DATA SRUCTURES AND ALGORITHMS MEDIUM PROBLEMS

By Jani Ahamed Habeeb Mohamed

Table of Contents

1. ARRAYS AND STRINGS

1.1 Three Sum

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

Notice that the solution set must not contain duplicate triplets.

Example 1:

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

Output: [[-1,-1,2],[-1,0,1]]

Explanation:

nums[0] + nums[1] + nums[2] = (-1) + 0 + 1 = 0.

nums[1] + nums[2] + nums[4] = 0 + 1 + (-1) = 0.

nums[0] + nums[3] + nums[4] = (-1) + 2 + (-1) = 0.

The distinct triplets are [-1,0,1] and [-1,-1,2].

Notice that the order of the output and the order of the triplets does not matter.
```

Example 2:

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

Output: []

Explanation: The only possible triplet does not sum up to 0.
```

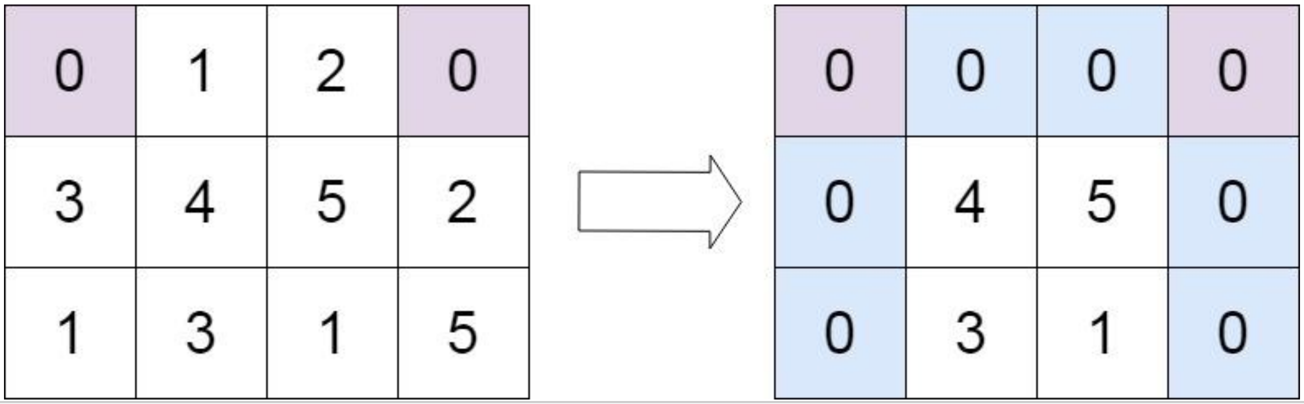
Example 3:

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

Output: [[0,0,0]]

Explanation: The only possible triplet sums up to 0.
```

Java:

```
class Solution {
    public List<List<Integer>> threeSum(int[] nums) {
        HashSet<Integer> hs = new HashSet<Integer>();
        HashMap<Integer,Integer> hm = new HashMap<Integer,Integer>();
        HashSet<List<Integer>> result = new HashSet<List<Integer>>();

        for (int i = 0; i < nums.length; i++) {
            if (hs.add(nums[i])) {
                for (int j = i+1; j < nums.length; j++) {
                    int target = -nums[i]-nums[j];
                    if (hm.containsKey(target) && hm.get(target) == i) {
                        List<Integer> entry = new ArrayList<Integer>();
                        entry.add(nums[i]);
                        entry.add(nums[j]);
                        entry.add(target);
                        Collections.sort(entry);
                        result.add(entry);
                    } else {
                        hm.put(nums[j],i);
                    }
                }
            }
        }
        return new ArrayList(result);
    }
}
```

1.2 **Set Matrix Zeros**



```
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]]
```

JAVA:

```
class Solution {
    public void setZeroes(int[][] matrix) {
        boolean first_col = false;
        boolean first_row = false;

        for (int i = 0; i < matrix.length; i++) {
            if (matrix[i][0] == 0) first_col = true;
        }

        for (int i = 0; i < matrix[0].length; i++) {
            if (matrix[0][i] == 0) first_row = true;
        }
        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;
                    matrix[0][j] = 0;
                }
            }
        }
        for (int i = 1; i < matrix.length; i++) {
            if(matrix[i][0] == 0) {
                for (int j = 1; j < matrix[0].length; j++) {
                    matrix[i][j] = 0;
                }
            }
        }
        for (int j = 1; j < matrix[0].length; j++) {
            if(matrix[0][j] == 0) {
                for (int i = 1; i < matrix.length; i++) {
                    matrix[i][j] = 0;
                }
            }
        }
        if (first_row == true) {
            for (int j = 0; j < matrix[0].length; j++) {
                matrix[0][j] = 0;
            }
        }
        if (first_col == true) {
            for (int i = 0; i < matrix.length; i++) {
                matrix[i][0] = 0;
            }
        }
    }
}
```

1.3 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"]]
```

JAVA:

```
class Solution {
    public List<List<String>> groupAnagrams(String[] strs) {
        List<List<String>> result = new ArrayList<List<String>>();
        HashMap<String,ArrayList<String>> hm = new HashMap<String, ArrayList<String>>();
        for (int i = 0; i < strs.length; i++) {
            StringBuilder sb = new StringBuilder();
            int count_map[] = new int[26];
            for (int j = 0; j < strs[i].length(); j++) {
                count_map[strs[i].charAt(j)-'a']++;
            }
            for (int k = 0; k < 26; k++) {
                sb.append(count_map[k]);
                sb.append("*");
            }
            if (!hm.containsKey(sb.toString())) {
                ArrayList<String> l = new ArrayList<String>();
                l.add(strs[i]);
                hm.put(sb.toString(), l);
            } else {
                ArrayList<String> l = hm.get(sb.toString());
                l.add(strs[i]);
                hm.put(sb.toString(),l);
            }
        }
        return new ArrayList(hm.values());
    }
}
```

1.4 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 = "bbbbb"
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 substr
ing.
```

JAVA:

```
class Solution {
    public int lengthOfLongestSubstring(String s) {
        if (s.length() == 0) return 0;
        else if (s.length() == 1) return 1;
        else {
            HashSet <Character> hs = new HashSet<Character>();
            int max_length = 0;
            for (int k = 0; k < s.length(); k++) {
                hs.add(s.charAt(k));
                for (int i = k+1; i < s.length(); i++) {
                    if (!hs.contains(s.charAt(i))) {
                        hs.add(s.charAt(i));
                        if (i == s.length()-1) {
                            max_length = Math.max(max_length,hs.size());
                            hs.clear();
                        }
                    }
                    else {
                        max_length = Math.max(max_length,hs.size());
                        hs.clear();
                        break;
                    }
                }
            }
            return max_length;
        }
    }
}
```

1.5 Longest Palindromic Substring

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

Example 1:

```
Input: s = "babad"

Output: "bab"

Explanation: "aba" is also a valid answer.
```

Example 2:

```
Input: s = "cbbd"

Output: "bb"
```

JAVA:

```
class Solution {
    public boolean check_palindrome(String s, int start, int end) {
        int i = start;
        int j = end;
        while (start <= end) {
            if (s.charAt(start) != s.charAt(end)) {
                return false;
            } else {
                start++;
                end--;
            }
        }
        return true;
    }
    public String longestPalindrome(String s) {
        int i = 0;
        int j = 0;
        int length = 0;
        String result_string = new String();
        for (i = 0 ; i < s.length(); i++) {
            for (j = s.length()-1; j >=0; j--) {
                if (s.charAt(i) == s.charAt(j)) {
                    if (check_palindrome(s, i, j)) {
                        length = Math.max(length,j-i+1);
                        if (j-i+1 == length) {
                            result_string = s.substring(i,j+1);
                        }

                        break;
                    }

                }
            }
        }
        return result_string;
    }
}
```

1.6 Increasing Triplet Subsequence

Given an integer array `nums`, return `true` *if there exists a triple of indices* `(i, j, k)` *such that* `i < j < k` *and* `nums[i] < nums[j] < 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 nums[3] == 0 < nums[4] == 4 < nu
ms[5] == 6.
```

JAVA:

```
class Solution {
    public boolean increasingTriplet(int[] nums) {
        int first = Integer.MAX_VALUE;
        int second = Integer.MAX_VALUE;
        for (int i = 0; i < nums.length; i++) {
            if (nums[i] <= first) first = nums[i];
            else if (nums[i] <= second) second = nums[i];
            else {
                return true;
            }
        }
        return false;
    }
}
```

1.7 Missing Ranges

You are given an inclusive range `[lower, upper]` and a **sorted unique** integer array `nums`, where all elements are within the inclusive range.

A number `x` is considered **missing** if `x` is in the range `[lower, upper]` and `x` is not in `nums`.

Return *the **shortest sorted** list of ranges that **exactly covers all the missing numbers***. That is, no element of `nums` is included in any of the ranges, and each missing number is covered by one of the ranges.

Example 1:

```
Input: nums = [0,1,3,50,75], lower = 0, upper = 99

Output: [[2,2],[4,49],[51,74],[76,99]]

Explanation: The ranges are:

[2,2]

[4,49]

[51,74]

[76,99]
```

Example 2:

```
Input: nums = [-1], lower = -1, upper = -1

Output: []

Explanation: There are no missing ranges since there are no missing numbers.
```

JAVA:

```
class Solution {
    public List<List<Integer>> findMissingRanges(int[] nums, int lower, int upper) {
        int n = nums.length;
        List<List<Integer>> missingRanges = new ArrayList<>();

        if (n == 0) {
            ArrayList<Integer> row = new ArrayList<Integer>();
            row.add(lower);
            row.add(upper);
            missingRanges.add(row);
            return missingRanges;
        }
        // Check for any missing numbers between the lower bound and nums[0].
        if (lower < nums[0]) {
            ArrayList<Integer> row = new ArrayList<Integer>();
            row.add(lower);
            row.add(nums[0]-1);
            missingRanges.add(row);
        }

        // Check for any missing numbers between successive elements of nums.
        for (int i = 0; i < n - 1; i++) {
            if (nums[i + 1] - nums[i] <= 1) {
                continue;
            }
            ArrayList<Integer> row = new ArrayList<Integer>();
            row.add(nums[i]+1);
            row.add(nums[i+1]-1);
            missingRanges.add(row);
        }

        // Check for any missing numbers between the last element of nums and the upper bound.
        if (upper > nums[n - 1]) {
            ArrayList<Integer> row = new ArrayList<Integer>();
            row.add(nums[n-1]+1);
            row.add(upper);
            missingRanges.add(row);
        }

        return missingRanges;
    }
}
```

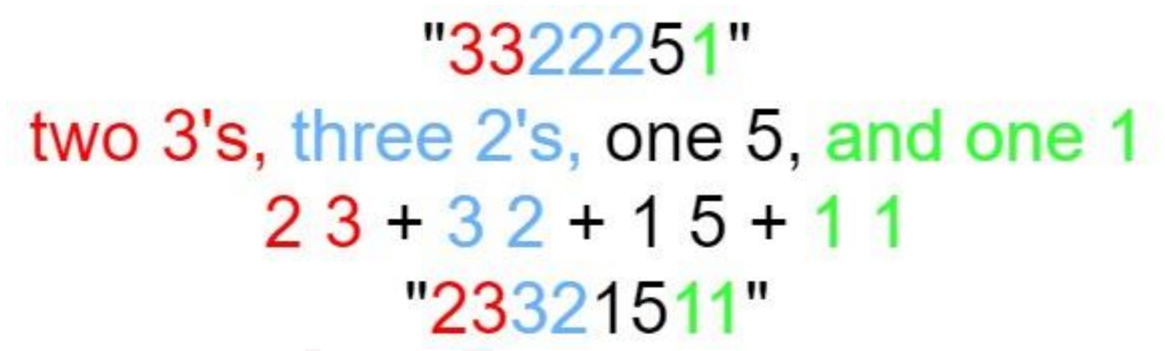
1.8 Count And Say

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 substrings such that each substring contains exactly **one** unique digit. Then for each substring, say the number of digits, then say the digit. Finally, concatenate every said digit.

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



Given a positive integer `n`, return *the* `n`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"
```

<u>JAVA:</u>

```
class Solution {
    public String countAndSay(int n) {
        String result = "1";

        for (int i = 2; i <= n; i++) {
            StringBuilder temp = new StringBuilder();
            Character prev = result.charAt(0);
            int count = 1;
            for (int j = 1; j < result.length(); j++) {
                Character curr = result.charAt(j);
                if (prev == curr) {
                    count++;
                }
                else {
                    temp.append(count);
                    temp.append(prev);
                    prev = curr;
                    count = 1;
                }
            }
            temp.append(count);
            temp.append(prev);
            result = temp.toString();
        }
        return result;
    }
}
```

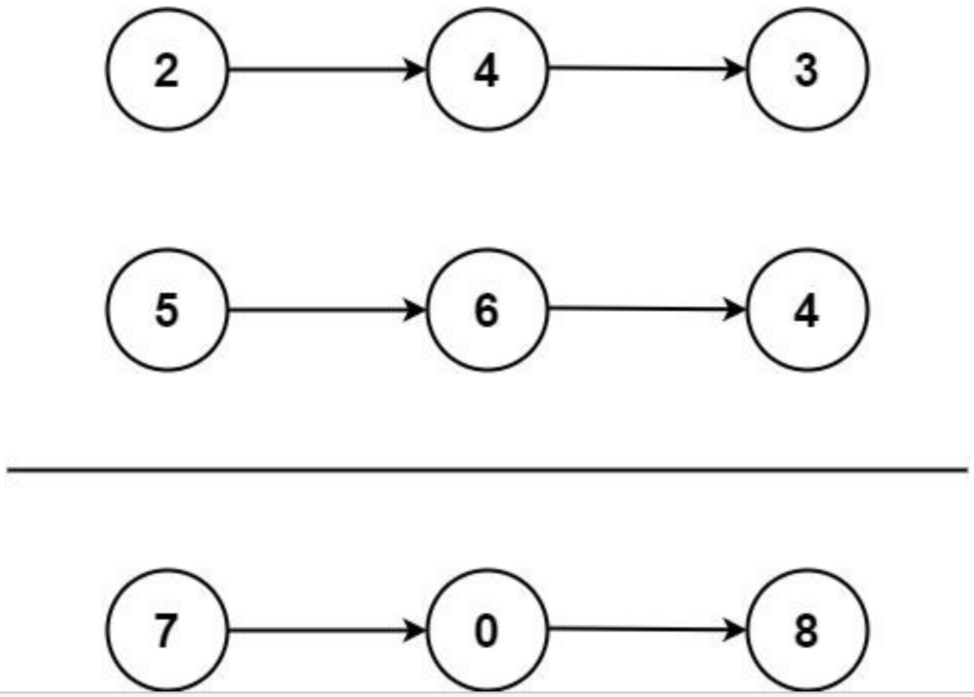
2. LINKED LISTS

2.1 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]
```

C:

```
struct ListNode* addTwoNumbers(struct ListNode* l1, struct ListNode* l2){
    struct ListNode* head = NULL;
    struct ListNode* tail = NULL;
    int carry = 0;
    while (l1 != NULL || l2 != NULL || carry != 0) {
        int temp = 0;
        if (l1 != NULL) {
            temp+=l1->val;
            l1 = l1->next;
        }
        if (l2 != NULL) {
            temp+=l2->val;
            l2 = l2->next;
        }
        if (carry != 0) {
            temp+=carry;
            carry = 0;
        }
        int value = temp%10;
        carry = temp/10;
        struct ListNode* entry = (struct ListNode*) malloc (sizeof(struct ListNode));
        entry->val=value;
        entry->next = NULL;
        if( head == NULL) {
            head = entry;
            tail = entry;
        }
        else {
            tail->next = entry;
            tail = tail->next;
        }
    }
    return head;
}
```

2.2 Odd Even Linked List

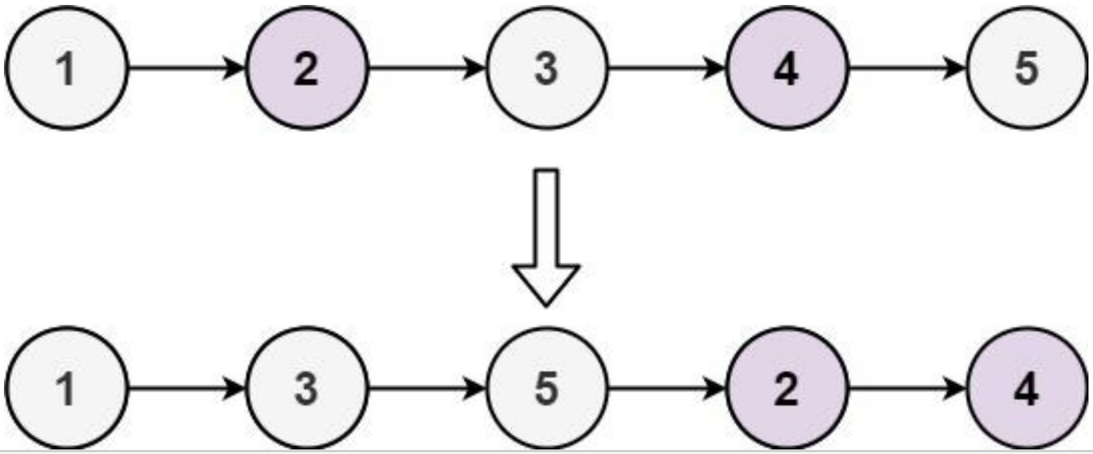
Given the head of a singly linked list, group all the nodes with odd indices together followed by the nodes with even indices, and return *the reordered list*.

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.

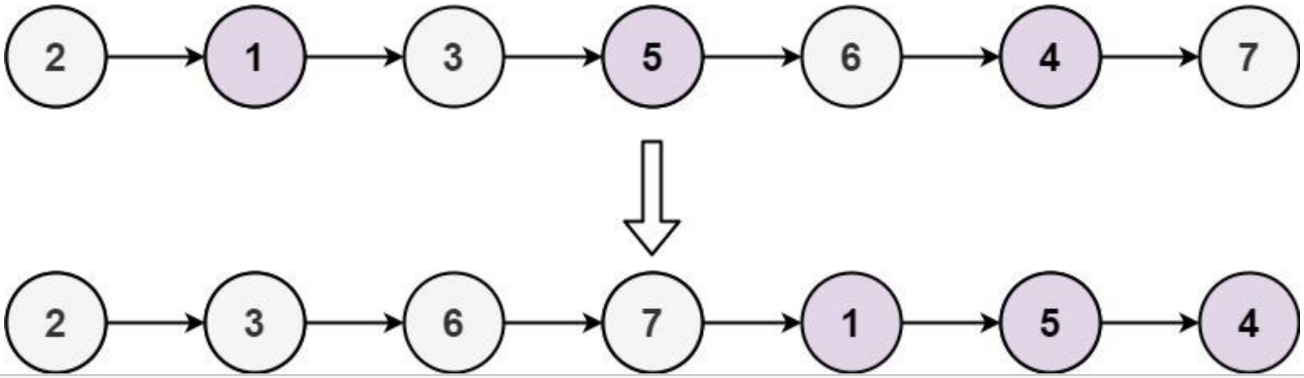
You must solve the problem in `O(1)` extra space complexity and `O(n)` time complexity.

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]

: 21
```

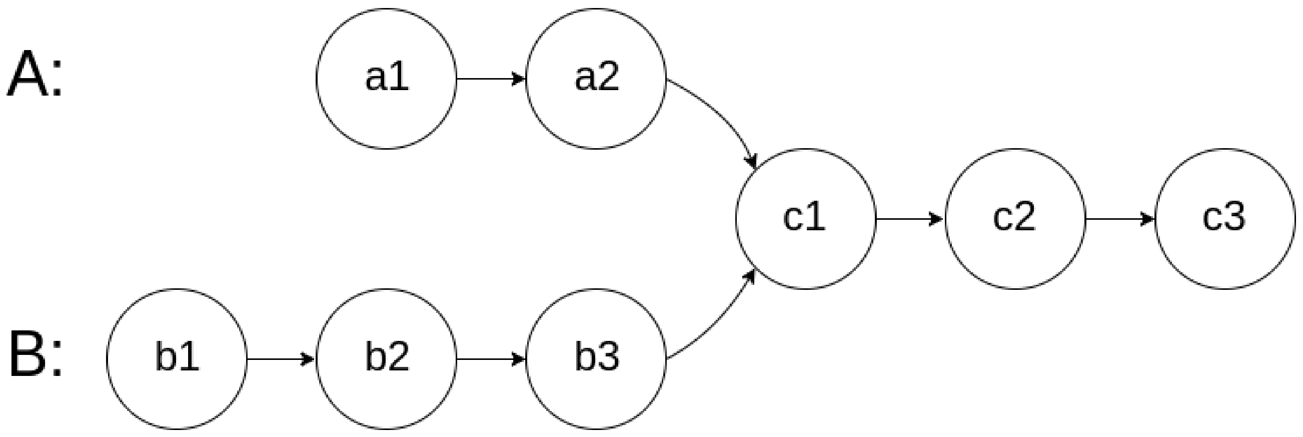
C:

```
struct ListNode* oddEvenList(struct ListNode* head){
    if (head == NULL) return NULL;
    struct ListNode* odd = head;
    struct ListNode *even = head->next;
    struct ListNode *evenhead = even;
    while (even!= NULL && even->next !=NULL) {
        odd->next = even->next;
        odd = odd->next;
        even->next=odd->next;
        even = even->next;
    }
    odd->next = evenhead;
    return head;
}
```

2.3 <u>Intersection Of Two Linked Lists</u>

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.

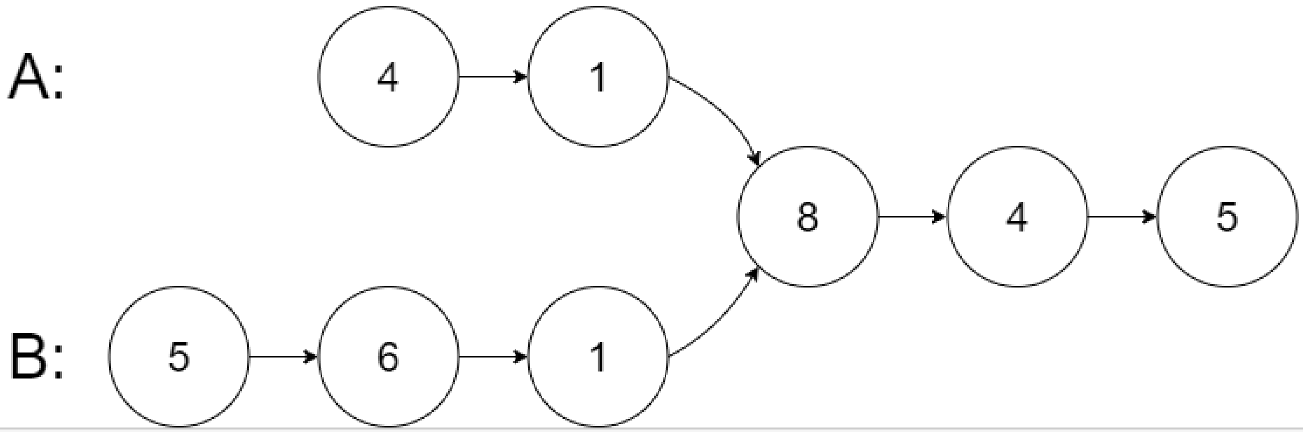
Custom Judge:

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.

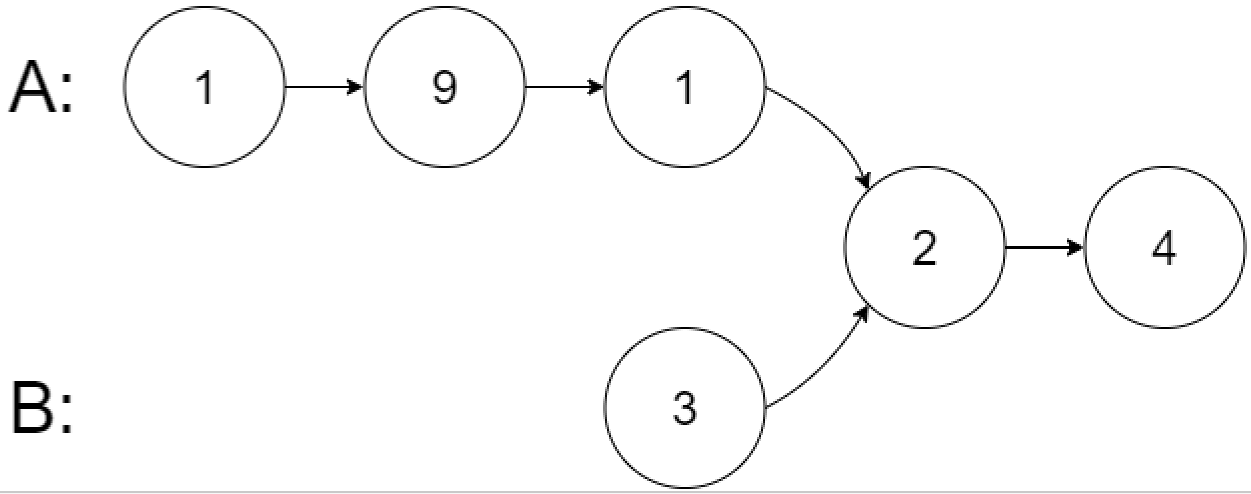
The judge will then create the linked structure based on these inputs and pass the two heads, `headA` and `headB` to your program. If you correctly return the intersected node, then your solution will be **accepted**.

Example 1:



```
Input: intersectVal = 8, listA = [4,1,8,4,5], listB = [5,6,1,8,4,5], skipA = 2, skip
B = 3

Output: Intersected at '8'

Explanation: The intersected node's value is 8 (note that this must not be 0 if the
two lists intersect).

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]. There are 2 nodes before the intersected node in A; There are 3 nodes before
 the intersected node in B.

- Note that the intersected node's value is not 1 because the nodes with value 1 in
A and B (2nd node in A and 3rd node in B) are different node references. In other word
s, they point to two different locations in memory, while the nodes with value 8 in
A and B (3rd node in A and 4th node in B) point to the same location in memory.
```

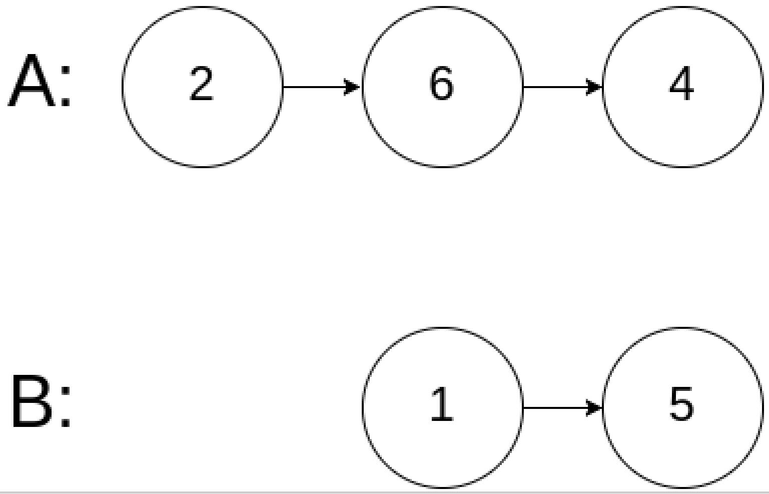
Example 2:



```
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 if the
two lists intersect).

From the head of A, it reads as [1,9,1,2,4]. From the head of B, it reads as [3,2,4]
. There are 3 nodes before the intersected node in A; There are 1 node before the in
tersected node in B.
```

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 reads a
s [1,5]. Since the two lists do not intersect, intersectVal must be 0, while skipA a
nd skipB can be arbitrary values.

Explanation: The two lists do not intersect, so return null.
```

C:

```
struct ListNode *getIntersectionNode(struct ListNode *headA, struct ListNode *headB) {
    struct ListNode *first = headA;
    struct ListNode *second = headB;
    while (first!=second) {
        first = (first==NULL)?headB:first->next;
        second = (second==NULL)?headA:second->next;

    }

    return first;
}
```

3. SORTING AND SEARCHING

3.1 Sort Colors

Given an array `nums` with `n` objects colored red, white, or blue, sort them **in-place** 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]
```

C:

```
void swap(int *nums, int a, int b) {
    int temp = nums[a];
    nums[a] = nums[b];
    nums[b] = temp;
}
void sortColors(int* nums, int numsSize){
    int left = 0;
    int right = numsSize-1;
    int mid = 0;
    while (mid <= right) {
        if (nums[mid] == 0) {
            swap(nums, left, mid);
            left++;
            mid++;
        }
        else if (nums[mid] == 2) {
            swap(nums, right, mid);
            right--;
        }
        else {
            mid++;
        }
    }
}
```

3.2 Top K Frequent Elementsd

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]
```

JAVA:

```
class Solution {
    public int[] topKFrequent(int[] nums, int k) {
        HashMap<Integer, Integer> hm = new HashMap<Integer,Integer>();

        if (k == nums.length) return nums;

        for (int i = 0; i < nums.length; i++) {
            hm.put(nums[i],hm.getOrDefault(nums[i],0)+1);
        }
        Queue<Integer> q = new PriorityQueue<Integer>((a,b)->hm.get(a)-hm.get(b));
        for (Map.Entry<Integer,Integer> entry:hm.entrySet()) {
            System.out.println(entry.getKey()+":"+entry.getValue());
            q.add(entry.getKey());
            if (q.size() > k) {
                q.poll();

            }
        }

        System.out.println(q);
        int result[] = new int[k];
        int c = 0;
        while (c < k) {
            result[c] = q.poll();
            c++;
        }


        return result;
    }
}
```

3.3 Kth Largest Element in Array

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

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

Can you solve it without sorting?

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
```

JAVA:

```
class Solution {
    public int findKthLargest(int[] nums, int k) {
        Queue<Integer> q = new PriorityQueue<Integer>();
        for (int n : nums) {
            q.add(n);
            if (q.size() > k) q.poll();
        }
        System.out.println(q);
        return q.poll();
    }
}
```

3.4 Find Peak Element

A peak element is an element that is strictly greater than its neighbors.

Given a **0-indexed** 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] = -∞`. In other words, an element is always considered to be strictly greater than a neighbor that is outside the array.

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 number 2.
```

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 element i
s 2, or index number 5 where the peak eleme
```

C:

```
int findPeakElement(int* nums, int numsSize) {
    if (numsSize == 1) return 0;

    // Check the first and last elements as special cases
    if (nums[0] > nums[1]) return 0;
    if (nums[numsSize - 1] > nums[numsSize - 2]) return numsSize - 1;

    // Perform binary search to find a peak element
    int left = 1;
    int right = numsSize - 2;

    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (nums[mid] > nums[mid - 1] && nums[mid] > nums[mid + 1]) {
            // Found a peak element
            return mid;
        } else if (nums[mid - 1] > nums[mid]) {
            // Search on the left side
            right = mid - 1;
        } else {
            // Search on the right side
            left = mid + 1;
        }
    }

    return -1;  // No peak element found
}
```

JAVA:

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

3.5 Search For A Range

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]
```

C:

```
/**
 * Note: The returned array must be malloced, assume caller calls free().
 */
int search(int* nums, int numsSize, int target, bool first) {
    int left = 0;
    int mid = 0;
    int right = numsSize-1;
    while (left <= right) {
        mid = left + (right-left)/2;
        if (first == true) {
            if (nums[mid] == target) {
                if (mid == 0 || nums[mid-1] != target) {
                    return mid;
                }
                                        right = mid-1;
            }
            else if (nums[mid] < target) {
                left = mid+1;
            }
            else {
                right = mid-1;
            }
        } else {
            if (nums[mid] == target) {
                if (mid == numsSize - 1 || nums[mid+1] != target) {
                    return mid;
                }
                                        left = mid+1;
            }
            else if (nums[mid] < target) {
                   left = mid+1;
            }
            else {
                right = mid-1;
            }
        }

    }
    return -1;
}

int* searchRange(int* nums, int numsSize, int target, int* returnSize){
    int* return_array = (int *)malloc(sizeof(int)*2);
    *returnSize = 2;
    int first_occurrence = search(nums, numsSize, target, true);
    int last_occurrence = search(nums, numsSize, target, false);
    return_array[0] = first_occurrence;
    return_array[1] = last_occurrence;
    return return_array;
}
```

JAVA:

```
class Solution {
    public int[] searchRange(int[] nums, int target) {
        int left = 0;
        int right = nums.length-1;
        int start = -1;
        int end = -1;

        while (left < nums.length) {
            if (nums[left] != target) {
                left++;
            }
            else {
                if (start == -1) {
                    start = left;
                }

                left++;
                if (end!= -1) break;
            }
            if (nums[right] != target) {
                if (end != -1) {

                    start = right+1;
                    break;
                }
                right--;
            }
            else {
                if (end == -1) {
                    end = right;
                }

                right--;
                if (start!= -1) break;
            }
        }
        return new int[]{start,end};
    }
```

}

3.6 Merge Intervals

Given an array of `intervals` where `intervals[i] = [starti, endi]`, 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] overlap, 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.
```

JAVA:

```
class Solution {
    public int[][] merge(int[][] intervals) {
        // Check for invalid or empty input
        if (intervals == null || intervals.length == 0) {
            return new int[0][];
        }

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

        int start = intervals[0][0];
        int end = intervals[0][1];

        for (int i = 1; i < intervals.length; i++) {
            int currentStart = intervals[i][0];
            int currentEnd = intervals[i][1];

            if (currentStart <= end) {
                // Overlapping intervals, merge them
                end = Math.max(end, currentEnd);
            } else {
                // Non-overlapping interval, add the current merged interval to the result list
                ArrayList<Integer> mergedInterval = new ArrayList<>();
                mergedInterval.add(start);
                mergedInterval.add(end);
                result.add(mergedInterval);

                // Update start and end with the current interval's values
                start = currentStart;
                end = currentEnd;
            }
        }

        // Add the last interval to the result list
        ArrayList<Integer> mergedInterval = new ArrayList<>();
        mergedInterval.add(start);
        mergedInterval.add(end);
        result.add(mergedInterval);

        // Convert the result list to a 2D array
        int[][] mergedIntervalsArray = new int[result.size()][2];
        for (int i = 0; i < result.size(); i++) {
            mergedIntervalsArray[i][0] = result.get(i).get(0);
            mergedIntervalsArray[i][1] = result.get(i).get(1);
        }

        return mergedIntervalsArray;
    }
}
```

3.7 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

.
```

C:

```c
int search(int* nums, int numsSize, int target) {
    int left = 0;
    int right = numsSize - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (nums[mid] == target) {
            return mid;  // Target found
        }

        if (nums[left] <= nums[mid]) {
            // Left half is sorted
            if (nums[left] <= target && target < nums[mid]) {
                right = mid - 1;  // Search in the left half
            } else {
                left = mid + 1;  // Search in the right half
            }
        } else {
            // Right half is sorted
            if (nums[mid] < target && target <= nums[right]) {
                left = mid + 1;  // Search in the right half
            } else {
                right = mid - 1;  // Search in the left half
            }
        }
    }

    return -1;  // Target not found
}
```

JAVA:

```
class Solution {
    public int search(int[] nums, int target) {
                int n = nums.length;
        int left = 0, right = n - 1;

        while (left <= right) {
            int mid = left + (right - left) / 2;

            // Case 1: find target
            if (nums[mid] == target) {
                return mid;
            }

            // Case 2: subarray on mid's left is sorted
            else if (nums[mid] >= nums[left]) {
                if (target >= nums[left] && target < nums[mid]) {
                    right = mid - 1;
                } else {
                    left = mid + 1;
                }
            }

            // Case 3: subarray on mid's right is sorted
            else {
                if (target <= nums[right] && target > nums[mid]) {
                    left = mid + 1;
                } else {
                    right = mid - 1;
                }
            }
        }

        return -1;
    }
}
```

3.8 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

.
```

C:

```
int search(int* nums, int numsSize, int target) {
    int left = 0;
    int right = numsSize - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (nums[mid] == target) {
            return mid;  // Target found
        }

        if (nums[left] <= nums[mid]) {
            // Left half is sorted
            if (nums[left] <= target && target < nums[mid]) {
                right = mid - 1;  // Search in the left half
            } else {
                left = mid + 1;  // Search in the right half
            }
        } else {
            // Right half is sorted
            if (nums[mid] < target && target <= nums[right]) {
                left = mid + 1;  // Search in the right half
            } else {
                right = mid - 1;  // Search in the left half
            }
        }
    }

    return -1;  // Target not found
}
```

JAVA:

```
class Solution {
    public int search(int[] nums, int target) {
                int n = nums.length;
        int left = 0, right = n - 1;

        while (left <= right) {
            int mid = left + (right - left) / 2;

            // Case 1: find target
            if (nums[mid] == target) {
                return mid;
            }

            // Case 2: subarray on mid's left is sorted
            else if (nums[mid] >= nums[left]) {
                if (target >= nums[left] && target < nums[mid]) {
                    right = mid - 1;
                } else {
                    left = mid + 1;
                }
            }

            // Case 3: subarray on mid's right is sorted
            else {
                if (target <= nums[right] && target > nums[mid]) {
                    left = mid + 1;
                } else {
                    right = mid - 1;
                }
            }
        }

        return -1;
    }
}
```

3.9 Meeting Rooms II

Given an array of meeting time intervals `intervals` where `intervals[i] = [start_i, end_i]`, 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
```

JAVA:

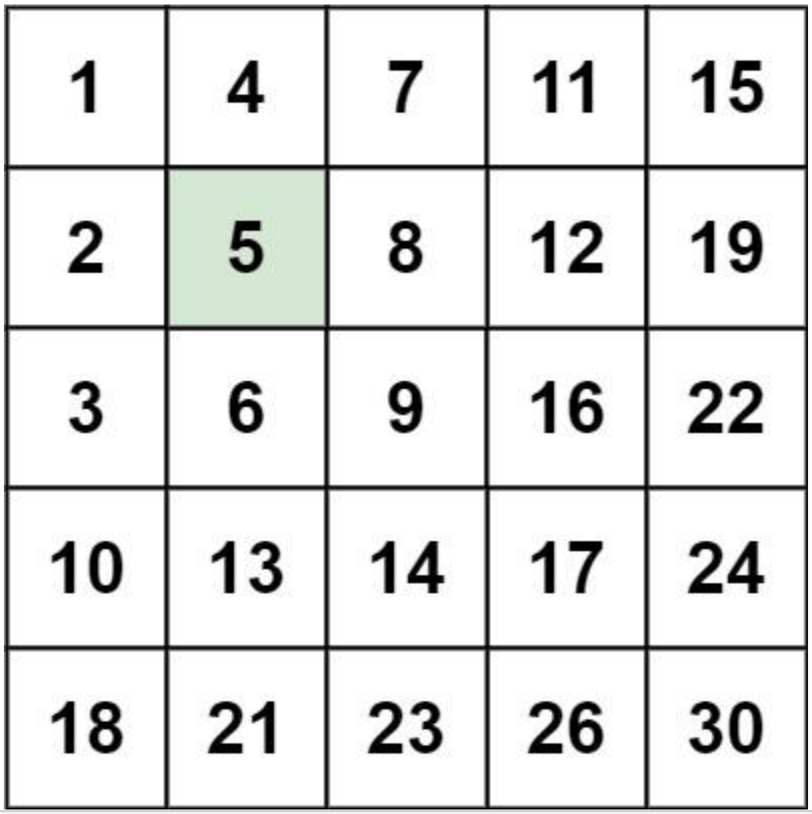
```
class Solution {
    public int minMeetingRooms(int[][] intervals) {
        Queue<Integer> q = new PriorityQueue<Integer>((a,b)->(a-b));
        for (int i = 0; i < intervals.length; i++)
        System.out.println(intervals[i][0]+":"+intervals[i][1]);
        Arrays.sort(intervals, (a,b)->(a[0]-b[0]));
        for (int i = 0; i < intervals.length; i++)
        System.out.println(intervals[i][0]+":"+intervals[i][1]);
        q.add(intervals[0][1]);
        for (int i = 1; i < intervals.length; i++) {
            if (intervals[i][0] >= q.peek()) {
                q.poll();
            }
            q.add(intervals[i][1]);
        }

        return q.size();
    }
}
```

3.10 Search A 2D Matrix II

Write an efficient algorithm that searches for a value `target` in an `m x n` integer matrix `matrix`. This 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.

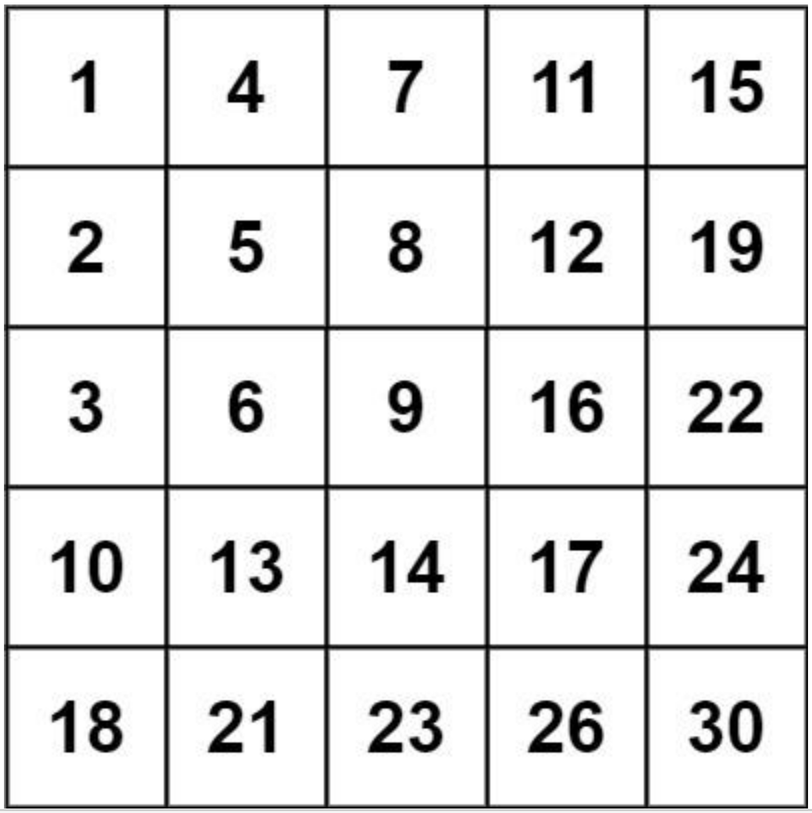
Example 1:



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],[18,21,2
3,26,30]], target = 5

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],[18,21,2
3,26,30]], target = 20

Output: false
```

C:

```
bool searchMatrix(int** matrix, int matrixSize, int* matrixColSize, int target) {
    int row = matrixSize - 1;
    int col = 0;

    while (row >= 0 && col < *matrixColSize) {
        if (matrix[row][col] > target) {
            row--;
        } else if (matrix[row][col] < target) {
            col++;
        } else { //a found it
            return true;
        }
    }

    return false;
}
```

JAVA:

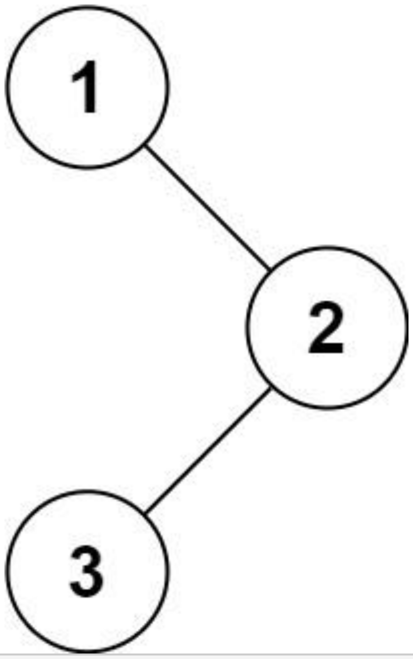
```
class Solution {
    public boolean searchMatrix(int[][] matrix, int target) {
        for (int i = matrix.length-1; i >= 0; i--) {
            for (int j = 0; j < matrix[0].length; j++) {
                if (target == matrix[i][j]) return true;
                if (target < matrix[i][j]) {
     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

4. TREES AND GRAPHS

4.1 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]
```

C:

static int k = 0;

```
void helper(struct TreeNode* root, int **result) {
    if (root != NULL) {
        helper(root->left, result);

        // Reallocate memory for the result array before adding a new value
        *result = (int*)realloc(*result, sizeof(int) * (k + 1));

        (*result)[k] = root->val;
        k++;
        helper(root->right, result);
    }
}

int* inorderTraversal(struct TreeNode* root, int* returnSize) {
    if (root == NULL) {
        *returnSize = 0;
        return NULL;
    }
    int *result = NULL;
    helper(root, &result);

    *returnSize = k;
    return result;
}
```

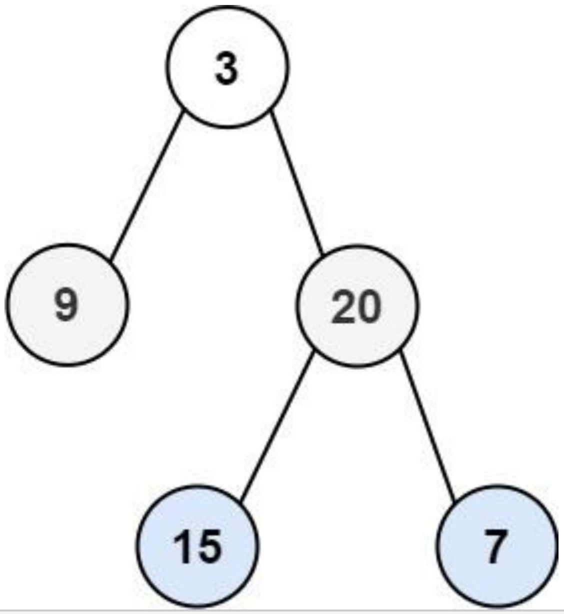
JAVA:

```
/**
 * 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<Integer> inorderTraversal(TreeNode root) {
        List<Integer> result = new ArrayList<Integer>();
        helper(root,result);
        return result;
    }
    public void helper(TreeNode root, List<Integer> result) {
        if (root!= null) {
            helper(root.left,result);
            result.add(root.val);
            helper(root.right,result);
        }
    }
}
```

4.2 Binary Tree Zig Zag 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: []
```

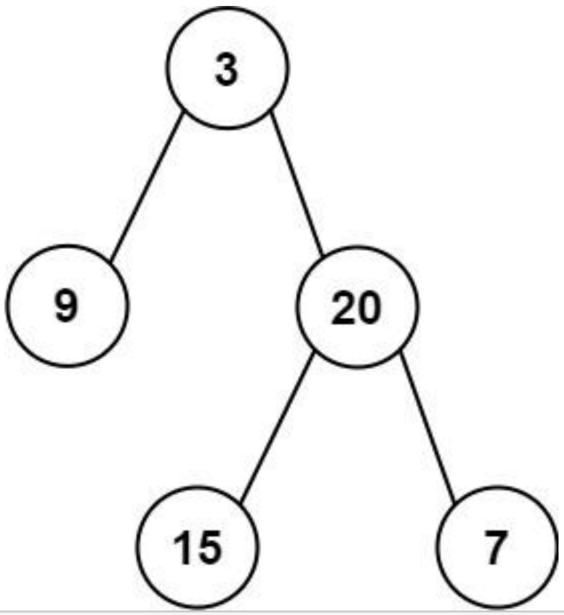
JAVA:

```
class Solution {
    public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
        List<List<Integer>> levels = new ArrayList<List<Integer>>();
        if (root == null) return levels;

        Queue<TreeNode> queue = new LinkedList<TreeNode>();
        queue.add(root);
        int level = 0;
        while ( !queue.isEmpty() ) {
            // start the current level
            levels.add(new ArrayList<Integer>());

            // number of elements in the current level
            int level_length = queue.size();

            for(int i = 0; i < level_length; ++i) {
                TreeNode node = queue.remove();

                // fulfill the current level
                levels.get(level).add(node.val);

                // add child nodes of the current level
                // in the queue for the next level
                if (node.left != null) queue.add(node.left);
                if (node.right != null) queue.add(node.right);
            }
            // go to next level
            level++;
        }
        for (int i = 0; i < levels.size(); i++) {
            if (i % 2 == 1) {
                Collections.reverse(levels.get(i));
            }
        }
        return levels;
    }
}
```

4.3 Construct Binary Tree From PreOrder And Inroder Traversal

Given two integer arrays `preorder` and `inorder` where `preorder` is the preorder traversal of a binary tree and `inorder` is the inorder traversal of the same tree, construct and return *the binary tree*.

Example 1:



```
Input: preorder = [3,9,20,15,7], inorder = [9,3,15,20,7]

Output: [3,9,20,null,null,15,7]
```

Example 2:

```
Input: preorder = [-1], inorder = [-1]

Output: [-1]
```

<u>JAVA:</u>

```
class Solution {
public TreeNode buildTree(int[] preorder, int[] inorder) {
    return helper(0, 0, inorder.length - 1, preorder, inorder);
}

public TreeNode helper(int preStart, int inStart, int inEnd, int[] preorder, int[] inorder) {
    if(inStart > inEnd) return null;

    TreeNode root = new TreeNode(preorder[preStart]);
    int inorderindex = 0;
    for (int i = inStart; i <= inEnd; i++) {
        if (root.val == inorder[i]) inorderindex = i;
    }
    root.left = helper(preStart + 1, inStart, inorderindex-1, preorder, inorder);
    root.right = helper(preStart + inorderindex - inStart + 1, inorderindex+1, inEnd, preorder, inorder);

    return root;
}
}

//Modified JAVA Code for PostOrder/Inorder Traversal Tree Construction
class Solution {
    public TreeNode buildTree(int[] inorder, int[] postorder) {
        return helper(postorder.length - 1, 0, inorder.length - 1, postorder, inorder);
    }

    public TreeNode helper(int postEnd, int inStart, int inEnd, int[] postorder, int[] inorder) {
        if (inStart > inEnd) return null;

        TreeNode root = new TreeNode(postorder[postEnd]);
        int inorderIndex = inStart;
        while (inorder[inorderIndex] != root.val) {
            inorderIndex++;
        }

        root.right = helper(postEnd - 1, inorderIndex + 1, inEnd, postorder, inorder);
        root.left = helper(postEnd - (inEnd - inorderIndex) - 1, inStart, inorderIndex - 1, postorder, inorder);

        return root;
    }
}
```

4.4 <u>Populating Next Right Pointers In Each Node</u>

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:

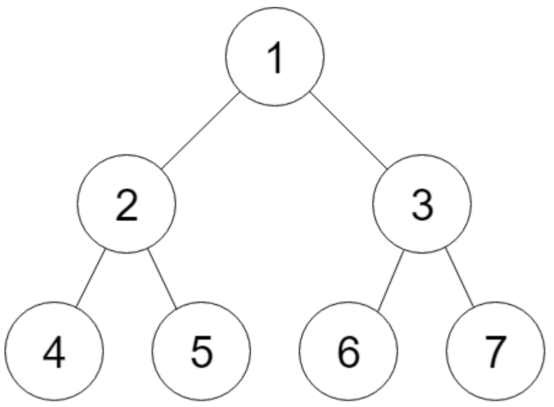


Figure A

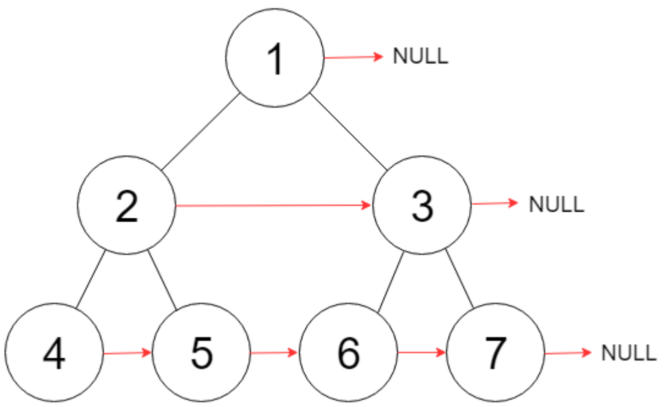


Figure B

```
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 should po
pulate each next pointer to point to its next right node, just like in Figure B. The
 serialized output is in level order as connected by the next pointers, with '#' sig
nifying the end of each level.
```

Example 2:

```
Input: root = []

Output: []
```

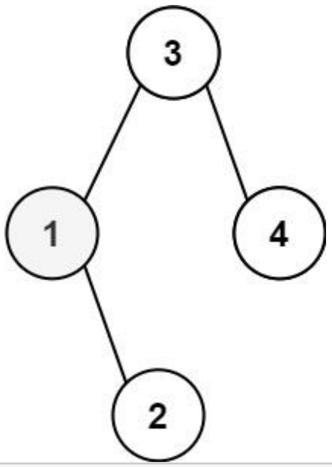
JAVA:

```
class Solution {
    public Node connect(Node root) {
        List<List<Integer>> result = new ArrayList<List<Integer>>();
        Queue<Node> q = new LinkedList<Node>();
        int level = 0;
        if (root == null) return null;
        q.add(root);
        while (!q.isEmpty()) {
            int level_size = q.size();
            result.add(new ArrayList<Integer>());
            for (int i = 0; i < level_size; i++) {
                Node node = q.remove();
                result.get(level).add(node.val);
                if (i < level_size-1) node.next = q.peek();
                if (node.left != null) q.add(node.left);
                if (node.right != null) q.add(node.right);
            }
            level++;
        }
        return root;
    }
}
```

4.5 Kth Smallest Element In A BST

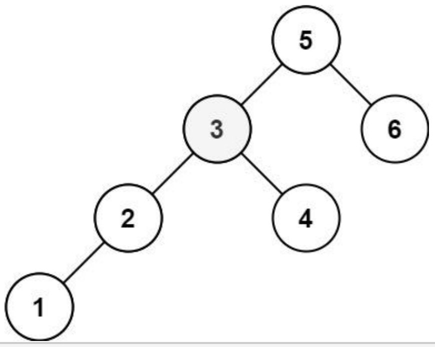
Given the `root` of a binary search tree, and an integer `k`, return *the* `k`th *smallest value (**1-indexed**) of all the values of the nodes in the tree.*

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
```

JAVA:

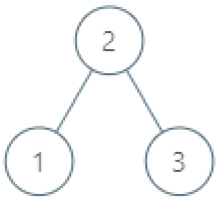
```
class Solution {
    public ArrayList<Integer> inorder(TreeNode root, ArrayList<Integer> arr) {
        if (root == null) return arr;
        inorder(root.left, arr);
        arr.add(root.val);
        inorder(root.right, arr);
        return arr;
    }

    public int kthSmallest(TreeNode root, int k) {
        ArrayList<Integer> nums = inorder(root, new ArrayList<Integer>());
        return nums.get(k - 1);
    }
}
```

4.6 InOrder Successor In A BST

Given the `root` of a binary search tree and a node `p` in it, return *the in-order successor of that node in the BST*. If the given node has no in-order successor in the tree, return `null`.

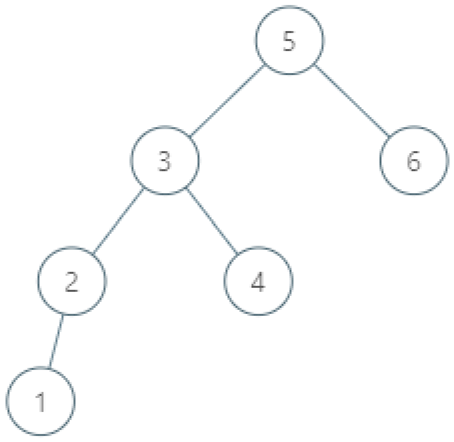
The successor of a node `p` is the node with the smallest key greater than `p.val`.

Example 1:



```
Input: root = [2,1,3], p = 1

Output: 2

Explanation: 1's in-order successor node is 2. Note that both p and the return value
 is of TreeNode type.
```

Example 2:



```
Input: root = [5,3,6,2,4,null,null,1], p = 6

Output: null

Explanation: There is no in-order successor of the current node, so the answer is nu
ll.
```

JAVA:

```
class Solution {
    public TreeNode inorderSuccessor(TreeNode root, TreeNode p) {
        TreeNode successor = null;

        while (root != null) {

            if (p.val >= root.val) {
                root = root.right;
            } else {
                successor = root;
                root = root.left;
            }
        }

        return successor;
    }

}
```

4.7 Number Of Islands

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

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

Example 1:

```
Input: grid = [

  ["1","1","1","1","0"],

  ["1","1","0","1","0"],

  ["1","1","0","0","0"],

  ["0","0","0","0","0"]

]

Output: 1
```

Example 2:

```
Input: grid = [

  ["1","1","0","0","0"],

  ["1","1","0","0","0"],

  ["0","0","1","0","0"],

  ["0","0","0","1","1"]

]

Output: 3
```

JAVA:

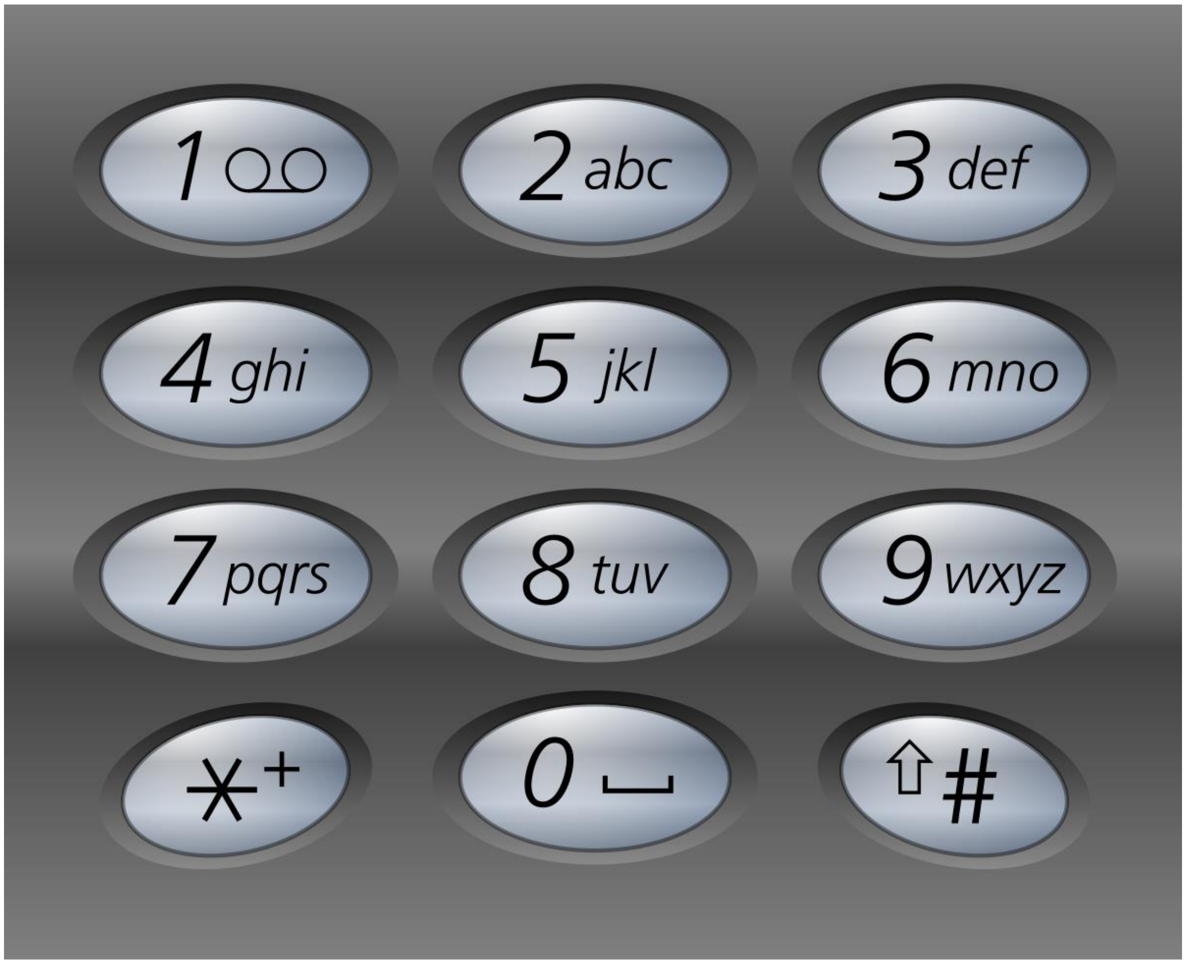
```
class Solution {
    public int numIslands(char[][] grid) {
        int num_of_islands = 0;
        if (grid == null || grid.length == 0) {
                return 0;
        }
        for (int i = 0; i < grid.length; i++) {
            for (int j = 0; j < grid[i].length; j++) {
                if (grid[i][j] == '1') {
                    num_of_islands++;
                    evaluate_grid(grid, i, j, grid.length, grid[i].length);
                }
            }
        }
        return num_of_islands;
    }
    public void evaluate_grid(char[][] grid, int i, int j, int row, int col) {
        if (i < 0 || j < 0 || i >= row || j >= col || grid[i][j] != '1') return;
        grid[i][j] = '2';
        evaluate_grid(grid,i-1,j,row,col);
        evaluate_grid(grid,i+1,j,row,col);
        evaluate_grid(grid,i,j-1,row,col);
        evaluate_grid(grid,i,j+1,row,col);
    }
}
```

5. BACKTRACKING

5.1 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 digits 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"]
```

JAVA:

```
class Solution {
    HashMap<Character, String> hm = new HashMap<Character, String>();

    public void generate_words(String current_word, String digits, int current_index, List<String> result) {
        if (current_index == digits.length()) {
            result.add(current_word);
            return;
        }

        String seq = hm.get(digits.charAt(current_index));

        for (int i = 0; i < seq.length(); i++) {
            generate_words(current_word+seq.charAt(i), digits, current_index+1, result);
        }

    }
    public void helper(String digits, List<String> result) {
        generate_words("", digits, 0, result);
    }
    public List<String> letterCombinations(String digits) {
        List<String> result = new ArrayList<String>();
        hm.put('2',"abc");
        hm.put('3',"def");
        hm.put('4',"ghi");
        hm.put('5',"jkl");
        hm.put('6',"mno");
        hm.put('7',"pqrs");
        hm.put('8',"tuv");
        hm.put('9',"wxyz");
        if (digits.length() == 0) return result;
        helper(digits,result);
        return result;
    }
}
```

5.2 <u>Generate Parantheses</u>

Given `n` pairs of parentheses, write a function to *generate all combinations of well-formed parentheses*.

Example 1:

```
Input: n = 3
Output: ["((()))","(()())","(())()","()(())","()()()"]
```

Example 2:

```
Input: n = 1
Output: ["()"]
```

<u>JAVA:</u>

```
class Solution {
    public List<String> generateParenthesis(int n) {
        List<String> ans = new ArrayList<>();
        backtrack("", n, n, ans);
        return ans;
    }

    private void backtrack(String curr, int leftCount, int rightCount, List<String> ans) {
        if (leftCount == 0 && rightCount == 0) {
            ans.add(curr);
            return;
        }

        if (leftCount > 0) {
            backtrack(curr + "(", leftCount - 1, rightCount, ans);
        }

        if (rightCount > leftCount) {
            backtrack(curr + ")", leftCount, rightCount - 1, ans);
        }
    }
}
```

5.3 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]]
```

JAVA:

```
class Solution {
    public List<List<Integer>> permute(int[] nums) {
        List<List<Integer>> ans = new ArrayList<>();
        backtrack(new ArrayList<>(), ans, nums);
        return ans;
    }
    public void backtrack(List<Integer> curr, List<List<Integer>> ans, int[] nums) {
        if (curr.size() == nums.length) {
            ans.add(new ArrayList<>(curr));
            return;
        }
        for (int num : nums) {
            if (!curr.contains(num)) {
                curr.add(num);
                backtrack(curr, ans, nums);
                curr.remove(curr.size() - 1);
            }
        }
    }
}
```

5.4 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:

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

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

Example 2:

```
Input: nums = [0]

Output: [[],[0]]
```

JAVA:

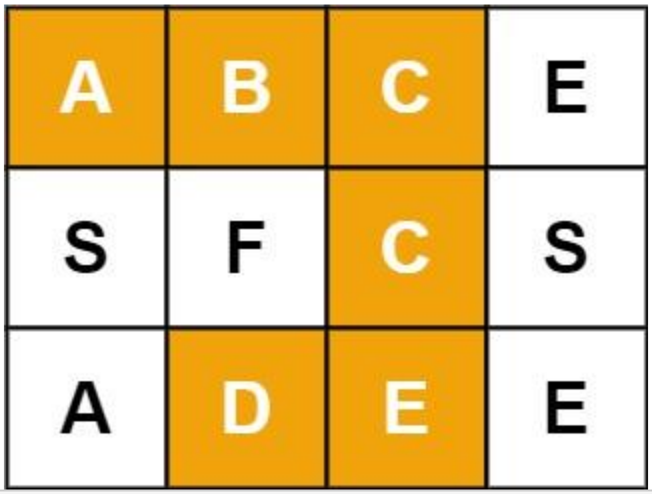
```
class Solution {
  List<List<Integer>> output = new ArrayList();
  int n, k;
  public void backtrack(ArrayList<Integer> curr,  int first, int[] nums) {
    // if the combination is done
    if (curr.size() == k) {
        output.add(new ArrayList(curr));
        return;
    }
    for (int i = first; i < n; ++i) {
      // add i into the current combination
      curr.add(nums[i]);
      // use next integers to complete the combination
      backtrack(curr, i + 1, nums);
      // backtrack
      curr.remove(curr.size() - 1);
    }
  }
  public List<List<Integer>> subsets(int[] nums) {
    n = nums.length;
    for (k = 0; k < n + 1; ++k) {
      backtrack(new ArrayList<Integer>(),0, nums);
    }
    return output;
  }
}
```

5.5 Word Search

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

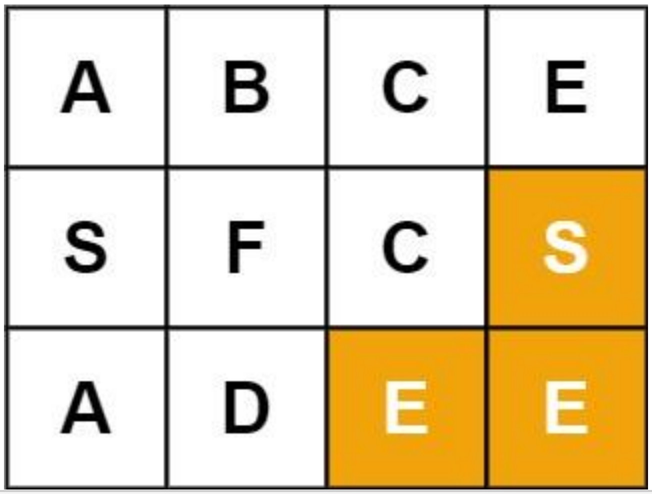
The word can be constructed from letters of sequentially adjacent cells, where adjacent cells are horizontally or vertically neighboring. The same letter cell may not be used more than once.

Example 1:



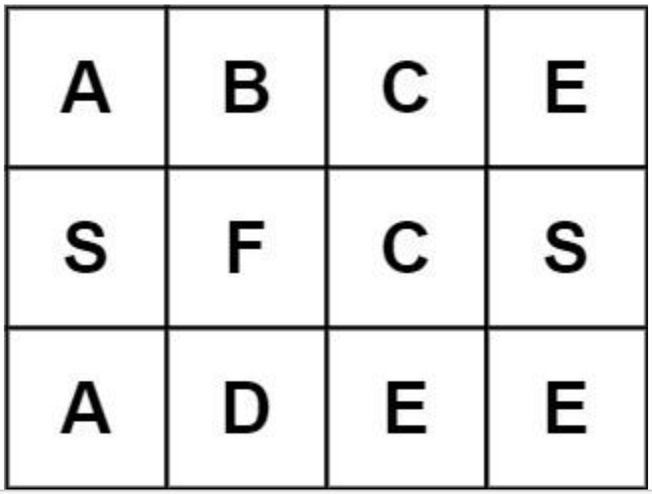
```
Input: board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "ABCC
ED"

Output: true
```

Example 2:



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

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 = "ABCB
"

Output: false
```

<u>JAVA:</u>

```
class Solution {
  private char[][] board;
  private int ROWS;
  private int COLS;

  public boolean exist(char[][] board, String word) {
    this.board = board;
    this.ROWS = board.length;
    this.COLS = board[0].length;

    for (int row = 0; row < this.ROWS; ++row)
      for (int col = 0; col < this.COLS; ++col)
        if (this.backtrack(row, col, word, 0))
          return true;
    return false;
  }

  protected boolean backtrack(int row, int col, String word, int index) {
    /* Step 1). check the bottom case. */
    if (index >= word.length())
      return true;

    /* Step 2). Check the boundaries. */
    if (row < 0 || row == this.ROWS || col < 0 || col == this.COLS
        || this.board[row][col] != word.charAt(index))
      return false;
    /* Step 3). explore the neighbors in DFS */
    boolean ret = false;
    // mark the path before the next exploration
    this.board[row][col] = '#';

    int[] rowOffsets = {0, 1, 0, -1};
    int[] colOffsets = {1, 0, -1, 0};
    for (int d = 0; d < 4; ++d) {
      ret = this.backtrack(row + rowOffsets[d], col + colOffsets[d], word, index + 1);
      if (ret)
        break;
    }

    /* Step 4). clean up and return the result. */
    this.board[row][col] = word.charAt(index);
    return ret;
  }
}
```

6. MATH

6.1 Happy Nuumber

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:

1² + 9² = 82

8² + 2² = 68

6² + 8² = 100

1² + 0² + 0² = 1
```

Example 2:

```
Input: n = 2

Output: false
```

JAVA:

```
class Solution {
   public boolean isHappy(int n) {
   int digit = 0;
   int quot = n;
   int result = 0;
   HashSet<Integer> seen = new HashSet<Integer>();
   while (quot != 0) {
       digit = quot%10;
```

```
        quot = quot/10;
        result += digit*digit;
        if (quot == 0) {
            if (result == 1) {
                return true;
            }
            else {
                if(seen.add(result)) {
                    quot = result;
                    result = 0;
                }
                else return false;
            }
        }
    }
    return false;
    }
}
```

6.2 Factorial Trailing Zeros

Given an integer `n`, return *the number of trailing zeroes in* `n!`.

Note that `n! = n * (n - 1) * (n - 2) * ... * 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
```

JAVA:

```
class Solution {

    public int trailingZeroes(int n) {
        int count = 0;
        while (n!=0) {
            n = n/5;
            count+=n;
        }
        return count;
    }


}
```

6.3 Excel Sheet Column Number

Given a string `columnTitle` that represents the column title as appears in an Excel sheet, return *its corresponding column number*.

For example:

```
A -> 1

B -> 2

C -> 3

...

Z -> 26

AA -> 27

AB -> 28

...
```

Example 1:

```
Input: columnTitle = "A"

Output: 1
```

Example 2:

```
Input: columnTitle = "AB"
```

```
Output: 28
```

Example 3:

```
Input: columnTitle = "ZY"

Output: 701
```

JAVA:

```
class Solution {
    public int titleToNumber(String columnTitle) {
        int result = 0;
        for (int i = columnTitle.length()-1; i >= 0; i--) {
            result += Math.pow(26,(columnTitle.length()-1-i))*(columnTitle.charAt(i)-'A'+1);
        }
        return result;
    }
}
```

6.4 Pow(x,n)

Implement pow(x, n), which calculates `x` raised to the power `n` (i.e., x^n).

Example 1:

```
Input: x = 2.00000, n = 10

Output: 1024.00000
```

Example 2:

```
Input: x = 2.10000, n = 3

Output: 9.26100
```

Example 3:

```
Input: x = 2.00000, n = -2

Output: 0.25000

Explanation: 2^-2 = 1/2^2 = 1/4 = 0.25
```

JAVA:

```
class Solution {
    public double binary (double x, long n) {
        double result = 1;
        if (n == 0) return 1;

        if (n < 0) {
            n = n*-1;
            x = 1/x;
        }

        while (n!= 0) {
            if (n%2 == 1) {
                result = result*x;
                n = n-1;
            }
            x = x*x;
            n = n/2;
        }
        return result;
    }
    public double myPow(double x, int n) {
        return binary(x, (long)n);
    }
}
```

6.5 Sqrt(x)

Given a non-negative integer `x`, return *the square root of* `x` *rounded down to the nearest integer*. The returned integer should be **non-negative** as well.

You **must not use** any built-in exponent function or operator.

- For example, do not use `pow(x, 0.5)` in c++ or `x ** 0.5` in python.

Example 1:

```
Input: x = 4

Output: 2

Explanation: The square root of 4 is 2, so we return 2.
```

Example 2:

```
Input: x = 8

Output: 2

Explanation: The square root of 8 is 2.82842..., and since we round it down to the n
earest integer, 2 is returned.
```

JAVA:

```
class Solution {
  public int mySqrt(int x) {
    if (x < 2) return x;

    long num;
    int pivot, left = 2, right = x / 2;
    while (left <= right) {
      pivot = left + (right - left) / 2;
      num = (long)pivot * pivot;
      if (num > x) right = pivot - 1;
      else if (num < x) left = pivot + 1;
      else return pivot;
    }
    return right;
  }
}
```

6.6 Divide Two Integers

Given two integers `dividend` and `divisor`, divide two integers **without** using multiplication, division, and mod operator.

The integer division should truncate toward zero, which means losing its fractional part. For example, `8.345` would be truncated to `8`, and `-2.7335` would be truncated to `-2`.

Return *the **quotient** after dividing* `dividend` *by* `divisor`.

Note: Assume we are dealing with an environment that could only store integers within the **32-bit** signed integer range: $[-2^{31}, 2^{31} - 1]$. For this problem, if the quotient is **strictly greater than** $2^{31} - 1$, then return $2^{31} - 1$, and if the quotient is **strictly less than** -2^{31}, then return -2^{31}.

Example 1:

```
Input: dividend = 10, divisor = 3

Output: 3
```

```
Explanation: 10/3 = 3.33333.. which is truncated to 3.
```

Example 2:

```
Input: dividend = 7, divisor = -3

Output: -2

Explanation: 7/-3 = -2.33333.. which is truncated to -2.
```

JAVA:

```
class Solution {
    public int divide(int dividend, int divisor) {
        if (dividend == Integer.MIN_VALUE && divisor == -1) {
            return Integer.MAX_VALUE; // Integer overflow edge case
        }
        int sign = (dividend < 0) ^ (divisor < 0) ? -1 : 1;
        long dvd = Math.abs((long) dividend);
        long dvs = Math.abs((long) divisor);
        int result = 0;
        while (dvd >= dvs) {
            long temp = dvs;
            long multiple = 1;
            while (dvd >= (temp << 1)) {
                temp <<= 1;
                multiple <<= 1;
            }
            dvd -= temp;
            result += multiple;
        }
        return result * sign;
    }
}
```

6.7 Fraction To Recurring Decimal

Given two integers representing the `numerator` and `denominator` of a fraction, return *the fraction in string format*.

If the fractional part is repeating, enclose the repeating part in parentheses.

If multiple answers are possible, return **any of them**.

It is **guaranteed** that the length of the answer string is less than 10^4 for all the given inputs.

Example 1:

```
Input: numerator = 1, denominator = 2

Output: "0.5"
```

Example 2:

```
Input: numerator = 2, denominator = 1

Output: "2"
```

Example 3:

```
Input: numerator = 4, denominator = 333

Output: "0.(012)"
```

JAVA:

```
class Solution {
    public String fractionToDecimal(int numerator, int denominator) {
    if (numerator == 0) {
      return "0";
    }
    StringBuilder fraction = new StringBuilder();
    // If either one is negative (not both)
    if (numerator < 0 ^ denominator < 0) {
      fraction.append("-");
    }
    // Convert to Long or else abs(-2147483648) overflows
    long dividend = Math.abs(Long.valueOf(numerator));
    long divisor = Math.abs(Long.valueOf(denominator));
    fraction.append(String.valueOf(dividend / divisor));
    long remainder = dividend % divisor;
    if (remainder == 0) {
      return fraction.toString();
    }
    fraction.append(".");
    Map<Long, Integer> map = new HashMap<>();
    while (remainder != 0) {
      if (map.containsKey(remainder)) {
          fraction.insert(map.get(remainder), "(");
          fraction.append(")");
          break;
```

```
        }
        map.put(remainder, fraction.length());
        remainder *= 10;
        fraction.append(String.valueOf(remainder / divisor));
        remainder %= divisor;
      }
      return fraction.toString();
  }
}
```

7. OTHERS

7.1 Sum Of Two Integers

Given two integers `a` and `b`, return *the sum of the two integers without using the operators* `+` *and* `-`.

Example 1:

```
Input: a = 1, b = 2
Output: 3
```

Example 2:

```
Input: a = 2, b = 3
Output: 5
```

JAVA:

```
class Solution {
    public int getSum(int a, int b) {
        while (b != 0) {
            int answer = a ^ b;
            int carry = (a & b) << 1;
            a = answer;
            b = carry;
        }
        return a;
    }
}
```

7.2 Evaluate Reverse Polish Notation

You are given an array of strings `tokens` that represents an arithmetic expression in a Reverse Polish Notation.

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

Note that:

- The valid operators are `'+'`, `'-'`, `'*'`, and `'/'`.
- Each operand may be an integer or another expression.
- The division between two integers always **truncates toward zero**.
- There will not be any division by zero.

- The input represents a valid arithmetic expression in a reverse polish notation.
- The answer and all the intermediate calculations can be represented in a **32-bit** integer.

Example 1:

```
Input: tokens = ["2","1","+","3","*"]

Output: 9

Explanation: ((2 + 1) * 3) = 9
```

Example 2:

```
Input: tokens = ["4","13","5","/","+"]

Output: 6

Explanation: (4 + (13 / 5)) = 6
```

Example 3:

```
Input: tokens = ["10","6","9","3","+","-11","*","/","*","17","+","5","+"]

Output: 22

Explanation: ((10 * (6 / ((9 + 3) * -11))) + 17) + 5

= ((10 * (6 / (12 * -11))) + 17) + 5

= ((10 * (6 / -132)) + 17) + 5

= ((10 * 0) + 17) + 5

= (0 + 17) + 5

= 17 + 5

= 22
```

JAVA:

```
class Solution {
    public int evalRPN(String[] tokens) {
        Stack<Integer> st = new Stack<Integer>();
        for (String token: tokens) {
            if (!"+-*/".contains(token)) {
                st.push(Integer.valueOf(token));
                continue;
            }
            int a = st.pop();
            int b = st.pop();
            int result = 0;
            switch (token) {
                case "+":
                    result = b+a;
                    break;
                case "-":
                    result = b-a;
                    break;
                case "*":
                    result = b*a;
                    break;
                case "/":
                    result = b/a;
                    break;
            }
            st.push(result);
        }
        return st.pop();
    }
}
```

7.3 <u>Majority Element</u>

Given an array `nums` of size `n`, return *the majority element*.

The majority element is the element that appears more than `⌊n / 2⌋` 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
```

JAVA:

```
public class Solution {
    public int majorityElement(int[] nums) {
        int count = 0;
        int candidate = 0;

        for (int num : nums) {
            if (count == 0) {
                candidate = num;
            }
            count += (num == candidate) ? 1 : -1;
        }

        return candidate;
    }
}
```

7.4 Find The Celebrity

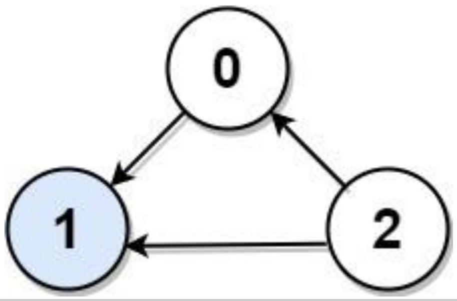
Suppose you are at a party with `n` people labeled from `0` to `n - 1` and among them, there may exist one celebrity. The definition of a celebrity is that all the other `n - 1` people know the celebrity, but the celebrity does not know any of them.

Now you want to find out who the celebrity is or verify that there is not one. You are only allowed 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)` that tells you whether `a` knows `b`. Implement a function `int findCelebrity(n)`. There will be exactly one celebrity if they are at the party.

Return *the celebrity's label if there is a celebrity at the party*. If there is no celebrity, return `-1`.

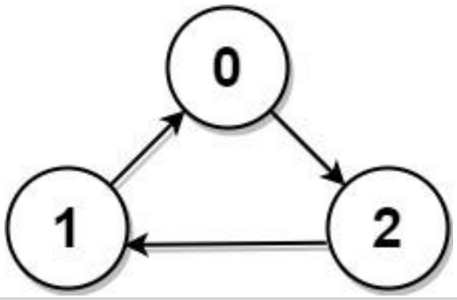
Example 1:



```
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 means
person i knows person j, otherwise graph[i][j] = 0 means person i does not know pers
on j. The celebrity is the person labeled as 1 because both 0 and 2 know him but 1 d
oes not know anybody.
```

Example 2:



```
Input: graph = [[1,0,1],[1,1,0],[0,1,1]]

Output: -1

Explanation: There is no celebrity.
```

JAVA:

```
public class Solution extends Relation {
    int number_of_people;
    public int findCelebrity(int n) {
        int candidate = 0;
        number_of_people = n;
        for (int i = 0; i < n; i++) {
            if (knows(candidate, i)) {
                candidate = i;
            }
        }
        if (isCelebrity(candidate)) {
            return candidate;
        }
        return -1;
    }
    public boolean isCelebrity(int candidate) {
        for (int i = 0; i < number_of_people; i++) {
            if (i == candidate) continue;
            if (knows(candidate,i) || !knows(i,candidate)) {
                return false;
            }
        }
        return true;
    }
}
```

7.5 Task Scheduler

Given a characters array `tasks`, representing the tasks a CPU needs to do, where each letter represents a different task. Tasks could be done in any order. Each task is done in one unit of time. For each unit of time, the CPU could complete either one task or just be idle.

However, there is a non-negative integer `n` that represents the cooldown period between two **same tasks** (the same letter in the array), that is that there must be at least `n` units of time between any two same tasks.

Return *the least number of units of times that the CPU will take to finish all the given tasks*.

Example 1:

```
Input: tasks = ["A","A","A","B","B","B"], n = 2

Output: 8

Explanation:

A -> B -> idle -> A -> B -> idle -> A -> B

There is at least 2 units of time between any two same tasks.
```

Example 2:

```
Input: tasks = ["A","A","A","B","B","B"], n = 0

Output: 6

Explanation: On this case any permutation of size 6 would work since n = 0.

["A","A","A","B","B","B"]

["A","B","A","B","A","B"]

["B","B","B","A","A","A"]

...

And so on.
```

Example 3:

```
Input: tasks = ["A","A","A","A","A","A","B","C","D","E","F","G"], n = 2

Output: 16
```

```
Explanation:

One possible solution is

A -> B -> C -> A -> D -> E -> A -> F -> G -> A -> idle -> idle -> A -> idle -> idle
-> A
```

JAVA:

```
class Solution {
    public int leastInterval(char[] tasks, int n) {
        HashMap<Character, Integer> mp = new HashMap<>();
        int fre = 0, ans = 0;
        for (char e : tasks) {
            fre = Math.max(fre, mp.getOrDefault(e, 0) + 1);
            mp.put(e, mp.getOrDefault(e, 0) + 1);
        }
        for (Map.Entry<Character, Integer> entry : mp.entrySet()) {
            if (entry.getValue() == fre) {
                ans++;
            }
        }
        return Math.max(tasks.length, ans + (fre - 1) * (n + 1));
    }
}
```

8. DYNAMIC PROGRAMMING

8.1 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 jump leng
th is 0, which makes it impossible to reach the last index.
```

JAVA:

```
class Solution {
    public boolean canJump(int[] nums) {
        int target = nums.length-1;
        for (int i = nums.length-2; i >= 0; i--) {
            if (i+nums[i] >= target) target = i;
        }
        return (target == 0);
    }
}
```

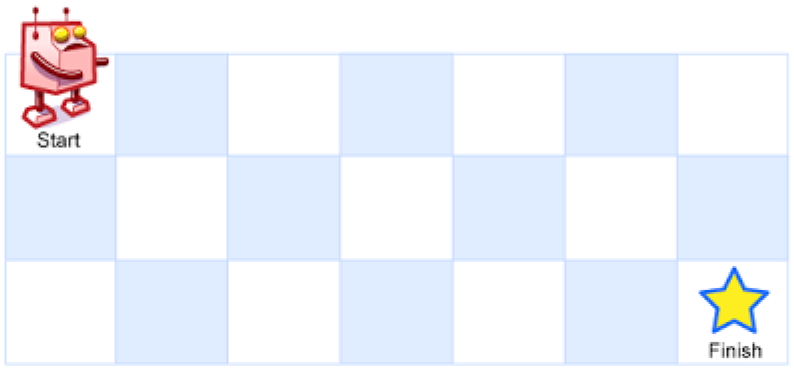
8.2 Unique Paths

There is a robot on an `m x n` grid. The robot is initially located at the **top-left corner** (i.e., `grid[0][0]`). The robot tries to move to the **bottom-right corner** (i.e., `grid[m - 1][n - 1]`). The robot can only move either down or right at any point in time.

Given the two integers `m` and `n`, return *the number of possible unique paths that the robot can take to reach the bottom-right corner*.

The test cases are generated so that the answer will be less than or equal to $2 * 10^9$.

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 bott
om-right corner:

1. Right -> Down -> Down

2. Down -> Down -> Right

3. Down -> Right -> Down
```

JAVA:

```
class Solution {
    public int uniquePaths(int m, int n) {
        int[][] dp = new int[m][n];

        // Fill the first row and first column with 1 (only one way to reach cells in the first row and first
column)
        for (int i = 0; i < m; i++) {
            dp[i][0] = 1;
        }
        for (int j = 0; j < n; j++) {
            dp[0][j] = 1;
        }
        // Fill the rest of the dp array using dynamic programming
        for (int i = 1; i < m; i++) {
            for (int j = 1; j < n; j++) {
                // The number of unique paths to cell (i, j) is the sum of paths from the cell above and the cell
to the left
                dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
            }
        }

        // The value in the bottom-right cell represents the total number of unique paths
        return dp[m - 1][n - 1];
    }
}
```

8.3 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
```

JAVA:

```
public class Solution {
  public int coinChange(int[] coins, int amount) {
    int max = amount + 1;
    int[] dp = new int[amount + 1];
    Arrays.fill(dp, max);
    dp[0] = 0;
    for (int i = 1; i <= amount; i++) {
      for (int j = 0; j < coins.length; j++) {
        if (coins[j] <= i) {
          dp[i] = Math.min(dp[i], dp[i - coins[j]] + 1);
        }
      }
    }
    return dp[amount] > amount ? -1 : dp[amount];
  }
}
```

8.4 Longest Increasing Subsequence

Given an integer array `nums`, return *the length of the longest* ***strictly increasing subsequence***.

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

[5,4,-1,7,8] has the largest sum 23.
```

JAVA:

```
class Solution {
    public int lengthOfLIS(int[] nums) {
        ArrayList<Integer> sub = new ArrayList<>();
        sub.add(nums[0]);

        for (int i = 1; i < nums.length; i++) {
            int num = nums[i];
            if (num > sub.get(sub.size() - 1)) {
                sub.add(num);
            } else {
                // Find the first element in sub that is greater than or equal to num
                int j = 0;
                while (num > sub.get(j)) {
                    j += 1;
                }

                sub.set(j, num);
            }
            System.out.println(sub.toString());
        }

        return sub.size();
    }
}
```

9. DESIGN

9.1 Flatten 2D Vector

Design an iterator to flatten a 2D vector. It should support the `next` and `hasNext` operations.

Implement the `Vector2D` class:

- `Vector2D(int[][] vec)` initializes the object with the 2D vector `vec`.
- `next()` returns the next element from the 2D vector and moves the pointer one step forward. You may assume that all the calls to `next` are valid.
- `hasNext()` returns `true` if there are still some elements in the vector, and `false` otherwise.

Example 1:

```
Input
["Vector2D", "next", "next", "next", "hasNext", "hasNext", "next", "hasNext"]
[[[[1, 2], [3], [4]]], [], [], [], [], [], [], []]
Output
[null, 1, 2, 3, true, true, 4, false]


Explanation
Vector2D vector2D = new Vector2D([[1, 2], [3], [4]]);
vector2D.next();    // return 1
vector2D.next();    // return 2
vector2D.next();    // return 3
vector2D.hasNext(); // return True
vector2D.hasNext(); // return True
vector2D.next();    // return 4
vector2D.hasNext(); // return False
it impossible to reach the last index.
```

JAVA:

```
class Vector2D {
  int [][] curr;
```

```
    int idx;
    int rows;
    int capacity;
    ArrayList<Integer> list= new ArrayList<Integer>();
    public Vector2D(int[][] vec) {
        curr = vec;
        idx = 0;
        rows = vec.length;
        int[] cols = new int[rows];

        System.out.println("Rows"+rows);
        for (int i = 0; i < rows; i++) {
                cols[i] = vec[i].length;
                System.out.println("Cols for multi entries:"+cols[i]);

        }

        for (int j = 0; j < rows; j++) {
            for (int k = 0; k < cols[j]; k++) {
                list.add(vec[j][k]);
            }
        }
        System.out.println(list.toString());
        capacity = list.size();
    }

    public int next() {
        int val = list.get(idx);
        list.remove(idx);
        capacity--;
        System.out.println("val:"+val+"idx:"+idx+"capacity:"+capacity);
        return val;
    }

    public boolean hasNext() {
        if (capacity > 0) return true;
        return false;
    }
}

/**
 * Your Vector2D object will be instantiated and called as such:
 * Vector2D obj = new Vector2D(vec);
 * int param_1 = obj.next();
 * boolean param_2 = obj.hasNext();
 */
```

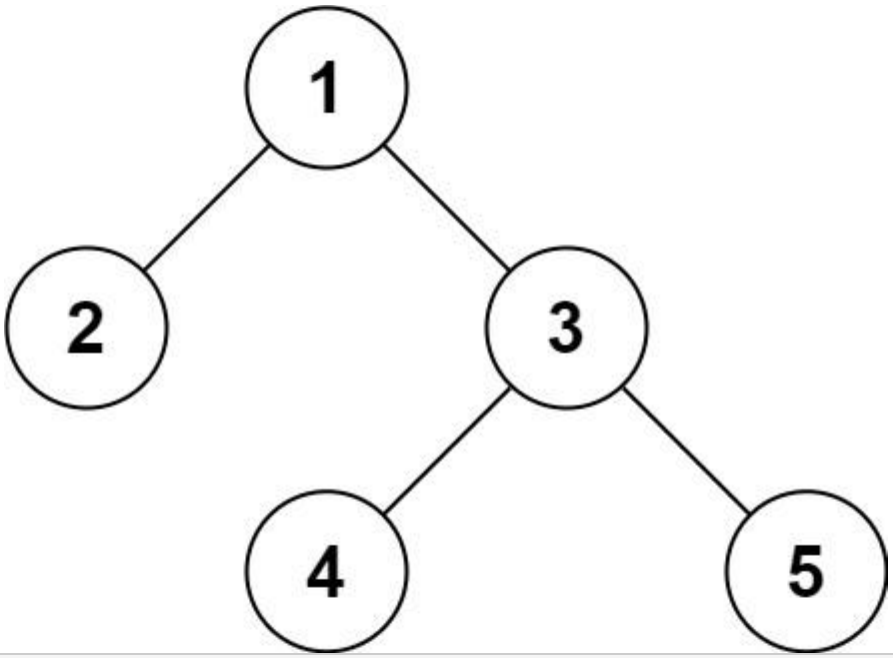
9.2 Serialize And Deserialize Binary Tree

Serialization is the process of converting a data structure or object into a sequence of bits so that it can be stored in a file or memory buffer, or transmitted across a network connection link to be reconstructed later in the same or another computer environment.

Design an algorithm to serialize and deserialize a binary tree. There is no restriction on how your serialization/deserialization algorithm should work. You just need to ensure that a binary tree can be serialized to a string and this string can be deserialized to the original tree structure.

Clarification: The input/output format is the same as how LeetCode serializes a binary tree. You do not necessarily need to follow this format, so please be creative and come up with different approaches yourself.

Example 1:



```
Input: root = [1,2,3,null,null,4,5]

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

Example 2:

```
Input: root = []

Output: []
```

JAVA:

/**

```
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Codec {
      public String rserialize(TreeNode root, String str) {
    // Recursive serialization.
    if (root == null) {
      str += "null,";
    } else {
      str += str.valueOf(root.val) + ",";
      str = rserialize(root.left, str);
      str = rserialize(root.right, str);
    }
    return str;
  }
  public TreeNode rdeserialize(List<String> l) {
    // Recursive deserialization.
    if (l.get(0).equals("null")) {
      l.remove(0);
      return null;
    }

    TreeNode root = new TreeNode(Integer.valueOf(l.get(0)));
    l.remove(0);
    root.left = rdeserialize(l);
    root.right = rdeserialize(l);

    return root;
  }
    // Encodes a tree to a single string.
    public String serialize(TreeNode root) {
        return rserialize(root, "");
    }

    // Decodes your encoded data to tree.
    public TreeNode deserialize(String data) {
        String[] data_array = data.split(",");
        List<String> data_list = new LinkedList<String>(Arrays.asList(data_array));
        return rdeserialize(data_list);
    }
}
```

// Your Codec object will be instantiated and called as such:
// Codec ser = new Codec();
// Codec deser = new Codec();
// TreeNode ans = deser.deserialize(ser.serialize(root));

9.3 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", "insert", "ge
tRandom"]

[[], [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 inserted suc
cessfully.

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 contains [1,
2].

randomizedSet.getRandom(); // getRandom() should return either 1 or 2 randomly.
randomizedSet.remove(1); // Removes 1 from the set, returns true. Set now contains [
2].

randomizedSet.insert(2); // 2 was already in the set, so return false.

randomizedSet.getRandom(); // Since 2 is the only number in the set, getRandom() wil
l always return 2.
```

JAVA:

```
class RandomizedSet {
  Map<Integer, Integer> dict;
  List<Integer> list;
  Random rand = new Random();

  /** Initialize your data structure here. */
  public RandomizedSet() {
    dict = new HashMap();
    list = new ArrayList();
  }

  /** Inserts a value to the set. Returns true if the set did not already contain the specified element. */
  public boolean insert(int val) {
    if (dict.containsKey(val)) return false;

    dict.put(val, list.size());
    list.add(list.size(), val);
    return true;
  }

  /** Removes a value from the set. Returns true if the set contained the specified element. */
  public boolean remove(int val) {
    if (! dict.containsKey(val)) return false;

    // move the last element to the place idx of the element to delete
    int lastElement = list.get(list.size() - 1);
    int idx = dict.get(val);
    list.set(idx, lastElement);
    dict.put(lastElement, idx);
    // delete the last element
    list.remove(list.size() - 1);
    dict.remove(val);
    return true;
  }

  /** Get a random element from the set. */
  public int getRandom() {
    return list.get(rand.nextInt(list.size()));
  }
}

/**
 * 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();
 */
```

9.4 Tic Tac Toe

Assume the following rules are for the tic-tac-toe game on an `n x n` board between two players:

1. A move is guaranteed to be valid and is placed on an empty block.
2. Once a winning condition is reached, no more moves are allowed.
3. A player who succeeds in placing `n` of their marks in a horizontal, vertical, or diagonal row wins the game.

Implement the `TicTacToe` class:

- `TicTacToe(int n)` Initializes the object the size of the board `n`.
- `int move(int row, int col, int player)` Indicates that the player with id `player` plays at the cell `(row, col)` of the board. The move is guaranteed to be a valid move, and the two players alternate in making moves. Return
  - `0` if there is **no winner** after the move,
  - `1` if **player 1** is the winner after the move, or
  - `2` if **player 2** is the winner after the move.

Example 1:

```
Input

["TicTacToe", "move", "move", "move", "move", "move", "move", "move"]

[[3], [0, 0, 1], [0, 2, 2], [2, 2, 1], [1, 1, 2], [2, 0, 1], [1, 0, 2], [2, 1, 1]]

Output

[null, 0, 0, 0, 0, 0, 0, 1]



Explanation

TicTacToe ticTacToe = new TicTacToe(3);
```

```
Assume that player 1 is "X" and player 2 is "O" in the board.

ticTacToe.move(0, 0, 1); // return 0 (no one wins)

|X| | |

| | | |    // Player 1 makes a move at (0, 0).

| | | |


ticTacToe.move(0, 2, 2); // return 0 (no one wins)

|X| |O|

| | | |    // Player 2 makes a move at (0, 2).

| | | |


ticTacToe.move(2, 2, 1); // return 0 (no one wins)

|X| |O|

| | | |    // Player 1 makes a move at (2, 2).

| | |X|


ticTacToe.move(1, 1, 2); // return 0 (no one wins)

|X| |O|

| |O| |    // Player 2 makes a move at (1, 1).

| | |X|


ticTacToe.move(2, 0, 1); // return 0 (no one wins)

|X| |O|

| |O| |    // Player 1 makes a move at (2, 0).

|X| |X|


ticTacToe.move(1, 0, 2); // return 0 (no one wins)

|X| |O|
|O|O| |    // Player 2 makes a move at (1, 0).

|X| |X|



ticTacToe.move(2, 1, 1); // return 1 (player 1 wins)

|X| |O|

|O|O| |    // Player 1 makes a move at (2, 1).

|X|X|X|
```

JAVA:

```
public class TicTacToe {
    int[] rows;
    int[] cols;
    int diagonal;
    int antiDiagonal;

    public TicTacToe(int n) {
        rows = new int[n];
        cols = new int[n];
    }

    public int move(int row, int col, int player) {
        int currentPlayer = (player == 1) ? 1 : -1;
        // update currentPlayer in rows and cols arrays
        rows[row] += currentPlayer;
        cols[col] += currentPlayer;
        // update diagonal
        if (row == col) {
            diagonal += currentPlayer;
        }
        //update anti diagonal
        if (col == (cols.length - row - 1)) {
            antiDiagonal += currentPlayer;
        }
        int n = rows.length;
        // check if the current player wins
        if (Math.abs(rows[row]) == n ||
                Math.abs(cols[col]) == n ||
                Math.abs(diagonal) == n ||
                Math.abs(antiDiagonal) == n) {
            return player;
        }
        // No one wins
        return 0;
    }
}

/**
 * Your TicTacToe object will be instantiated and called as such:
 * TicTacToe obj = new TicTacToe(n);
 * int param_1 = obj.move(row,col,player);
 */
```