRow-1][currCol] = newWt;
 pq.add(new int[]{currRow-1, currCol, newWt});
 }
 }
 //up
 if (currCol -1 >= 0) {
 int newWt = Math.max(currWt, Math.abs(heights[currRow][curr
Col] - heights[currRow][currCol-1]));
 if (effort[currRow][currCol-1] > newWt) {
 effort[currRow][currCol-1] = newWt;
 pq.add(new int[]{currRow, currCol-1, newWt});
 }
 }
 }
 }
}

```

```

 Col] - heights[currRow][currCol-1]));
 if (effort[currRow][currCol-1] > newWt) {
 effort[currRow][currCol-1] = newWt;
 pq.add(new int[]{currRow, currCol-1, newWt});
 }
 }
 //down
 if (currCol +1 < heights[0].length) {
 int newWt = Math.max(currWt, Math.abs(heights[currRow][curr
Col] - heights[currRow][currCol+1]));
 if (effort[currRow][currCol+1] > newWt) {
 effort[currRow][currCol+1] = newWt;
 pq.add(new int[]{currRow, currCol+1, newWt});
 }
 }
}
return 0;
}

}

```

## 1704. Determine if String Halves Are Alike ↗ ▾

You are given a string `s` of even length. Split this string into two halves of equal lengths, and let `a` be the first half and `b` be the second half.

Two strings are **alike** if they have the same number of vowels ('a', 'e', 'i', 'o', 'u', 'A', 'E', 'I', 'O', 'U'). Notice that `s` contains uppercase and lowercase letters.

Return `true` if `a` and `b` are **alike**. Otherwise, return `false`.

### Example 1:

**Input:** `s = "book"`

**Output:** `true`

**Explanation:** `a = "bo"` and `b = "ok"`. `a` has 1 vowel and `b` has 1 vowel. Therefore,

### Example 2:

**Input:** s = "textbook"

**Output:** false

**Explanation:** a = "text" and b = "book". a has 1 vowel whereas b has 2. Therefor Notice that the vowel o is counted twice.

### Example 3:

**Input:** s = "MerryChristmas"

**Output:** false

### Example 4:

**Input:** s = "AbCdEfGh"

**Output:** true

### Constraints:

- $2 \leq s.length \leq 1000$
- $s.length$  is even.
- $s$  consists of **uppercase and lowercase** letters.

```

class Solution {
 public boolean halvesAreAlike(String s)
 {
 s = s.toLowerCase();
 char chars[] = s.toCharArray();

 int l = chars.length;

 int i = 0;
 int j = l/2;
 //System.out.print("odd " + 3/2 + "even " + 2/2); 1,1
 int ce = 0;
 int co = 0;

 while (i < l/2 && j < l)
 {
 if (chars[i] == 'a' || chars[i] == 'e' || chars[i] == 'i' || chars[i] == 'o' || chars[i] == 'u') ce++;

 if (chars[j] == 'a' || chars[j] == 'e' || chars[j] == 'i' || chars[j] == 'o' || chars[j] == 'u') co++;

 i++;
 j++;
 }

 if (ce == co) return true;

 return false;
 }
}

```

## 1773. Count Items Matching a Rule ↗

You are given an array `items`, where each `items[i] = [typei, colori, namei]` describes the type, color, and name of the  $i^{\text{th}}$  item. You are also given a rule represented by two strings, `ruleKey` and `ruleValue`.

The  $i^{\text{th}}$  item is said to match the rule if **one** of the following is true:

- `ruleKey == "type"` and `ruleValue == typei`.
- `ruleKey == "color"` and `ruleValue == colori`.

- $\text{ruleKey} == \text{"name"}$  and  $\text{ruleValue} == \text{name}_i$ .

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

### Example 1:

**Input:** items = [["phone", "blue", "pixel"], ["computer", "silver", "lenovo"], ["pho

**Output:** 1

**Explanation:** There is only one item matching the given rule, which is ["compu

### Example 2:

**Input:** items = [["phone", "blue", "pixel"], ["computer", "silver", "phone"], ["pho

**Output:** 2

**Explanation:** There are only two items matching the given rule, which are ["pho

### Constraints:

- $1 \leq \text{items.length} \leq 10^4$
- $1 \leq \text{type}_i.\text{length}, \text{color}_i.\text{length}, \text{name}_i.\text{length}, \text{ruleValue.length} \leq 10$
- $\text{ruleKey}$  is equal to either "type", "color", or "name".
- All strings consist only of lowercase letters.

```

class Solution {
 public int countMatches(List<List<String>> items, String ruleKey, String ruleValue)
 {
 //List<List<String>> ans = ArrayList<List<String>>();
 int count = 0;
 int indexToCheck = 0;

 if (ruleKey.equals("color"))
 {
 indexToCheck = 1;
 }
 else if (ruleKey.equals("type"))
 {
 indexToCheck = 0;
 }
 else if (ruleKey.equals("name"))
 {
 indexToCheck = 2;
 }

 for (int i = 0; i < items.size(); i++)
 {
 if (items.get(i).get(indexToCheck).equals(ruleValue))
 {
 //ans.add(items.get(i));
 ++count;
 }
 }
 return count;
 }
}

```

## 1762. Buildings With an Ocean View ↗

There are `n` buildings in a line. You are given an integer array `heights` of size `n` that represents the heights of the buildings in the line.

The ocean is to the right of the buildings. A building has an ocean view if the building can see the ocean without obstructions. Formally, a building has an ocean view if all the buildings to its right have a **smaller** height.

Return a list of indices (**0-indexed**) of buildings that have an ocean view, sorted in increasing order.

### Example 1:

**Input:** heights = [4,2,3,1]

**Output:** [0,2,3]

**Explanation:** Building 1 (0-indexed) does not have an ocean view because building 2 is taller than it.

### Example 2:

**Input:** heights = [4,3,2,1]

**Output:** [0,1,2,3]

**Explanation:** All the buildings have an ocean view.

### Example 3:

**Input:** heights = [1,3,2,4]

**Output:** [3]

**Explanation:** Only building 3 has an ocean view.

### Example 4:

**Input:** heights = [2,2,2,2]

**Output:** [3]

**Explanation:** Buildings cannot see the ocean if there are buildings of the same height to their left.

### Constraints:

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

```

class Solution {
 public int[] findBuildings(int[] heights) {

 Stack<Integer> st = new Stack<>();
 //maintain largest to the right
 int maxHeight = 0;
 for (int i = heights.length -1; i >= 0; i--) {

 if (heights[i] > maxHeight) {
 st.push(i);
 maxHeight = heights[i];
 }
 }
 int[] oceanViewBuildings = new int[st.size()];
 int index = 0;

 while (!st.isEmpty()) {

 oceanViewBuildings[index] = st.pop();
 index++;
 }
 return oceanViewBuildings;
 }
}

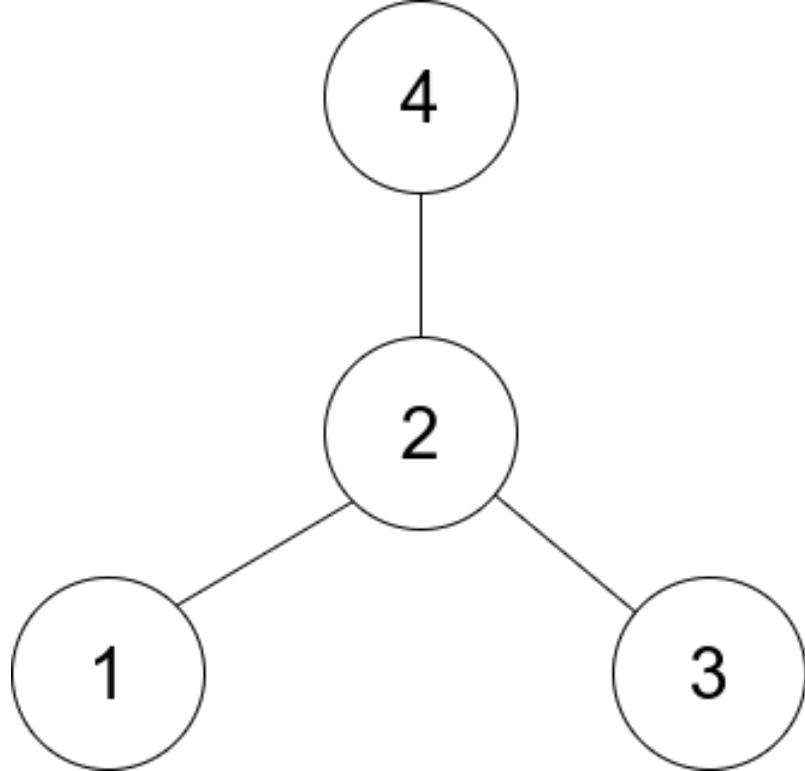
```

## 1791. Find Center of Star Graph ↗

There is an undirected **star** graph consisting of  $n$  nodes labeled from  $1$  to  $n$ . A star graph is a graph where there is one **center** node and **exactly**  $n - 1$  edges that connect the center node with every other node.

You are given a 2D integer array `edges` where each `edges[i] = [ui, vi]` indicates that there is an edge between the nodes  $u_i$  and  $v_i$ . Return the center of the given star graph.

**Example 1:**



**Input:** edges = [[1,2],[2,3],[4,2]]

**Output:** 2

**Explanation:** As shown in the figure above, node 2 is connected to every other

### Example 2:

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

**Output:** 1

### Constraints:

- $3 \leq n \leq 10^5$
- $\text{edges.length} == n - 1$
- $\text{edges}[i].length == 2$
- $1 \leq u_i, v_i \leq n$
- $u_i \neq v_i$
- The given edges represent a valid star graph.

```
class Solution {
 public int findCenter(int[][] edges) {

 Set<Integer> visited = new HashSet<>();

 for (int[] edge : edges){
 if (visited.contains(edge[0])) return edge[0];
 if (visited.contains(edge[1])) return edge[1];

 visited.add(edge[0]);
 visited.add(edge[1]);
 }
 return 0;
 }
}
```

```

class Solution {
 public int findCenter(int[][] edges) {

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

 for (int i = 0; i <= edges.length+1; i++) {
 adj.add(new ArrayList<>());
 }
 // System.out.print(adj.size());

 for (int[] edge : edges){

 adj.get(edge[0]).add(edge[1]);
 adj.get(edge[1]).add(edge[0]);
 }
 int maxEdgeLength = Integer.MIN_VALUE;
 int maxEdgeNode = Integer.MIN_VALUE;
 for (int i = 0; i < adj.size(); i++) {

 if (adj.get(i).size() > maxEdgeLength) {
 maxEdgeLength = adj.get(i).size();
 maxEdgeNode = i;
 }
 }

 return maxEdgeNode;
 }
}

```

## 1832. Check if the Sentence Is Pangram ↗

A **pangram** is a sentence where every letter of the English alphabet appears at least once.

Given a string `sentence` containing only lowercase English letters, return `true` if `sentence` is a **pangram**, or `false` otherwise.

**Example 1:**

**Input:** sentence = "thequickbrownfoxjumpsoverthelazydog"

**Output:** true

**Explanation:** sentence contains at least one of every letter of the English alphabet.

### Example 2:

**Input:** sentence = "leetcode"

**Output:** false

### Constraints:

- $1 \leq \text{sentence.length} \leq 1000$
- sentence consists of lowercase English letters.

```
class Solution {
 public boolean checkIfPangram(String sentence)
 {
 boolean[] check = new boolean[26]; //26 character

 for (int i = 0; i < sentence.length(); i++)
 {
 int index = (int)(sentence.charAt(i) - 'a'); //for a index is
0, z -> 25
 check[index] = true;
 }
 for (int i = 0; i < check.length; i++)
 {
 if (check[i] == false) return false;
 }
 return true;
 }
}
```

You are given  $n$  tasks labeled from  $0$  to  $n - 1$  represented by a 2D integer array `tasks`, where `tasks[i] = [enqueueTimei, processingTimei]` means that the  $i^{\text{th}}$  task will be available to process at `enqueueTimei` and will take `processingTimei` to finish processing.

You have a single-threaded CPU that can process **at most one** task at a time and will act in the following way:

- If the CPU is idle and there are no available tasks to process, the CPU remains idle.
- If the CPU is idle and there are available tasks, the CPU will choose the one with the **shortest processing time**. If multiple tasks have the same shortest processing time, it will choose the task with the smallest index.
- Once a task is started, the CPU will **process the entire task** without stopping.
- The CPU can finish a task then start a new one instantly.

Return *the order in which the CPU will process the tasks*.

### Example 1:

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

**Output:** `[0,2,3,1]`

**Explanation:** The events go as follows:

- At time = 1, task 0 is available to process. Available tasks =  $\{0\}$ .
- Also at time = 1, the idle CPU starts processing task 0. Available tasks =
- At time = 2, task 1 is available to process. Available tasks =  $\{1\}$ .
- At time = 3, task 2 is available to process. Available tasks =  $\{1, 2\}$ .
- Also at time = 3, the CPU finishes task 0 and starts processing task 2 as it is.
- At time = 4, task 3 is available to process. Available tasks =  $\{1, 3\}$ .
- At time = 5, the CPU finishes task 2 and starts processing task 3 as it is.
- At time = 6, the CPU finishes task 3 and starts processing task 1. Available tasks =
- At time = 10, the CPU finishes task 1 and becomes idle.

### Example 2:

**Input:** tasks = [[7,10],[7,12],[7,5],[7,4],[7,2]]

**Output:** [4,3,2,0,1]

**Explanation:** The events go as follows:

- At time = 7, all the tasks become available. Available tasks = {0,1,2,3,4}.
- Also at time = 7, the idle CPU starts processing task 4. Available tasks =
- At time = 9, the CPU finishes task 4 and starts processing task 3. Available tasks = {0,1,2,3}
- At time = 13, the CPU finishes task 3 and starts processing task 2. Available tasks = {0,1,2}
- At time = 18, the CPU finishes task 2 and starts processing task 0. Available tasks = {1,0}
- At time = 28, the CPU finishes task 0 and starts processing task 1. Available tasks = {0}
- At time = 40, the CPU finishes task 1 and becomes idle.

### **Constraints:**

- `tasks.length == n`
  - $1 \leq n \leq 10^5$
  - $1 \leq \text{enqueueTime}_i, \text{processingTime}_i \leq 10^9$
-

```

//while (ansIndex < len) top ki condition...jb tk hume array pura nahi mila
tb tk continue karo
class Solution {
 public int[] getOrder(int[][] tasks) {
 int len = tasks.length;
 int[] ans = new int[len];
 Task[] ToDo = new Task[len]; //task with index,inTime,processTime
 for (int i = 0; i < len; i++) {
 ToDo[i] = new Task(i, tasks[i][0], tasks[i][1]);
 }

 Arrays.sort(ToDo, (a,b) -> a.inTime - b.inTime); //sorted array based on inTime

 PriorityQueue<Task> pq = new PriorityQueue<>((a,b) -> { //based on
processing time
 if (a.processTime == b.processTime) {
 return a.index - b.index;
 }
 else return a.processTime - b.processTime;
 });
 //First task hum add he kar de rai..jissey jyada complicated na ho
 int ansIndex = 0, ToDoIndex = 1, currentTime = ToDo[0].inTime;
 pq.add(ToDo[0]);
 while (ansIndex < len) { //abhi array pura bhara nahi h

 if (pq.isEmpty()) { //CPU khali baitha hai..to array se next le
lo
 pq.add(ToDo[ToDoIndex]); //adding 1st task from array
 currentTime = Math.max(ToDo[ToDoIndex].inTime, currentTim
e); //max islia qki chances hai ki current time aage ho is task k in time se
 ToDoIndex++;
 }
 while (ToDoIndex < len && ToDo[ToDoIndex].inTime <= currentTim
e) {
 pq.add(ToDo[ToDoIndex]);
 ToDoIndex++;
 }

 Task currTask = pq.remove();
 currentTime = currentTime+ currTask.processTime; //processed
 ans[ansIndex] = currTask.index;
 ansIndex++;
 }
 return ans;
 }
}

```

```

class Task{

 int index;
 int inTime;
 int processTime;

 Task(int index, int inTime, int processTime) {
 this.index = index;
 this.inTime = inTime;
 this.processTime = processTime;
 }

}

```

## 1877. Minimize Maximum Pair Sum in Array ↗

The **pair sum** of a pair  $(a, b)$  is equal to  $a + b$ . The **maximum pair sum** is the largest **pair sum** in a list of pairs.

- For example, if we have pairs  $(1, 5)$ ,  $(2, 3)$ , and  $(4, 4)$ , the **maximum pair sum** would be  $\max(1+5, 2+3, 4+4) = \max(6, 5, 8) = 8$ .

Given an array `nums` of **even** length  $n$ , pair up the elements of `nums` into  $n / 2$  pairs such that:

- Each element of `nums` is in **exactly one** pair, and
- The **maximum pair sum** is **minimized**.

Return *the minimized **maximum pair sum** after optimally pairing up the elements.*

### Example 1:

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

**Output:** 7

**Explanation:** The elements can be paired up into pairs  $(3, 3)$  and  $(5, 2)$ . The maximum pair sum is  $\max(3+3, 5+2) = \max(6, 7) = 7$ .

### Example 2:

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

**Output:** 8

**Explanation:** The elements can be paired up into pairs (3,5), (4,4), and (6,2)

The maximum pair sum is  $\max(3+5, 4+4, 6+2) = \max(8, 8, 8) = 8$ .

### Constraints:

- $n == \text{nums.length}$
- $2 \leq n \leq 10^5$
- $n$  is **even**.
- $1 \leq \text{nums}[i] \leq 10^5$

```
//mistakes...mujhe laga subarray and binaray search sochne lagi... but this
is paise so 2 pointer can be used....
```

```
//sort karo2 pointer...find max sum a pair can have
```

```
class Solution {
 public int minPairSum(int[] nums) {

 Arrays.sort(nums);

 int left = 0, right = nums.length-1;
 int maxSum = Integer.MIN_VALUE;
 // int sum = 0;

 while (left < right) {
 // sum = nums[left] + nums[right];
 maxSum = Math.max(maxSum, nums[left++] + nums[right--]);
 // left++;
 // right--;
 }
 return maxSum;
 }
}
```



Given an integer array `nums` and an integer `k`, you are asked to construct the array `ans` of size  $n-k+1$  where `ans[i]` is the number of **distinct** numbers in the subarray `nums[i:i+k-1] = [nums[i], nums[i+1], \dots, nums[i+k-1]]`.

Return *the array ans*.

### Example 1:

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

**Output:** `[3,2,2,2,3]`

**Explanation:** The number of distinct elements in each subarray goes as follows

- `nums[0:2] = [1,2,3]` so `ans[0] = 3`
- `nums[1:3] = [2,3,2]` so `ans[1] = 2`
- `nums[2:4] = [3,2,2]` so `ans[2] = 2`
- `nums[3:5] = [2,2,1]` so `ans[3] = 2`
- `nums[4:6] = [2,1,3]` so `ans[4] = 3`

### Example 2:

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

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

**Explanation:** The number of distinct elements in each subarray goes as follows

- `nums[0:3] = [1,1,1,1]` so `ans[0] = 1`
- `nums[1:4] = [1,1,1,2]` so `ans[1] = 2`
- `nums[2:5] = [1,1,2,3]` so `ans[2] = 3`
- `nums[3:6] = [1,2,3,4]` so `ans[3] = 4`

### Constraints:

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

```
//bahut c confusions hai ye he patter yaad rakho simple hai
class Solution {
 public int[] distinctNumbers(int[] nums, int k) {

 if (k > nums.length) return new int[0];

 HashMap<Integer, Integer> map = new HashMap<>();
 int[] ans = new int[nums.length - k + 1];
 int start = 0;
 int end = 0;
 int index = 0;

 while (end < nums.length) {

 map.put(nums[end], map.getOrDefault(nums[end], 0)+1); //jo aa r
a rakh lo

 if (end - start+1 < k){
 end++; //map me to add kar he chuke h pahle he
 }
 else if (end- start +1 == k) { //jb window size mil gaya ab kaa
m karo

 ans[index] = map.size(); //itne he unique honge

 //remove 1st
 int startFreq = map.get(nums[start]);
 if (startFreq == 1) {
 map.remove(nums[start]); //removed qki ab freq 0 ho jat
i
 }
 else {
 map.put(nums[start], map.get(nums[start])-1);

 }
 start++;
 index++;
 end++; //end ko increase karna na bhulna*****
 }
 }
 return ans;
}
}
```

# 1910. Remove All Occurrences of a Substring

Given two strings `s` and `part`, perform the following operation on `s` until **all** occurrences of the substring `part` are removed:

- Find the **leftmost** occurrence of the substring `part` and **remove** it from `s`.

Return `s` *after removing all occurrences of part*.

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

## Example 1:

**Input:** `s = "daabcbaabcbc"`, `part = "abc"`

**Output:** `"dab"`

**Explanation:** The following operations are done:

- `s = "daabcbaabcbc"`, remove "abc" starting at index 2, so `s = "dabaabcbc"`.
- `s = "dabaabcbc"`, remove "abc" starting at index 4, so `s = "dababc"`.
- `s = "dababc"`, remove "abc" starting at index 3, so `s = "dab"`.

Now `s` has no occurrences of "abc".

## Example 2:

**Input:** `s = "axxxxxyyyyb"`, `part = "xy"`

**Output:** `"ab"`

**Explanation:** The following operations are done:

- `s = "axxxxxyyyyb"`, remove "xy" starting at index 4 so `s = "axxxyyyb"`.
- `s = "axxxyyyb"`, remove "xy" starting at index 3 so `s = "axxyyb"`.
- `s = "axxyyb"`, remove "xy" starting at index 2 so `s = "axyb"`.
- `s = "axyb"`, remove "xy" starting at index 1 so `s = "ab"`.

Now `s` has no occurrences of "xy".

## Constraints:

- `1 <= s.length <= 1000`
- `1 <= part.length <= 1000`
- `s` and `part` consists of lowercase English letters.

```

//s.indexOf(part) give index of first occ
class Solution {
 public String removeOccurrences(String s, String part) {

 // if (part.length() > s.length()) return ""; don't add failing for
 "a""aa"

 int start = 0; int end = s.length();
 int partLen = part.length();

 while (true) {

 int firstOcc = s.indexOf(part);

 if (firstOcc != -1) {

 s = s.substring(start, firstOcc) + s.substring(firstOcc+pa
rtLen);
 }
 else if (firstOcc == -1) break; //ab or substring "abc" nahi h
 }

 return s;
 }
}

```

Slow hai but concept k lia..kafe corner case the

```

class Solution {
 public String removeOccurrences(String s, String part) {

 Stack<Character> st = new Stack<>();
 char lastCharP = part.charAt(part.length()-1);
 boolean flag = true;
 for (int i = 0; i < s.length();) {

 while (i < s.length() && s.charAt(i) != lastCharP) {
 st.push(s.charAt(i));
 i++;
 }
 if (i < s.length()) st.push(s.charAt(i));
 i++;
 while (!st.isEmpty()){
 int popCount = part.length();
 String temp = "";
 while (!st.isEmpty() && popCount > 0)
 {
 temp = st.pop() + temp;
 popCount--;
 }

 if (!temp.equals(part)) {
 for (int j = 0; j < temp.length(); j++) {
 st.push(temp.charAt(j));
 }
 break;
 }
 }
 }
 String ans = "";
 while (!st.isEmpty()) {
 ans = st.pop()+ ans;
 }
 return ans;
 }
}

```

## 1937. Maximum Number of Points with Cost ↗

You are given an  $m \times n$  integer matrix **points (0-indexed)**. Starting with 0 points, you want to **maximize** the number of points you can get from the matrix.

To gain points, you must pick one cell in **each row**. Picking the cell at coordinates  $(r, c)$  will **add**  $\text{points}[r][c]$  to your score.

However, you will lose points if you pick a cell too far from the cell that you picked in the previous row. For every two adjacent rows  $r$  and  $r + 1$  (where  $0 \leq r < m - 1$ ), picking cells at coordinates  $(r, c_1)$  and  $(r + 1, c_2)$  will **subtract**  $\text{abs}(c_1 - c_2)$  from your score.

Return *the maximum number of points you can achieve*.

$\text{abs}(x)$  is defined as:

- $x$  for  $x \geq 0$ .
- $-x$  for  $x < 0$ .

### Example 1:

|   |   |   |
|---|---|---|
| 1 | 2 | 3 |
| 1 | 5 | 1 |
| 3 | 1 | 1 |

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

**Output:** 9

#### Explanation:

The blue cells denote the optimal cells to pick, which have coordinates  $(0, 2)$ ,  $(1, 1)$ , and  $(2, 0)$ . You add  $3 + 5 + 3 = 11$  to your score.

However, you must subtract  $\text{abs}(2 - 1) + \text{abs}(1 - 0) = 2$  from your score.

Your final score is  $11 - 2 = 9$ .

### Example 2:

|   |   |
|---|---|
| 1 | 5 |
| 2 | 3 |
| 4 | 2 |

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

**Output:** 11

**Explanation:**

The blue cells denote the optimal cells to pick, which have coordinates (0, 1) and (1, 0). You add  $5 + 3 + 4 = 12$  to your score.

However, you must subtract  $\text{abs}(1 - 1) + \text{abs}(1 - 0) = 1$  from your score.

Your final score is  $12 - 1 = 11$ .

### Constraints:

- $m == \text{points.length}$
- $n == \text{points[r].length}$
- $1 \leq m, n \leq 10^5$
- $1 \leq m * n \leq 10^5$
- $0 \leq \text{points}[r][c] \leq 10^5$

//TLE

```
class Solution {
 public long maxPoints(int[][] points) {

 int row = points.length;
 int col = points[0].length;
 long[][] dp = new long[row][col];
 long maximumSum = 0;
 for (int i = col-1; i >=0; i--) {
 dp[row-1][i] = points[row-1][i];
 }

 for (int i = row-2; i >=0; i--) {
 for (int j = col-1; j >=0; j--) {

 long max = 0;
 for (int k = col-1; k >=0; k--){
 max= Math.max(points[i][j]+dp[i+1][k] - Math.abs(j-k),
max);
 }
 dp[i][j] = max;
 //maximumSum = Math.max(max, maximumSum);
 }
 }
 for (int i = col-1; i >= 0; i--) {

 maximumSum = Math.max(dp[0][i], maximumSum);
 }

 // System.out.print(Arrays.deepToString(dp));
 return maximumSum;
 }
}
```

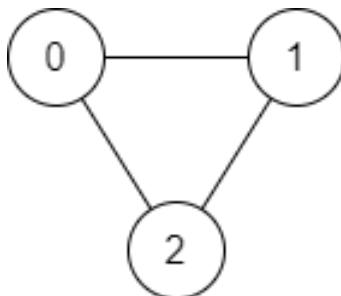
## 1971. Find if Path Exists in Graph ↗

There is a **bi-directional** graph with  $n$  vertices, where each vertex is labeled from  $0$  to  $n - 1$  (**inclusive**). The edges in the graph are represented as a 2D integer array  $\text{edges}$ , where each  $\text{edges}[i] = [u_i, v_i]$  denotes a bi-directional edge between vertex  $u_i$  and vertex  $v_i$ . Every vertex pair is connected by **at most one** edge, and no vertex has an edge to itself.

You want to determine if there is a **valid path** that exists from vertex `start` to vertex `end`.

Given `edges` and the integers `n`, `start`, and `end`, return `true` *if there is a valid path* from `start` to `end`, or `false` otherwise.

### Example 1:



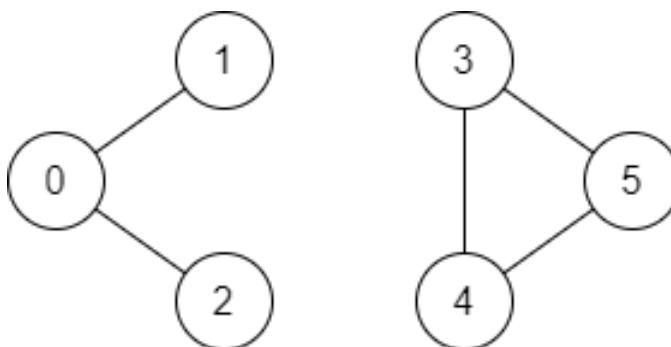
**Input:** `n = 3, edges = [[0,1],[1,2],[2,0]]`, `start = 0, end = 2`

**Output:** `true`

**Explanation:** There are two paths from vertex 0 to vertex 2:

- $0 \rightarrow 1 \rightarrow 2$
- $0 \rightarrow 2$

### Example 2:



**Input:** `n = 6, edges = [[0,1],[0,2],[3,5],[5,4],[4,3]]`, `start = 0, end = 5`

**Output:** `false`

**Explanation:** There is no path from vertex 0 to vertex 5.

### Constraints:

- $1 \leq n \leq 2 * 10^5$
- $0 \leq \text{edges.length} \leq 2 * 10^5$
- $\text{edges}[i].length == 2$
- $0 \leq u_i, v_i \leq n - 1$
- $u_i \neq v_i$
- $0 \leq \text{start}, \text{end} \leq n - 1$
- There are no duplicate edges.

- There are no self edges.

```

class Solution {
 public boolean validPath(int n, int[][] edges, int start, int end) {

 boolean[] visited = new boolean[n];
 List<List<Integer>> adj = new ArrayList<>();
 //creating AL of AL
 for (int i = 0; i < n; i++) {
 adj.add(new ArrayList<>());
 }
 //filling values
 for (int[] edge : edges){
 adj.get(edge[0]).add(edge[1]);
 adj.get(edge[1]).add(edge[0]);
 }

 return dfs(adj, start, end, visited);
 }

 public boolean dfs(List<List<Integer>> adj, int start, int end, boolean []
 visited){

 if(start == end) return true;

 visited[start] = true;
 for (int neighbour : adj.get(start)){
 if (!visited[neighbour]) {
 boolean isPath = dfs(adj, neighbour, end, visited); //storing ans in "isPath" is imp
 if(isPath) return true;
 }
 }
 return false;
 }
}

```