# TABLE OF CONTENTS

| S. NO | PROBLEM | TOPIC |
|---|---|---|
| 31 | WORD LADDER | TREES AND GRAPHS |
| 32 | COURSE SCHEDULE | TREES AND GRAPHS |
| 33 | DIAMETER OF BINARY TREE | TREES AND GRAPHS |
| 34 | FLOOD FILL | TREES AND GRAPHS |
| 35 | SYMMETRIC TREE | TREES AND GRAPHS |
| 36 | BINARY TREE ZIGZAG LEVEL ORDER TRAVERSAL | TREES AND GRAPHS |
| 37 | WORD LADDER 2 | TREES AND GRAPHS |
| 38 | NUMBER OF ISLANDS | TREES AND GRAPHS |
| 39 | LOWEST COMMON ANCESTOR OF A BINARY TREE | TREES AND GRAPHS |
| 40 | CUT OFF TREES FOR GOLF EVENT | TREES AND GRAPHS |
| 41 | LETTERS COMBINATIONS OF A PH NUMBER | RECURSION |
| 42 | GENERATE PARANTHESIS | RECURSION |
| 43 | WORD SEARCH | RECURSION |
| 44 | WORD SEARCH II | RECURSION |
| 45 | MEDIAN OF TWO SORTED ARRAYS | SORTING AND SEARCHING |
| 46 | MERGE INTERVALS | SORTING AND SEARCHING |
| 47 | Kth LARGEST ELEMENT IN AN ARRAY | SORTING AND SEARCHING |
| 48 | TOP K FREQUENT ELEMENTS | SORTING AND SEARCHING |
| 49 | SEARCH IN ROTATED SORTED ARRAY | SORTING AND SEARCHING |
| 50 | TWO SUM II - INPUT ARRAY IS SORTED | SORTING AND SEARCHING |
| 51 | MEETING ROOMS II | SORTING AND SEARCHING |
| 52 | K CLOSEST POINTS TO ORIGIN | SORTING AND SEARCHING |
| 53 | LONGEST PALLINDROMIC SUBSTRING | DP |
| 54 | BEST TIME TO BUY AND SELL STOCK | DP |
| 55 | COIN CHANGE | DP |
| 56 | MAXIMUM SUBARRAY | DP |
| 57 | WORD BREAK | DP |
| 58 | LRU CACHE | DESIGN |
| 59 | FIND MEDIAN FROM DATA STREAM | DESIGN |
| 60 | DESIGN TIC TAC TOE | DESIGN |
| 61 | MAXIMUM FREQUENCY STACK | DESIGN |
| 62 | MIN STACK | DESIGN |

| S. NO | PROBLEM | TOPIC |
|---|---|---|
| **63** | SERIALISE AND DE-SERIALISE BINARY TREE | DESIGN |
| **64** | DESIGN SEARCH AUTOCOMPLETE SYSTEM | DESIGN |
| **65** | REVERSE INTEGER | OTHERS |
| **66** | PARTITION LABELS | OTHERS |
| **67** | SECOND HIGHEST SALARY | OTHERS |
| **68** | PRISON CELLS AFTER N DAYS | OTHERS |

**1. ROMAN TO INTEGER**

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 a roman numeral, convert it to an integer.

```
import java.util.Map;
import java.util.HashMap;

class Solution {
    public int romanToInt(String s) {
        Map<Character, Integer> m = new HashMap<>();
        m.put('I', 1);
        m.put('V', 5);
        m.put('X', 10);
        m.put('L', 50);
        m.put('C', 100);
        m.put('D', 500);
        m.put('M', 1000);

        int sum = 0;
        char prev = '\0';
        for(char c: s.toCharArray()) {
            if(prev == 'I') {
                if(c == 'V' || c == 'X') {
                    sum -= 2;
                }
            } else if (prev == 'X') {
                if(c == 'L' || c == 'C') {
                    sum -= 20;
                }
            } else if (prev == 'C') {
                if(c == 'D' || c == 'M') {
                    sum -= 200;
                }
            }
            sum += m.get(c);
            prev = c;
        }
        return sum;
    }
}
```

**2. INTEGER TO ROMAN**

```
class Solution {
    public String intToRoman(int num) {

        StringBuilder romanBuilder = new StringBuilder();

        while (num != 0) {
            if (num >= 1000) {
                num -= 1000;
                romanBuilder.append("M");
            } else if (num >= 900) {
                num -= 900;
                romanBuilder.append("CM");
            } else if (num >= 500) {
                num -= 500;
                romanBuilder.append("D");
            } else if (num >= 400) {
                num -= 400;
                romanBuilder.append("CD");
            } else if (num >= 100) {
                num -= 100;
                romanBuilder.append("C");
            } else if (num >= 90) {
                num -= 90;
                romanBuilder.append("XC");
            } else if (num >= 50) {
                num -= 50;
                romanBuilder.append("L");
            } else if (num >= 40) {
                num -= 40;
                romanBuilder.append("XL");
            } else if (num >= 10) {
                num -= 10;
                romanBuilder.append("X");
            } else if (num >= 9) {
                num -= 9;
                romanBuilder.append("IX");
            } else if (num >= 5) {
                num -= 5;
                romanBuilder.append("V");
            } else if (num >= 4) {
                num -= 4;
                romanBuilder.append("IV");
            } else {
                num -= 1;
                romanBuilder.append("I");
            }
        }
        return romanBuilder.toString();
    }
```

**3. 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].
```

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

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

        for(int i = 0; i < nums.length; i++){
            if(map.containsKey(target - nums[i])){
                return new int[] {map.get(target - nums[i]), i};
            }
            else{
                map.put(nums[i], i);
            }
        }
        return new int[] { };
    }
}
```

## 4. STRING TO INTEGER

Implement `atoi` which converts a string to an integer.
The function first discards as many whitespace characters as necessary until the first non-whitespace character is found. Then, starting from this character takes an optional initial plus or minus sign followed by as many numerical digits as possible, and interprets them as a numerical value.
The string can contain additional characters after those that form the integral number, which are ignored and have no effect on the behaviour of this function.
If the first sequence of non-whitespace characters in str is not a valid integral number, or if no such sequence exists because either str is empty or it contains only whitespace characters, no conversion is performed.
If no valid conversion could be performed, a zero value is returned.
**Note:**

- Only the space character `' '` is considered a whitespace character.
- Assume we are dealing with an environment that could only store integers within the 32-bit signed integer range: `[−231,  231 − 1]`. If the numerical value is out of the range of representable values, `231 − 1` or `−231` is returned.

**Example 1:**
**Input:** `str = "42"`
**Output:** `42`

```
class Solution {
    public int myAtoi(String str) {
        if(str.trim().length() == 0) return 0;

        Scanner s = new Scanner(str.trim()).useDelimiter("\\s");
        // System.out.println(
        String t = s.next();

        char ch;
        int res = 0, sign_flag = 1;
        for(int i = 0; i<t.length();i++){
            ch = t.charAt(i);
            if(i == 0 && (ch == '-' || ch == '+')){
                if(ch == '-'){
                    sign_flag = -1;
                }
            } else {
                if(ch > 47 && ch <58){
                    System.out.println(sign_flag);
                    int k = Character.getNumericValue(ch);
                    if((sign_flag == -1) && (res > Integer.MAX_VALUE/10 || (res == Integer.MAX_VALUE/10
&& k >= 8))) return Integer.MIN_VALUE;
                    else if(res > Integer.MAX_VALUE/10 || (res == Integer.MAX_VALUE/10 && k > 7)) return
Integer.MAX_VALUE;
                    res = res *10 + k;
                } else {
                    break;
                }
            }
        }
        System.out.println(res);
        return res * sign_flag;
    }
}
```

5. **3SUM**

Given an array nums of n integers, are there elements a, b, c in nums such that a + b + c = 0? Find all unique triplets in the array which gives the sum of zero.
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: []

```
class Solution {
    public List<List<Integer>> threeSum(int[] nums) {
        // Map<Integer, Integer> m = new HashMap<>();
        Arrays.sort(nums);
        List<List<Integer>> list = new ArrayList<List<Integer>> ();
        int i = 0, k = nums.length-1, j = i+1, sum;

        while(i+1 < k) {
            if(nums[i]>0) break;
            // System.out.println( i + " " +j + " " + k);
            sum = nums[i] + nums[j] + nums[k];
            // System.out.println(sum);
            if(sum == 0) {
                list.add(Arrays.asList(nums[i], nums[j], nums[k]));
                j++;
                while(j < k && nums[j] == nums[j-1]) j++;
                k--;
                while(k > j && nums[k] == nums[k+1]) k--;
            } else if (sum > 0)
                k--;
            else
                j++;
            if(j >= k) {
                i++;
                while(i < nums.length-1 && nums[i] == nums[i-1]) i++;
                k = nums.length -1;
                j = i+1;
            }
        }

        return list;
    }
}
```

**6. IMPLEMENT strStr()**

Implement 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() and Java's indexOf().

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

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

        if (needle.length() == 0)
            return 0;
        if (haystack.length() == 0)
            return -1;

        for (int i = 0; i < haystack.length(); i++) {
            // no enough places for needle after i
            if (i + needle.length() > haystack.length()) break;

            for (int j = 0; j < needle.length(); j++) {
                if (haystack.charAt(i+j) != needle.charAt(j))
                    break;
                if (j == needle.length()-1)
                    return i;
            }
        }

        return -1;
    }
}
```

## 7. 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"]]

```
class Solution {
    public List<List<String>> groupAnagrams(String[] strs) {
        if (strs.length == 0) return new ArrayList();
        Map<String, List> ans = new HashMap<String, List>();
        int[] count = new int[26];

        for (String s : strs) {
            Arrays.fill(count, 0);
            for (char c : s.toCharArray()) count[c - 'a']++;

            StringBuilder sb = new StringBuilder("");
            for (int i = 0; i < 26; i++) {
                sb.append('#');
                sb.append(count[i]);
            }
            String key = sb.toString();
            if (!ans.containsKey(key)) ans.put(key, new ArrayList());
            ans.get(key).add(s);
        }
        return new ArrayList(ans.values());
    }
}
```

8. **COMPARE VERSION NUMBER**

Given two version numbers, version1 and version2, compare them.
Version numbers consist of one or more revisions joined by a dot '.'. Each revision consists of digits and may contain leading zeros. Every revision contains at least one character.
Revisions are 0-indexed from left to right, with the leftmost revision being revision 0, the next revision being revision 1, and so on.
For example 2.5.33 and 0.1 are valid version numbers.

To compare version numbers, compare their revisions in left-to-right order. Revisions are compared using their integer value ignoring any leading zeros. This means that revisions 1 and 001 are considered equal. If a version number does not specify a revision at an index, then treat the revision as 0.

For example, version 1.0 is less than version 1.1 because their revision 0s are the same, but their revision 1s are 0 and 1 respectively, and 0 < 1.
Return the following:

- If version1 < version2, return -1.
- If version1 > version2, return 1.
- Otherwise, return 0.

Example 1:
Input: version1 = "1.01", version2 = "1.001"
Output: 0
Explanation: Ignoring leading zeroes, both "01" and "001" represent the same integer "1".

*I solve it by making the two version number same length. For example, if version1 = "1.0.2", and version2 = "1.0", the I will convert the version2 to "1.0.0".*

```
public int compareVersion(String version1, String version2) {

    String[] v1 = version1.split("\\.");
    String[] v2 = version2.split("\\.");

    for ( int i = 0; i < Math.max(v1.length, v2.length); i++ ) {
        int num1 = i < v1.length ? Integer.parseInt( v1[i] ) : 0;
        int num2 = i < v2.length ? Integer.parseInt( v2[i] ) : 0;
        if ( num1 < num2 ) {
            return -1;
        } else if ( num1 > num2 ) {
            return +1;
        }
    }

    return 0;
}
```

## 9.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.
Follow up: Could you implement a solution using only O(1) extra space complexity and O(n) runtime complexity?

Example 1:
Input: nums = [3,0,1]
Output: 2
Explanation: n = 3 since there are 3 numbers, so all numbers are in the range [0,3]. 2 is the missing number in the range since it does not appear in nums.

Example 2:
Input: nums = [0,1]
Output: 2
Explanation: n = 2 since there are 2 numbers, so all numbers are in the range [0,2]. 2 is the missing number in the range since it does not appear in nums.

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

        int sum = 0;
        for(int i : nums){
            sum += i;
        }

        int len = nums.length;

        return len*(len+1)/2 - sum;
    }
}
```

## 10. FIRST UNIQUE CHARACTER IN A STRING

Given a string, find the first non-repeating character in it and return its index. If it doesn't exist, return -1.

**Examples**:
s = "leetcode"
return 0.

s = "loveleetcode"
return 2.

**Note**: You may assume the string contains only lowercase English letters.

```
class Solution {
    public int firstUniqChar(String s) {

        //Create HashMap
        LinkedHashMap<Character,Integer> hm = new LinkedHashMap<>();

        //Iterate over string and store in HashMap
        char[] ch = s.toCharArray();
        for(int i=0;i<s.length();i++)
        {
            if(!hm.containsKey(ch[i]))
                hm.put(ch[i],i);
            else
                hm.put(ch[i],-1);
        }


        //Iterate over string to find first non repeated and check count in HashMap
        for(Map.Entry<Character,Integer> m : hm.entrySet())
        {
            if(m.getValue() != -1)
                return m.getValue();
        }
    return -1;
    }
}
```

## 11. MOST COMMON WORD

Given a paragraph and a list of banned words, return the most frequent word that is not in the list of banned words. It is guaranteed there is at least one word that isn't banned, and that the answer is unique.
Words in the list of banned words are given in lowercase, and free of punctuation. Words in the paragraph are not case sensitive. The answer is in lowercase.
**Example**:
**Input**:
paragraph = "Bob hit a ball, the hit BALL flew far after it was hit."
banned = ["hit"]
**Output**: “ball"
**Explanation**:
"hit" occurs 3 times, but it is a banned word.
"ball" occurs twice (and no other word does), so it is the most frequent non-banned word in the paragraph.
Note that words in the paragraph are not case sensitive,
that punctuation is ignored (even if adjacent to words, such as "ball,"),
and that "hit" isn't the answer even though it occurs more because it is banned.

```
class Solution {
    public String mostCommonWord(String paragraph, String[] banned) {
        HashMap<String, Integer> wordCount = new HashMap<>();
        int maxCount = 0;
        String ans="";
        Set<String> bannedWords = new HashSet<>(Arrays.asList(banned));
        //Convert every letter to lowercase
        paragraph = paragraph.toLowerCase();
        //remove all the punctuations
        paragraph = paragraph.replaceAll("\\p{Punct}"," ");
        //split using whitespace and insert word in array
        String[] words = paragraph.split("\\s+");
        //insert words with count in hashmap
        for(int i=0; i<words.length; i++){
          String word = words[i];
            if (!bannedWords.contains(word)) {
                    int newCount = wordCount.getOrDefault(word, 0) + 1;
                    wordCount.put(word, newCount);
                    if (newCount > maxCount) {
                        ans = word;
                        maxCount = newCount;
                    }
                }
        }

        return ans;


    }
}
```

**12. 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 [0,1,0,2,1,0,1,3,2,1,2,1]. In this case, 6 units of rain water (blue section) are being trapped.

## Constraints:

- n == height.length
- 0 <= n <= 3 * 104
- 0 <= height[i] <= 105

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

        int max_height=0;
        int max_height_ind =0;
        int temp_store=0;
        int store=0;

        if(height.length == 0 || height.length == 1)
        {
            return 0;
        }

        //find max height
        for(int i=0 ; i < height.length ; i++)
        {
            if(height[i] >= max_height)
            {
                max_height = height[i];
                max_height_ind = i;
```

```
            }
        }

        //left to max ht.
        for(int i=0, j=1; j<=max_height_ind ; )
        {
            if(height[i] > height[j]){
                temp_store += height[i]-height[j];
                j++;
            }
            else{
                store+=temp_store;
                temp_store=0;
                i=j;
                j++;
            }
        }

        //right to max ht.
        for(int i=height.length-1, j=i-1; j>=max_height_ind ; )
        {
            if(height[j] < height[i]){
                temp_store += height[i]-height[j];
                j--;
            }
            else{
                store+=temp_store;
                temp_store=0;
                i=j;
                j--;
            }
        }

        return store;


    }
}
```

## 13. 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 substring.

**Example 4:**
Input: s = ""
Output: 0

**Constraints:**

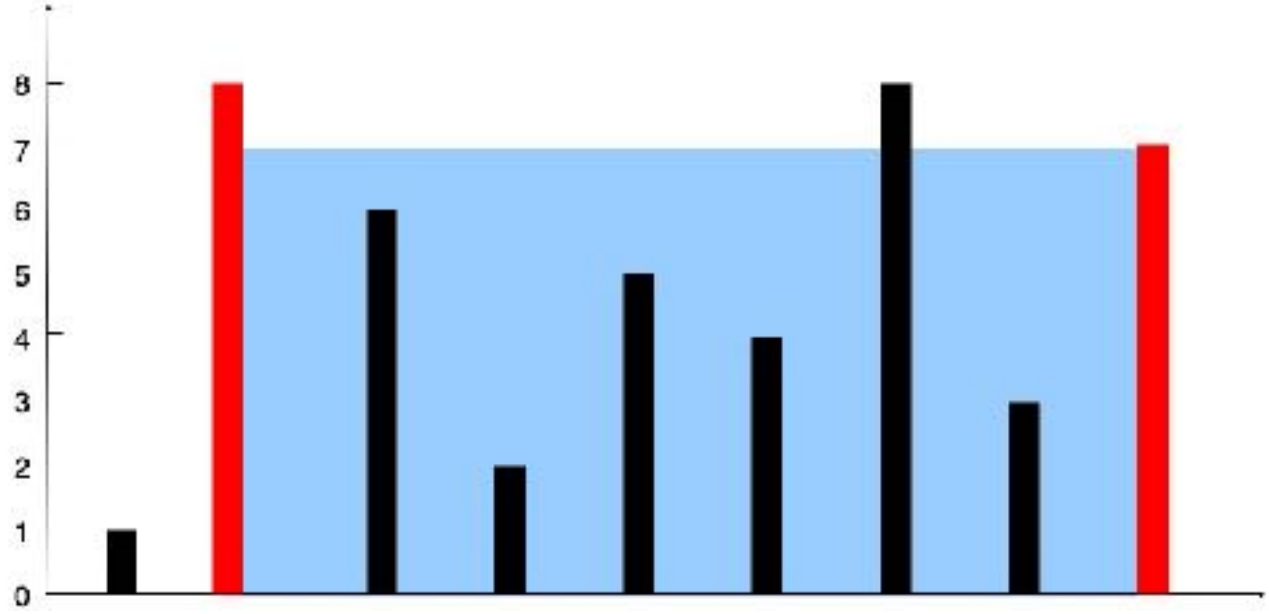
- 0 <= s.length <= 5 * 104
- s consists of English letters, digits, symbols and spaces.

```
class Solution {
    public int lengthOfLongestSubstring(String s) {

        int n = s.length();
        Set<Character> set = new HashSet<>();
        int ans = 0, i = 0, j = 0;
        while (i < n && j < n) {
            // try to extend the range [i, j]
            if (!set.contains(s.charAt(j))){
                set.add(s.charAt(j++));
                ans = Math.max(ans, j - i);
            }
            else {
                set.remove(s.charAt(i++));
            }
        }
        return ans;
    }
}
```

## 14. CONTAINER WITH MOST WATER

Given n non-negative integers a1, a2, ..., an , where each represents a point at coordinate (i, ai). n vertical lines are drawn such that the two endpoints of the line i is at (i, ai) 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 [1,8,6,2,5,4,8,3,7]. In this case, the max area of water (blue section) the container can contain is 49.

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

        int maxarea = 0, l = 0, r = height.length - 1;

        while (l < r) {
            maxarea = Math.max(maxarea, Math.min(height[l], height[r]) * (r - l));

            if (height[l] < height[r])
                l++;
            else
                r--;
        }
        return maxarea;
    }
}
```

**15. 3SUM CLOSEST**

Given an array nums of n integers 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).

Constraints:

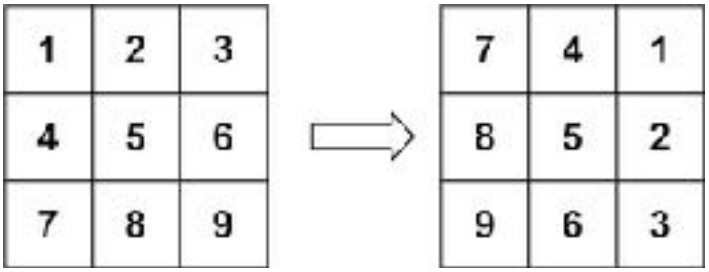
- 3 <= nums.length <= 10^3
- -10^3 <= nums[i] <= 10^3
- -10^4 <= target <= 10^4

```
class Solution {
    public int threeSumClosest(int[] nums, int target) {
        Arrays.sort(nums);
        int i, j, k, sum, bestSum, targetWidth, prevTargetWidth;
        sum = bestSum = targetWidth = prevTargetWidth = Integer.MAX_VALUE;

        for(i = 0, j = 1, k = nums.length-1; i+1 < k;) {

            sum = nums[i] + nums[j] + nums[k];
            targetWidth = Math.abs(sum - target);

            // System.out.println(sum + " " + clsum);
            if(targetWidth == 0){
                return sum;
            } else if(sum > target) {
                k--;
            } else {
                j++;
            }
            if(j == k) {
                i++;
                j = i+1;
                k = nums.length-1;
            }
            if(targetWidth < prevTargetWidth){
                prevTargetWidth = targetWidth;
                bestSum = sum;
            }
        }
        return bestSum;
    }
```

## 16. ROTATE IMAGE

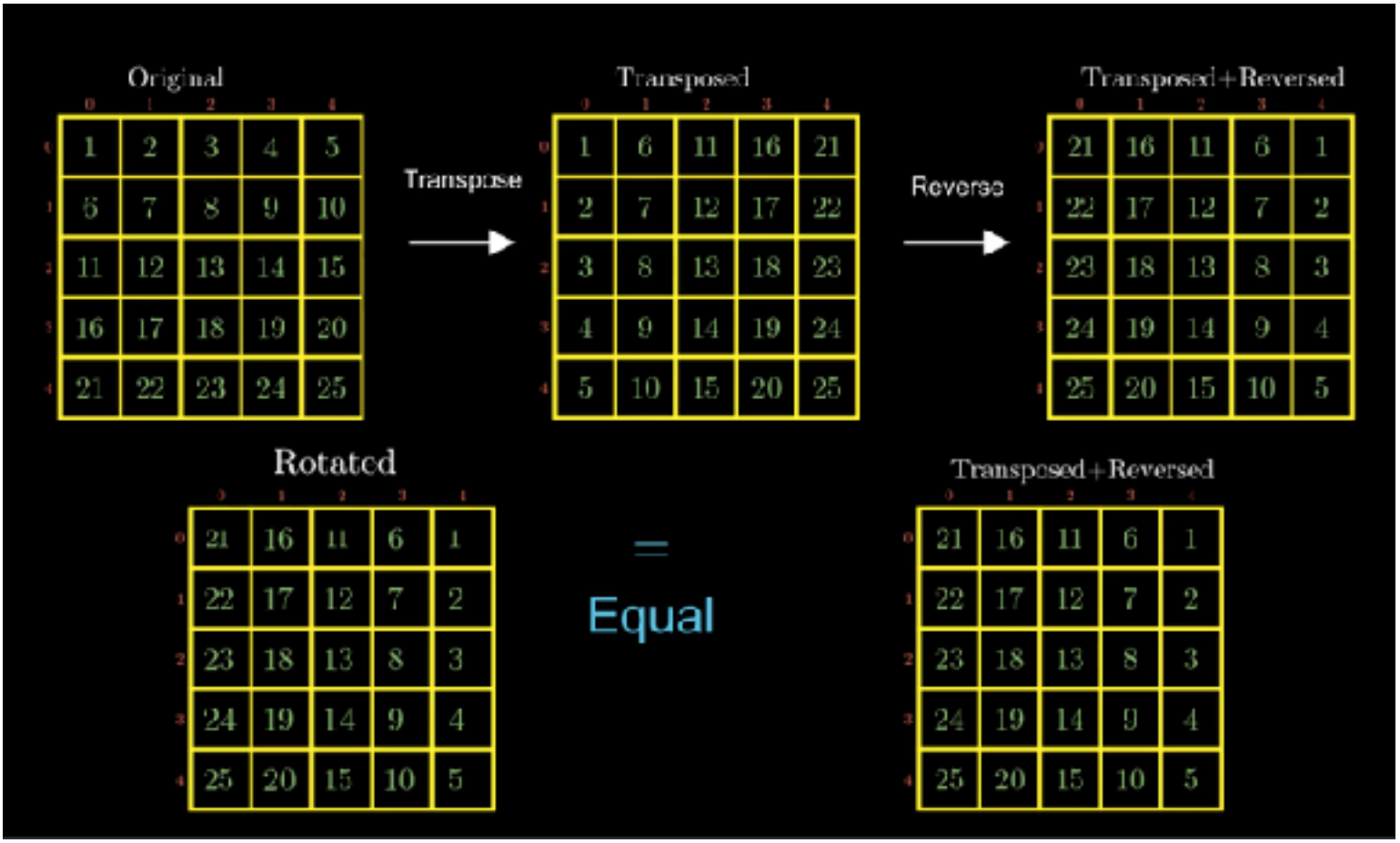
You are given an n x n 2D matrix representing an image, rotate the image by 90 degrees (clockwise).
You have to rotate the image in-place, which means you have to modify the input 2D matrix directly. DO NOT allocate another 2D matrix and do the rotation.

Example 1:




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

## SOLUTION: TRANSPOSE AND THEN REVERSE



```
class Solution {
  public void rotate(int[][] matrix) {
    int n = matrix.length;

    // transpose matrix
    for (int i = 0; i < n; i++) {
      for (int j = i; j < n; j++) {
        int tmp = matrix[j][i];
        matrix[j][i] = matrix[i][j];
        matrix[i][j] = tmp;
      }
    }

    // reverse each row
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < n / 2; j++) {
        int tmp = matrix[i][j];
        matrix[i][j] = matrix[i][n - j - 1];
        matrix[i][n - j - 1] = tmp;
      }
    }
  }
}
```

## 17. MINIMUM WINDOW SUBSTRING

Given two strings s and t, return the minimum window in s which will contain all the characters in t. If there is no such window in s that covers all characters in t, return the empty string "".
Note that If there is such a window, it is guaranteed that there will always be only one unique minimum window in s.

**Example 1:**
Input: s = "ADOBECODEBANC", t = "ABC"
Output: “BANC"

**Example 2:**
Input: s = "a", t = "a"
Output: "a"

**Constraints:**

- 1 <= s.length, t.length <= 105
- s and t consist of English letters.

```
public class Solution {
    public String minWindow(String s, String t) {
        if (s.length() == 0 || t.length() == 0 || s.length() < t.length()) return "";
        String result = "";
        int left = t.length(), start = 0, end = s.length();
        Deque<Integer> q = new ArrayDeque<>();
        Map<Character, Integer> m = new HashMap<>();

        for (char c: t.toCharArray()) m.put(c, m.getOrDefault(c, 0)+1);

        for (int i = 0; i < s.length(); i++) {
            char c = s.charAt(i);
            if (!m.containsKey(c)) continue;
            if (m.get(c) > 0) left--;
            m.put(c, m.get(c) -1);
            q.add(i);

            char head = s.charAt(q.peek());
            while (m.get(head) < 0) {
                q.poll();
                m.put(head, m.get(head)+1);
                head = s.charAt(q.peek());
            }
            if (left == 0 &&( q.peekLast() - q.peek() < end - start)) {
                start = q.peek();
                end = q.peekLast() + 1;
                result = s.substring(start,end);
            }
        }
        return result;
    }
}
```

## 18. PRODUCT OF ARRAY EXCEPT SELF

Given an array nums of n integers where n > 1,  return an array output such that output[i] is equal to the product of all the elements of nums except nums[i].

**Example**:
**Input**:  [1,2,3,4]
**Output**: [24,12,8,6]

**Constraint**: It's guaranteed that the product of the elements of any prefix or suffix of the array (including the whole array) fits in a 32 bit integer.

**Note**: Please solve it without division and in O(n).
**Follow up:**
Could you solve it with constant space complexity? (The output array does not count as extra space for the purpose of space complexity analysis.)

```
class Solution {
    public int[] productExceptSelf(int[] nums) {
        // The length of the input array
        int length = nums.length;
        // Final answer array to be returned
        int[] answer = new int[length];
        // answer[i] contains the product of all the elements to the left
        // Note: for the element at index '0', there are no elements to the left,
        // so the answer[0] would be 1
        answer[0] = 1;
        for (int i = 1; i < length; i++) {
            // answer[i - 1] already contains the product of elements to the left of 'i - 1'
            // Simply multiplying it with nums[i - 1] would give the product of all
            // elements to the left of index 'i'
            answer[i] = nums[i - 1] * answer[i - 1];
        }

        // R contains the product of all the elements to the right
        // Note: for the element at index 'length - 1', there are no elements to the right,
        // so the R would be 1
        int R = 1;
        for (int i = length - 1; i >= 0; i--) {

            // For the index 'i', R would contain the
            // product of all elements to the right. We update R accordingly
            answer[i] = answer[i] * R;
            R *= nums[i];
        }

        return answer;
    }
}
```

**19. INTEGER TO ENGLISH WORDS**

Convert a non-negative integer num to its English words representation.

**Example 1:**
Input: num = 123
Output: "One Hundred Twenty Three”

**Example 2:**
Input: num = 12345
Output: "Twelve Thousand Three Hundred Forty Five"

**Example 3:**
Input: num = 1234567
Output: "One Million Two Hundred Thirty Four Thousand Five Hundred Sixty Seven"

**Example 4:**
Input: num = 1234567891
Output: "One Billion Two Hundred Thirty Four Million Five Hundred Sixty Seven Thousand Eight Hundred Ninety One"

**Constraints:**

- 0 <= num <= 231 - 1

```
class Solution {

    String tillNine[] = {"","One ", "Two ", "Three ", "Four ", "Five ", "Six ", "Seven ", "Eight ", "Nine “};

    String tens[] = {"","", "Twenty ", "Thirty ", "Forty ", "Fifty ", "Sixty ", "Seventy ", "Eighty ", "Ninety "};

    String teens[] = {"Ten ", "Eleven ", "Twelve ", "Thirteen ", "Fourteen ", "Fifteen ", "Sixteen ", "Seventeen ", "Eighteen ", "Nineteen "};

    int BILLION = 1000000000;
    int MILLION = 1000000;
    int THOUSAND = 1000;
    int HUNDRED = 100;
    int TWENTY = 20;
    int TEN = 10;

    public String numberToWords(int num) {
        if(num==0)  return "Zero";
        return toWords(num).trim();
    }

    public String toWords(int num) {
        if(num > 0) {
            if(num >= BILLION) {
```

```
            return toWords(num/BILLION) + "Billion " + toWords(num%BILLION);
        } else if(num >= MILLION) {
            return toWords(num/MILLION) + "Million " + toWords(num%MILLION);
        } else if(num >= THOUSAND) {
            return toWords(num/THOUSAND) + "Thousand " +
toWords(num%THOUSAND);
        } else if(num >= HUNDRED) {
            return toWords(num/HUNDRED) + "Hundred " + toWords(num%HUNDRED);
        } else if(num >= TWENTY) {
            return tens[num/TEN] + toWords(num%TEN);
        } else if(num >= TEN) {
            return teens[num%10];
        } else {
            return tillNine[num];
        }
      }
      return "";
   }
}
```

## 20. VALID PARENTHESIS

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

**Constraints:**

- 1 <= s.length <= 104
- s consists of parentheses only '()[]{}'.

```
class Solution {
    public boolean isValid(String s) {

        // Using ArrayDeque is faster than using Stack class
        Deque<Character> stack = new ArrayDeque<Character>();

        // Traversing the Expression
        for (int i = 0; i < s.length(); i++)
        {
            char x = s.charAt(i);

            if (x == '(' || x == '[' || x == '{')
            {
                // Push the element in the stack
                stack.push(x);
                continue;
            }

            // IF current current character is not opening bracket, then it must be
            // closing. So, stack cannot be empty at this point.

            if (stack.isEmpty())
                return false;
            char check;
            switch (x) {
            case ')':
                check = stack.pop();
                if (check == '{' || check == '[')
                    return false;
```

```
            break;

        case '}':
            check = stack.pop();
            if (check == '(' || check == '[')
                return false;
            break;

        case ']':
            check = stack.pop();
            if (check == '(' || check == '{')
                return false;
            break;
        }
    }

    // Check Empty Stack
    return (stack.isEmpty());

  }
}
```

## 21. REORDER DATA IN LOG FILES

You have an array of logs. Each log is a space delimited string of words.
For each log, the first word in each log is an alphanumeric identifier. Then, either:

- Each word after the identifier will consist only of lowercase letters, or;
- Each word after the identifier will consist only of digits.

We will call these two varieties of logs letter-logs and digit-logs. It is guaranteed that each log has at least one word after its identifier.
Reorder the logs so that all of the letter-logs come before any digit-log. The letter-logs are ordered lexicographically ignoring identifier, with the identifier used in case of ties. The digit-logs should be put in their original order.
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","let3 art zero"]
Output: ["let1 art can","let3 art zero","let2 own kit dig","dig1 8 1 5 1","dig2 3 6"]

**Constraints:**

- 0 <= logs.length <= 100
- 3 <= logs[i].length <= 100
- logs[i] is guaranteed to have an identifier, and a word after the identifier.

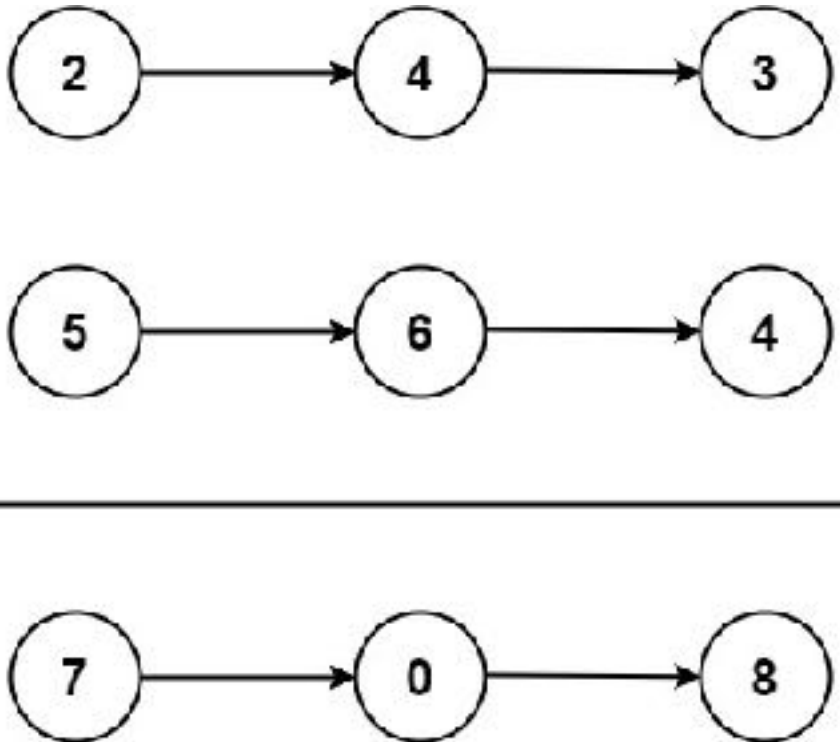
```
class Solution {
    public String[] reorderLogFiles(String[] logs) {
        Arrays.sort(logs, new Comparator<String>(){
            @Override
            public int compare(String s1, String s2) {
                String[] first = s1.split(" ", 2);
                String[] second = s2.split(" ", 2);
                boolean isdig1 = Character.isDigit(first[1].charAt(0));
                boolean isdig2 = Character.isDigit(second[1].charAt(0));
                if(isdig1 && isdig2) {
                    return 0;
                } else if(isdig2) {
                    return -1;
                } else if(isdig1) {
                    return 1;
                }

                if(first[1].compareTo(second[1]) == 0) {
                    return s1.compareTo(s2);
                } else {
                    return first[1].compareTo(second[1]);
                }
            }
        });
             return logs;
    }
}
```

**22. 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 <= Node.val <= 9
- It is guaranteed that the list represents a number that does not have leading zeros.

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */


public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
    ListNode dummyHead = new ListNode(0);
    ListNode p = l1, q = l2, curr = dummyHead;

    int carry = 0;
    while (p != null || q != null) {
        int x = (p != null) ? p.val : 0;
        int y = (q != null) ? q.val : 0;
        int sum = carry + x + y;
        carry = sum / 10;
        curr.next = new ListNode(sum % 10);
        curr = curr.next;
        if (p != null) p = p.next;
        if (q != null) q = q.next;
    }
    if (carry > 0) {
        curr.next = new ListNode(carry);
    }
    return dummyHead.next;
}
```

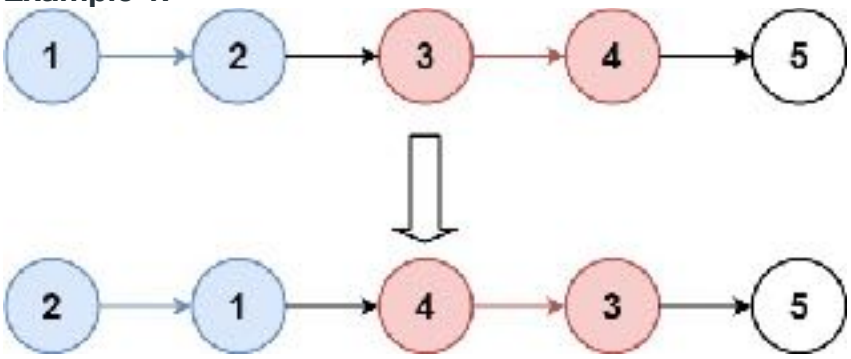
## 23. REVERSE NODES IN K GROUPS

Given a linked list, reverse the nodes of a linked list k at a time and return its modified list. k is a positive integer and is less than or equal to the length of the linked list. If the number of nodes is not a multiple of k then left-out nodes, in the end, should remain as it is.

**Follow up:**

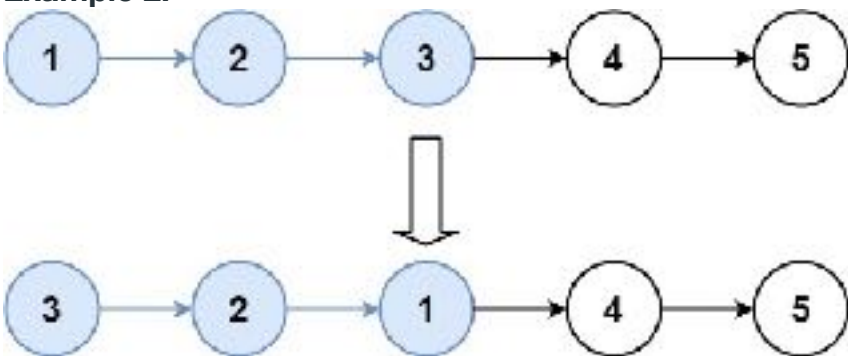
- Could you solve the problem in O(1) extra memory space?
- You may not alter the values in the list's nodes, only nodes itself may be changed.

**Example 1:**




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

**Example 2:**




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

**Constraints:**

- The number of nodes in the list is in the range sz.
- 1 <= sz <= 5000
- 0 <= Node.val <= 1000
- 1 <= k <= sz

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */


class Solution {
    public ListNode reverseKGroup(ListNode head, int k) {
        int count = 0;
        ListNode p = head;
        while(p!=null) {
            if(++count == k){
                break;
            }
            p = p.next;
        }
        if(count < k){
            return head;
        }
        ListNode last = reverseKGroup(p.next, k);

        ListNode prev = null;
        ListNode cur = head;
        ListNode next = null;
        for(int i = 0; i<k; i++) {
            next = cur.next;
            cur.next = prev;
            prev = cur;
            cur = next;
        }
        head.next = last;
        return prev;
    }
    }
}
```

## 24. REVERSE LINKED LIST

Reverse a singly linked list.

**Example**:
Input: 1->2->3->4->5->NULL
Output: 5->4->3->2->1->NULL

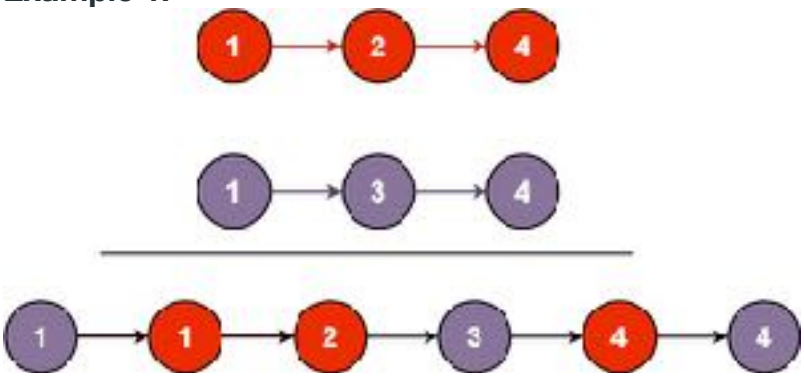
**Follow up:**
A linked list can be reversed either iteratively or recursively. Could you implement both?

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */

class Solution {
    public ListNode reverseList(ListNode head) {
        if(head == null || head.next == null) {
            return head;
        }
        ListNode head_node = reverseList(head.next);
        head.next.next = head;
        head.next = null;
        return head_node;
    }
}
```

## 25. 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 <= Node.val <= 100
- Both l1 and l2 are sorted in non-decreasing order.

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */

class Solution {
    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        if(l1 == null) return l2;
        if(l2 == null) return l1;

        ListNode head;
        if(l1.val < l2.val) {
            head = l1;
            head.next = mergeTwoLists(l1.next, l2);
        } else {
```

```
            head = l2;
            head.next = mergeTwoLists(l1, l2.next);
        }
        return head;


        // ListNode p = new ListNode(Integer.MIN_VALUE);
        // head = p;
        // while(l1 != null && l2 != null) {
        //     if(l1.val < l2.val) {
        //         p.next = l1;
        //         l1 = l1.next;
        //     } else {
        //         p.next = l2;
        //         l2 = l2.next;
        //     }
        //     p = p.next;
        // }
        // if(l1 != null){
        //     p.next = l1;
        // } else {
        //     p.next = l2;
        // }
        // return head.next;


    }
}
```

## 26. COPY LIST WITH RANDOM POINTER

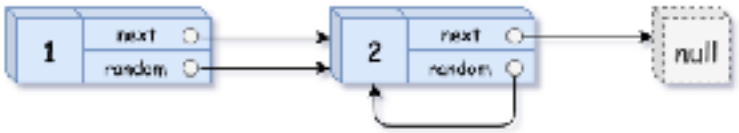
A linked list is given such that each node contains an additional random pointer which could point to any node in the list or null.
Return a deep copy of the list.
The Linked List is represented in the input/output as a list of n nodes. Each node is represented as a pair of [val, random_index] where:

- val: an integer representing Node.val
- random_index: the index of the node (range from 0 to n-1) where random pointer points to, or null if it does not point to any node.

**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: []

**Constraints**:

- -10000 <= Node.val <= 10000
- Node.random is null or pointing to a node in the linked list.
- The number of nodes will not exceed 1000.

```
/*
class Node {
    public int val;
    public Node next;
    public Node random;
    public Node() {}
    public Node(int _val,Node _next,Node _random) {
        val = _val;
        next = _next;
        random = _random;
    } }; */

class Solution {
    public Node copyRandomList(Node headMain) {
        Map<Node, Node> m = new HashMap<>();
        Node tmpRandom = null, tmpNext = null;
        Node head = headMain;
        Node curNode;
        while(head != null) {
            if(m.get(head) == null) {
                curNode = new Node(head.val, null, null);
            } else {
                curNode = m.get(head);
            }
            if(head.random == null){
                tmpRandom = null;
            } else if(head.random == head) {
                tmpRandom = curNode;
            } else{
                tmpRandom = m.get(head.random);
                if(tmpRandom == null) {
                    tmpRandom = new Node(head.random.val, null, null);
                    m.put(head.random, tmpRandom);
                }
            }
            if(head.next == null){
                tmpNext = null;
            } else{
                tmpNext = m.get(head.next);
                if(tmpNext == null) {
                    tmpNext = new Node(head.next.val, null, null);
                    m.put(head.next, tmpNext);
                }
            }
            curNode.next = tmpNext;
            curNode.random = tmpRandom;
            m.put(head, curNode);
            head = head.next;
        }
        return m.get(headMain);
    }
}
```

## 27. MERGE K-SORTED LIST

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.

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
```

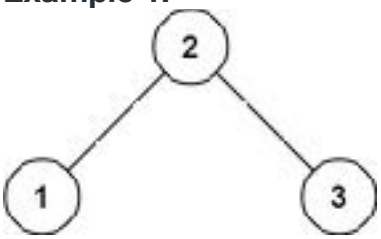
```
class Solution {
    public ListNode mergeKLists(ListNode[] lists) {
        if(lists.length == 1) {
            return lists[0];
        }
        if(lists.length == 0){
            return null;
        }
        Comparator<ListNode> cm = new Comparator<ListNode>(){
            @Override
            public int compare(ListNode l1, ListNode l2) {
                return l1.val-l2.val;
            }
        };
        Queue<ListNode> q = new PriorityQueue<>(cm);
        for(ListNode l:lists){
            if(l !=null)
            q.offer(l);
        }
        ListNode res = new ListNode(0);
        ListNode head = res;
        ListNode p = res;
        while(!q.isEmpty()){
            ListNode tmp = q.poll();
            p.next = tmp;
            p = p.next;
            if(tmp.next != null){
                q.offer(tmp.next);
            }
        }
        return head.next;
    }

}
```

## 28. 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:

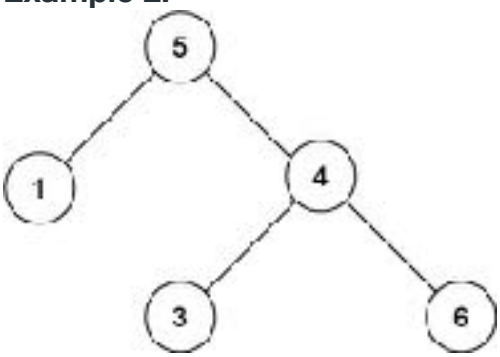
- •The left subtree of a node contains only nodes with keys less than the node's key.
- •The right subtree of a node contains only nodes with keys greater than the node's key.
- •Both the left and right subtrees must also be binary search trees.

**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, 104].
- -231 <= Node.val <= 231 - 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 boolean validate(TreeNode root, Integer low, Integer high) {
        // Empty trees are valid BSTs.
        if (root == null) {
            return true;
        }
       // The current node's value must be between low and high.
        if ((low != null && root.val <= low) || (high != null && root.val >= high)) {
            return false;
        }
        // The left and right subtree must also be valid.
        return validate(root.right, root.val, high) && validate(root.left, low, root.val);
    }
    public boolean isValidBST(TreeNode root) {
        return validate(root, null, null);
    }
}

class Solution {

    private Deque<TreeNode> stack = new LinkedList();
    private Deque<Integer> upperLimits = new LinkedList();
    private Deque<Integer> lowerLimits = new LinkedList();

    public void update(TreeNode root, Integer low, Integer high) {
        stack.add(root);
        lowerLimits.add(low);
        upperLimits.add(high);
    }

    public boolean isValidBST(TreeNode root) {
        Integer low = null, high = null, val;
        update(root, low, high);
        while (!stack.isEmpty()) {
            root = stack.poll();
            low = lowerLimits.poll();
            high = upperLimits.poll();

            if (root == null) continue;
            val = root.val;
            if (low != null && val <= low) {
                return false;
            }
            if (high != null && val >= high) {
                return false;
            }
            update(root.right, val, high);
            update(root.left, low, val);
        }
        return true;
    }
}
```

## 29. BINARY TREE LEVEL ORDER TRAVERSAL

Given a binary tree, return the level order traversal of its nodes' values. (ie, from left to right, level by level).

**For example:**
Given binary tree [3,9,20,null,null,15,7],

```
    3
   / \
  9  20
    /  \
   15   7
```

return its level order traversal as:

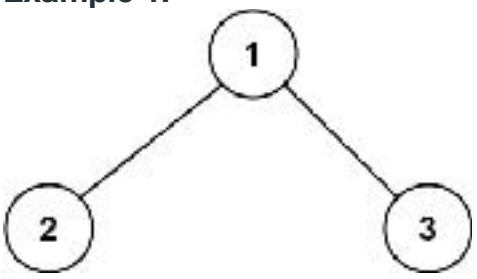
```
[
  [3],
  [9,20],
  [15,7]
]
```

```
/**
 * 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;
 *     }
 * }
 */

public List<List<Integer>> levelOrder(TreeNode root) {
        List<List<Integer>> res = new ArrayList<List<Integer>>();
        levelHelper(res, root, 0);
        return res;
    }

  public void levelHelper(List<List<Integer>> res, TreeNode root, int height) {
        if (root == null) return;
        if (height >= res.size()) {
            res.add(new LinkedList<Integer>());
        }
        res.get(height).add(root.val);
        levelHelper(res, root.left, height+1);
        levelHelper(res, root.right, height+1);
    }
```

## 30. BINARY TREE MAXIMUM PATH SUM

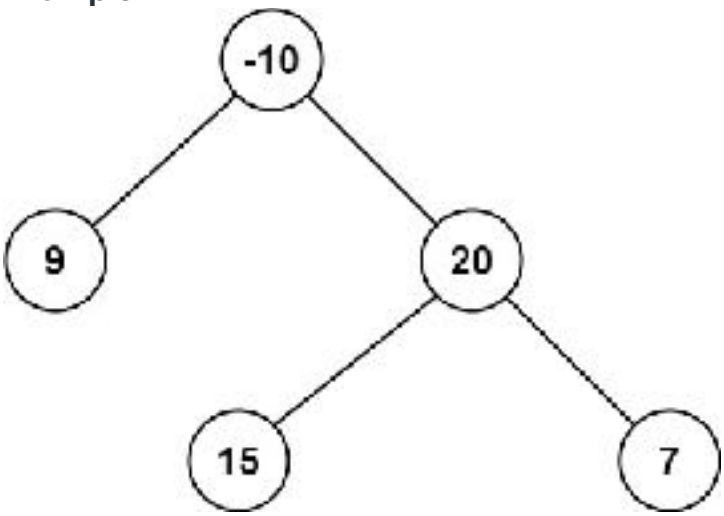
Given the root of a binary tree, return the maximum path sum.
For this problem, a path is defined as any node sequence from some starting node to any node in the tree along the parent-child connections. The path must contain at least one node and does not need to go through the root.

**Example 1:**




Input: root = [1,2,3]
Output: 6

**Example 2:**




Input: root = [-10,9,20,null,null,15,7]
Output: 42

**Constraints:**

- The number of nodes in the tree is in the range [1, 3 * 104].
- -1000 <= Node.val <= 1000

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
```

```
class Solution {

    int max_sum = Integer.MIN_VALUE;

    public int maxPathSum(TreeNode root) {
        if(root.left == null && root.right == null) return root.val;
        util(root);
        return max_sum;
    }

    public int util(TreeNode root) {
        if(root == null) return 0;

        int left = util(root.left);
        int right = util(root.right);
        int sum = root.val;

        if(left > 0) sum += left;

        if(right > 0) sum += right;

        max_sum = Math.max(sum, max_sum);

        return Math.max(Math.max(left, right)+root.val , root.val);
    }
}
```

**31. WORD LADDER**

Given two words beginWord and endWord, and a dictionary wordList, return the length of the shortest transformation sequence from beginWord to endWord, such that:

- Only one letter can be changed at a time.
- Each transformed word must exist in the word list.

Return 0 if there is no such transformation sequence.

**Example 1:**

Input: beginWord = "hit",
endWord = "cog",
wordList = [“hot","dot","dog","lot","log","cog"]

Output: 5

Explanation: As one shortest transformation is "hit" -> "hot" -> "dot" -> "dog" -> "cog", return its length 5.

**Example 2:**
Input: beginWord = "hit",
endWord = "cog",
wordList = [“hot","dot","dog","lot","log"]

Output: 0

Explanation: The endWord "cog" is not in wordList, therefore no possible transformation.

**Constraints**:

- 1 <= beginWord.length <= 100
- endWord.length == beginWord.length
- 1 <= wordList.length <= 5000
- wordList[i].length == beginWord.length
- beginWord, endWord, and wordList[i] consist of lowercase English letters.
- beginWord != endWord
- All the strings in wordList are unique.

```
class Solution {
    public int ladderLength(String beginWord, String endWord, List<String> wordList) {

        Set<String> list = new HashSet<>(wordList);
        Queue<String> q = new LinkedList<>();

        if(!list.contains(endWord)) return 0;

        int count = 1;
        q.offer(beginWord);

        while(!q.isEmpty()) {
            int size = q.size();

            for(int i = 0; i < size; i++) {
                char[] cur = q.poll().toCharArray();

                for(int j = 0; j < cur.length; j++) {
                    char tmp = cur[j];

                    for(char k = 'a'; k <= 'z'; k++) {
                        cur[j] = k;
                        String twisted = String.valueOf(cur);

                        if(list.contains(twisted)){
                                if(twisted.equals(endWord)) {
                                    return count + 1;
                                }
                                q.offer(twisted);
                                list.remove(twisted);
                            }
                        }
                        cur[j] = tmp;
                    }
                }
                count++;
            }
            return 0;
        }
}
```

## 32. COURSE SCHEDULE

There are a total of numCourses courses you have to take, labeled from 0 to numCourses-1.
Some courses may have prerequisites, for example to take course 0 you have to first take course 1, which is expressed as a pair: [0,1]
Given the total number of courses and a list of prerequisite pairs, is it possible for you to finish all courses?

**Example 1:**
Input: numCourses = 2, prerequisites = [[1,0]]
Output: true
Explanation: There are a total of 2 courses to take. To take course 1 you should have finished course 0. So it is possible.

**Example 2:**
Input: numCourses = 2, prerequisites = [[1,0],[0,1]]
Output: false
Explanation: There are a total of 2 courses to take. To take course 1 you should have finished course 0, and to take course 0 you should also have finished course 1. So it is impossible.

**Constraints:**
•The input prerequisites is a graph represented by a list of edges, not adjacency matrices. Read more about how a graph is represented.

•You may assume that there are no duplicate edges in the input prerequisites.

•1 <= numCourses <= 10^5

```
class Solution {
    public boolean canFinish(int numCourses, int[][] prerequisites) {
        if (numCourses == 0 || prerequisites.length == 0) {
            return true;
        }

        Map<Integer, List<Integer>> edges = new HashMap<>();
        for (int[] pair : prerequisites) {
            if (edges.containsKey(pair[0])) {
                edges.get(pair[0]).add(pair[1]);
            } else {
                List<Integer> dependencies = new LinkedList<Integer>();
                dependencies.add(pair[1]);
                edges.put(pair[0], dependencies);
            }
        }
```

```
        Set<Integer> passed = new HashSet<>();
        Set<Integer> marked = new HashSet<>();
        for (int i = 0; i < numCourses; i ++) {
            if (passed.contains(i)) {
                continue;
            }

            if (containsCycle(i, marked, edges, passed)) {
                return false;
            }
        }

        return true;
    }

    private boolean containsCycle(int root, Set<Integer> marked, Map<Integer,
List<Integer>> edges, Set<Integer> passed) {
        // Don't go looking for cycles in sub-graphs we already know
        // don't have cycles because we've checked them already.
        if (passed.contains(root)) {
            return false;
        }

        // We've seen the root node before, which means that it must
        // have appeard as a root node earlier in the current path
        // we're walking in the sub-graph. In other words, there's a cycle.
        if (marked.contains(root)) {
            return true;
        }


        // Mark the root. If we see it again before we're done with
        // its sub-graph, we know there's a cycle in its sub-graph.
        marked.add(root);
        for (int dependency : edges.getOrDefault(root, Collections.<Integer>emptyList())) {
            if (containsCycle(dependency, marked, edges, passed)) {
                return true;
            }
        }

        // "Pass" the root because we didn't see any cycles in its sub-graph.
        passed.add(root);
        return false;
    }
}
```

## 33. DIAMETER OF A BINARY TREE

Given a binary tree, you need to compute 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.

**Example:**
Given a binary tree

Return 3, which is the length of the path [4,2,1,3] or [5,2,1,3].
**Note**: The length of path between two nodes is represented by the number of edges between them.

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */

class Solution {
    int max = 0;

    public int diameterOfBinaryTree(TreeNode root) {
        util(root);
        return max;
    }

    public int util(TreeNode root){
        if(root == null) {
            return 0;
        }

        int left = util(root.left);
        int right = util(root.right);

        max = Math.max(max, left+right);

        return Math.max(left, right) + 1;
    }
}
```

**34. FLOOD FILL**

An image is represented by a 2-D array of integers, each integer representing the pixel value of the image (from 0 to 65535).
Given a coordinate (sr, sc) representing the starting pixel (row and column) of the flood fill, and a pixel value newColor, "flood fill" the image.
To perform a "flood fill", consider the starting pixel, plus any pixels connected 4-directionally to the starting pixel of the same color as the starting pixel, plus any pixels connected 4-directionally to those pixels (also with the same color as the starting pixel), and so on. Replace the color of all of the a fore mentioned pixels with the newColor.
At the end, return the modified image.
**Example 1:**
Input:
image = [[1,1,1],[1,1,0],[1,0,1]]
sr = 1, sc = 1, newColor = 2
Output: [[2,2,2],[2,2,0],[2,0,1]]
Explanation:
From the center of the image (with position (sr, sc) = (1, 1)), all pixels connected by a path of the same color as the starting pixel are colored with the new color.
Note the bottom corner is not colored 2, because it is not 4-directionally connected to the starting pixel.

**Note:**
The length of image and image[0] will be in the range [1, 50].
The given starting pixel will satisfy 0 <= sr < image.length and 0 <= sc < image[0].length.
The value of each color in image[i][j] and newColor will be an integer in [0, 65535].

**Approach #1: Depth-First Search**
**Intuition -** paint the starting pixels, plus adjacent pixels of the same color, and so on.
**Algorithm**
Say color is the color of the starting pixel. Let's floodfill the starting pixel: we change the color of that pixel to the new color, then check the 4 neighboring pixels to make sure they are valid pixels of the same color, and of the valid ones, we floodfill those, and so on.
We can use a function dfs to perform a floodfill on a target pixel.

```
class Solution {
    public int[][] floodFill(int[][] image, int sr, int sc, int newColor) {
        int color = image[sr][sc];
        if (color != newColor) dfs(image, sr, sc, color, newColor);
        return image;
    }
    public void dfs(int[][] image, int r, int c, int color, int newColor) {
        if (image[r][c] == color) {
            image[r][c] = newColor;
            if (r >= 1) dfs(image, r-1, c, color, newColor);
            if (c >= 1) dfs(image, r, c-1, color, newColor);
            if (r+1 < image.length) dfs(image, r+1, c, color, newColor);
            if (c+1 < image[0].length) dfs(image, r, c+1, color, newColor);
        }
    }
}
```

## 35. SYMMETRIC TREE

Given a binary tree, check whether it is a mirror of itself (ie, symmetric around its center).
For example, this binary tree [1,2,2,3,4,4,3] is symmetric:

```
    1
   / \
  2   2
 / \ / \
3  4 4  3
```

But the following [1,2,2,null,3,null,3] is not:

```
    1
   / \
  2   2
   \   \
   3    3
```

```
/**
 * 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;
 *     }
 * }
 */
```

**RECURSIVE SOLUTION :**

```
class Solution {
    public boolean isSymmetric(TreeNode root) {
        if(root == null) {
            return true;
        }
        return util(root.left,root.right) && util(root.right,root.left);
    }

    boolean util(TreeNode root1, TreeNode root2) {
        if(root1 == null && root2 == null) return true;
        if((root1 == null && root2 != null) || (root1 != null && root2 == null)) return false;
        return root1.val == root2.val && util(root1.right,root2.left) && util(root1.left,root2.right);
    }
}
```

**ITERATIVE SOLUTION**

```
class Solution {
    public boolean isSymmetric(TreeNode root) {

        if(root == null){
            return true;
        }

        return util(root.left, root.right);

    }
    public boolean util(TreeNode root1, TreeNode root2){

        if(root1 == null && root2 == null) return true;

        if(root1 != null && root2 != null && root1.val == root2.val){
            return util(root1.left, root2.right) && util(root1.right, root2.left);
        }
        return false;
    }
}
```

## 36. BINARY TREE ZIG ZAG LEVEL ORDER TRAVERSAL

Given a binary tree, return the zigzag level order traversal of its nodes' values. (ie, from left to right, then right to left for the next level and alternate between).

**For example:**

Given binary tree [3,9,20,null,null,15,7],

```
    3
   / \
  9  20
    /  \
   15   7
```

return its zigzag level order traversal as:

```
[
  [3],
  [20,9],
  [15,7]
]
```

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
```

```
class Solution {
    public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
        if(root == null) return new ArrayList<>();
        boolean reverse = false;
        Stack<TreeNode> cl = new Stack<>();
        Stack<TreeNode> nl = new Stack<>();

        List<List<Integer>> res = new ArrayList<List<Integer>>();
        cl.push(root);
        List<Integer> tmp = new ArrayList<>();

        while(!cl.isEmpty()) {
                TreeNode cur = cl.pop();
                tmp.add(cur.val);
                if(reverse) {
                    if(cur.right != null) {
                        nl.push(cur.right);
                    }
                    if(cur.left != null) {
                        nl.push(cur.left);
                    }
                } else {
                    if(cur.left != null) {
                        nl.push(cur.left);
                    }
                    if(cur.right != null) {
                        nl.push(cur.right);
                    }
                }
            if(cl.isEmpty()) {
                res.add(tmp);
                tmp =new ArrayList<>();
                reverse = !reverse;
                Stack<TreeNode> tmp1 = cl;
                cl = nl;
                nl = tmp1;
             }

        }
        return res;
    }
}
```

## 37. WORD LADDER II

Given two words (beginWord and endWord), and a dictionary's word list, find all shortest transformation sequence(s) from beginWord to endWord, such that:

1 Only one letter can be changed at a time

2 Each transformed word must exist in the word list. Note that beginWord is not a transformed word.

**Note:**

- Return an empty list if there is no such transformation sequence.
- All words have the same length.
- All words contain only lowercase alphabetic characters.
- You may assume no duplicates in the word list.
- You may assume beginWord and endWord are non-empty and are not the same.

**Example 1:**
Input:
beginWord = "hit",
endWord = "cog",
wordList = ["hot","dot","dog","lot","log","cog"]

Output:
[
  ["hit","hot","dot","dog","cog"],
  ["hit","hot","lot","log","cog"]
]

**Example 2:**
Input:
beginWord = "hit"
endWord = "cog"
wordList = ["hot","dot","dog","lot","log"]

Output: []

Explanation: The endWord "cog" is not in wordList, therefore no possible transformation.

The basic idea is:

1). Use BFS to find the shortest distance between start and end, tracing the distance of crossing nodes from start node to end node, and store node's next level neighbors to HashMap;

2). Use DFS to output paths with the same distance as the shortest distance from distance HashMap: compare if the distance of the next level node equals the distance of the current node + 1.

```
public List<List<String>> findLadders(String start, String end, List<String> wordList) {
   HashSet<String> dict = new HashSet<String>(wordList);
   List<List<String>> res = new ArrayList<List<String>>();
   HashMap<String, ArrayList<String>> nodeNeighbors = new HashMap<String,
ArrayList<String>>();// Neighbors for every node
   HashMap<String, Integer> distance = new HashMap<String, Integer>();// Distance of
every node from the start node
   ArrayList<String> solution = new ArrayList<String>();

   dict.add(start);
   bfs(start, end, dict, nodeNeighbors, distance);
   dfs(start, end, dict, nodeNeighbors, distance, solution, res);
   return res;
}

// BFS: Trace every node's distance from the start node (level by level).
private void bfs(String start, String end, Set<String> dict, HashMap<String,
ArrayList<String>> nodeNeighbors, HashMap<String, Integer> distance) {
  for (String str : dict)
      nodeNeighbors.put(str, new ArrayList<String>());

  Queue<String> queue = new LinkedList<String>();
  queue.offer(start);
  distance.put(start, 0);

  while (!queue.isEmpty()) {
      int count = queue.size();
      boolean foundEnd = false;
      for (int i = 0; i < count; i++) {
          String cur = queue.poll();
          int curDistance = distance.get(cur);
          ArrayList<String> neighbors = getNeighbors(cur, dict);

          for (String neighbor : neighbors) {
              nodeNeighbors.get(cur).add(neighbor);
              if (!distance.containsKey(neighbor)) {// Check if visited
                  distance.put(neighbor, curDistance + 1);
                  if (end.equals(neighbor))// Found the shortest path
                      foundEnd = true;
                  else
                      queue.offer(neighbor);
                  }
              }
          }

          if (foundEnd)
              break;
      }
  }

// Find all next level nodes.
private ArrayList<String> getNeighbors(String node, Set<String> dict) {
```

```
  ArrayList<String> res = new ArrayList<String>();
  char chs[] = node.toCharArray();

  for (char ch ='a'; ch <= 'z'; ch++) {
      for (int i = 0; i < chs.length; i++) {
          if (chs[i] == ch) continue;
          char old_ch = chs[i];
          chs[i] = ch;
          if (dict.contains(String.valueOf(chs))) {
              res.add(String.valueOf(chs));
          }
          chs[i] = old_ch;
      }

  }
  return res;
}

// DFS: output all paths with the shortest distance.
private void dfs(String cur, String end, Set<String> dict, HashMap<String,
ArrayList<String>> nodeNeighbors, HashMap<String, Integer> distance,
ArrayList<String> solution, List<List<String>> res) {
    solution.add(cur);
    if (end.equals(cur)) {
       res.add(new ArrayList<String>(solution));
    } else {
       for (String next : nodeNeighbors.get(cur)) {
            if (distance.get(next) == distance.get(cur) + 1) {
                dfs(next, end, dict, nodeNeighbors, distance, solution, res);
            }
        }
    }
   solution.remove(solution.size() - 1);
}
```

1. in dfs , thereason for if (distance.get(next) == distance.get(cur) + 1) is just in case that the next node is the next level of current node,  otherwise it can be one of the parent nodes of current node,  or it is not the shortest node . Since in BFS, we record both the next level nodes and the parent node as neighbors of current node. use distance.get(cur) +1 we can make sure the path is the shortest one.

2. in BFS , we can be sure that the distance of each node is the shortest one , because once we have visited a node, we update the distance , if we first met one node ,it must be the shortest distance. if we met the node again ,its distance must not be less than the distance we first met and set.
then I made some improvements,since dfs cost a lot of time ,so we decide whether to add a node to the neighbors when we are in BFS, we can save a lot of time,the improved code is as below, most of them are same as the original code.But it cost 184MS, beat 50%,much faster.

## 38. NUMBER OF ISLANDS

Given an m x n 2d grid map of '1's (land) and '0's (water), return the number of islands. An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

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

**Example 2:**
Input: grid = [
  ["1","1","0","0","0"],
  ["1","1","0","0","0"],
  ["0","0","1","0","0"],
  ["0","0","0","1","1"]
]
Output: 3

**Constraints:**

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

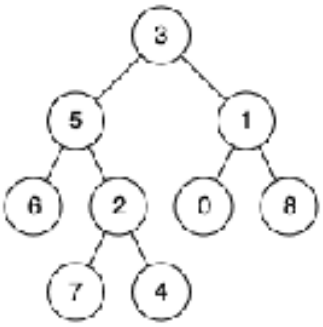
```
class Solution {
    public int numIslands(char[][] grid) {
        if(grid.length == 0) return 0;
        int r = grid.length;
        int c = grid[0].length;
        int i,j;
        int num = 0;

        for(i =0;i<r;i++) {
            for(j = 0;j<c;j++) {
                if(grid[i][j] == '1'){
                    num += dfs(grid, i, j);
                }
            }
        }
        return num;
    }
```

```
    public int dfs(char[][] grid, int i, int j) {
        if(i<0 || j>grid[0].length-1 || i>grid.length-1 || j<0 || grid[i][j] == '0'){
            return 0;
        }
        grid[i][j] = '0';
        dfs(grid, i+1, j);
        dfs(grid, i-1, j);
        dfs(grid, i, j+1);
        dfs(grid, i, j-1);
        return 1;
    }
}
```

## 39. LOWEST COMMON ANCESTOR OF A BINARY TREE

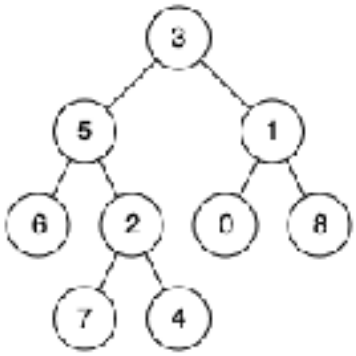
Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree. According to the definition of LCA on Wikipedia: “The lowest common ancestor is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow a node to be a descendant 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 of itself according to the LCA definition.

**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, 105].
- -109 <= Node.val <= 109
- All Node.val are unique.
- p != q
- p and q will exist in the tree.

```java
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */


class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        // System.out.println(root.val);
        if(root == null){
            return root;
        }
        if(root.val == p.val || root.val == q.val){
            return root;
        }
        TreeNode L = lowestCommonAncestor(root.left, p, q);
        TreeNode R = lowestCommonAncestor(root.right, p, q);
        if(L!= null && R!=null){
            return root;
        }
        if(L!=null) return L;
        else return R;
    }
}
```

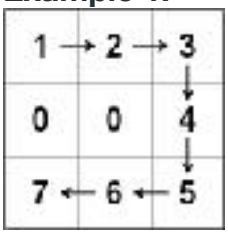
## 40. CUT OFF TREES FOR GOLF EVENT

You are asked to cut off all the trees in a forest for a golf event. The forest is represented as an m x n matrix. In this matrix:

- 0 means the cell cannot be walked through.
- 1 represents an empty cell that can be walked through.
- A number greater than 1 represents a tree in a cell that can be walked through, and this number is the tree's height.

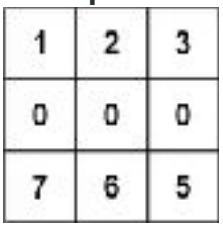
In one step, you can walk in any of the four directions: north, east, south, and west. If you are standing in a cell with a tree, you can choose whether to cut it off.
You must cut off the trees in order from shortest to tallest. When you cut off a tree, the value at its cell becomes 1 (an empty cell).
Starting from the point (0, 0), return the minimum steps you need to walk to cut off all the trees. If you cannot cut off all the trees, return -1.
You are guaranteed that no two trees have the same height, and there is at least one tree needs to be cut off.

**Example 1:**




Input: forest = [[1,2,3],[0,0,4],[7,6,5]]
Output: 6
Explanation: Following the path above allows you to cut off the trees from shortest to tallest in 6 steps.

**Example 2:**




Input: forest = [[1,2,3],[0,0,0],[7,6,5]]
Output: -1
Explanation: The trees in the bottom row cannot be accessed as the middle row is blocked.

**Example 3:**

Input: forest = [[2,3,4],[0,0,5],[8,7,6]]
Output: 6
Explanation: You can follow the same path as Example 1 to cut off all the trees.
Note that you can cut off the first tree at (0, 0) before making any steps.

Constraints:

- m == forest.length
- n == forest[i].length
- 1 <= m, n <= 50
- 0 <= forest[i][j] <= 109

**Intuition and Algorithm**

We perform a breadth-first-search, processing nodes (grid positions) in a queue. seen keeps track of nodes that have already been added to the queue at some point - those nodes will be already processed or are in the queue awaiting processing.

For each node that is next to be processed, we look at it's neighbours. If they are in the forest (grid), they haven't been enqueued, and they aren't an obstacle, we will enqueue that neighbour.

We also keep a side count of the distance travelled for each node. If the node we are processing is our destination 'target' (tr, tc), we'll return the answer.

```
public int bfs(List<List<Integer>> forest, int sr, int sc, int tr, int tc) {
    int R = forest.size(), C = forest.get(0).size();
    Queue<int[]> queue = new LinkedList();
    queue.add(new int[]{sr, sc, 0});
    boolean[][] seen = new boolean[R][C];
    seen[sr][sc] = true;
    while (!queue.isEmpty()) {
        int[] cur = queue.poll();
        if (cur[0] == tr && cur[1] == tc) return cur[2];
        for (int di = 0; di < 4; ++di) {
            int r = cur[0] + dr[di];
            int c = cur[1] + dc[di];
            if (0 <= r && r < R && 0 <= c && c < C &&
                    !seen[r][c] && forest.get(r).get(c) > 0) {
                seen[r][c] = true;
                queue.add(new int[]{r, c, cur[2]+1});
            }
        }
    }
    return -1;
}
```

## 41. LETTER COMBINATION 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"]

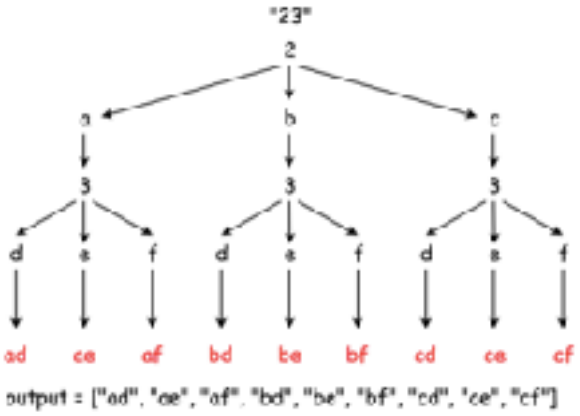
**Constraints:**

- 0 <= digits.length <= 4
- digits[i] is a digit in the range ['2', ‘9'].

Backtracking is an algorithm for finding all solutions by exploring all potential candidates. If the solution candidate turns to be not a solution (or at least not the last one), backtracking algorithm discards it by making some changes on the previous step, i.e. backtracks and then try again.

Here is a backtrack function backtrack(combination, next_digits) which takes as arguments an ongoing letter combination and the next digits to check.

If there is no more digits to check that means that the current combination is done.
If there are still digits to check :
Iterate over the letters mapping the next available digit.
Append the current letter to the current combination combination = combination + letter.
Proceed to check next digits : backtrack(combination + letter, next_digits[1:]).

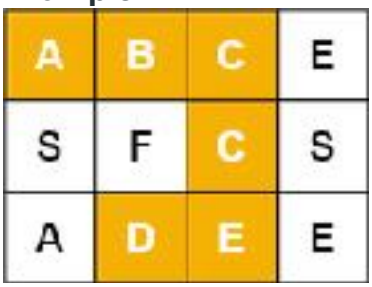



```
class Solution {
  Map<String, String> phone = new HashMap<String, String>() {{
    put("2", "abc");
    put("3", "def");
    put("4", "ghi");
    put("5", "jkl");
    put("6", "mno");
    put("7", "pqrs");
    put("8", "tuv");
    put("9", "wxyz");
  }};

  List<String> output = new ArrayList<String>();

  public void backtrack(String combination, String next_digits) {
    // if there is no more digits to check
    if (next_digits.length() == 0) {
      // the combination is done
      output.add(combination);
    }
    // if there are still digits to check
    else {
      // iterate over all letters which map the next available digit
      String digit = next_digits.substring(0, 1);
      String letters = phone.get(digit);
      for (int i = 0; i < letters.length(); i++) {
        String letter = phone.get(digit).substring(i, i + 1);
        // append the current letter to the combination and proceed to the next digits
        backtrack(combination + letter, next_digits.substring(1));
      }
    }
  }
  public List<String> letterCombinations(String digits) {
    if (digits.length() != 0)
      backtrack("", digits);
    return output;
  }
}
```

## 42. WORD SEARCH

Given an m x n board and a word, find if the 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:**

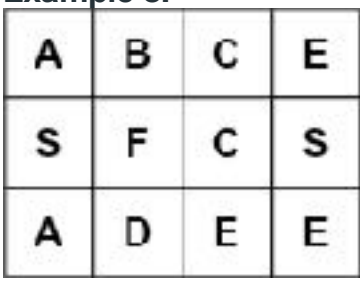



Input: board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "ABCCED"
Output: true

**Example 2:**


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

**Example 3:**




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

**Constraints:**

- m == board.length
- n = board[i].length
- 1 <= m, n <= 200
- 1 <= word.length <= 103
- board and word consists only of lowercase and uppercase English letters.

Here accepted solution based on recursion. To save memory I decuded to apply bit mask for every visited cell. Please check board[y][x] ^= 256;

```
public boolean exist(char[][] board, String word) {
    char[] w = word.toCharArray();
    for (int y=0; y<board.length; y++) {
        for (int x=0; x<board[y].length; x++) {
            if (exist(board, y, x, w, 0)) return true;
        }
    }
    return false;
}
```

```
private boolean exist(char[][] board, int y, int x, char[] word, int i) {
        if (i == word.length) return true;
        if (y<0 || x<0 || y == board.length || x == board[y].length) return false;
        if (board[y][x] != word[i]) return false;
        board[y][x] ^= 256;
        boolean exist = exist(board, y, x+1, word, i+1)
            || exist(board, y, x-1, word, i+1)
            || exist(board, y+1, x, word, i+1)
            || exist(board, y-1, x, word, i+1);
        board[y][x] ^= 256;
        return exist;
}
```

## 43. GENERATE PARENTHESIS

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: ["()"]

**Constraints:**

- 1 <= n <= 8

**SOLUTION**

My method is DP.
First consider how to get the result f(n) from previous result f(0)...f(n-1).
Actually, the result f(n) will be put an extra () pair to f(n-1).
Let the "(" always at the first position, to produce a valid result, we can only put ")" in a way that there will be i pairs () inside the extra () and n - 1 - i pairs () outside the extra pair.

Let us consider an example to get clear view:

f(0): ""

f(1): "("f(0)")"

f(2): "("f(0)")"f(1), "("f(1)")"

f(3): "("f(0)")"f(2), "("f(1)")"f(1), "("f(2)")"

So f(n) = "("f(0)")"f(n-1) , "("f(1)")"f(n-2) "("f(2)")"f(n-3) ... "("f(i)")"f(n-1-i) ... "(f(n-1)")"

```
public class Solution
{
    public List<String> generateParenthesis(int n)
    {
        List<List<String>> lists = new ArrayList<>();
        lists.add(Collections.singletonList(""));

        for (int i = 1; i <= n; ++i)
        {
            final List<String> list = new ArrayList<>();
            for (int j = 0; j < i; ++j)
            {
                for (final String first : lists.get(j))
                {
                    for (final String second : lists.get(i - 1 - j))
                    {
                        list.add("(" + first + ")" + second);
                    }
                }
            }

            lists.add(list);
        }

        return lists.get(lists.size() - 1);
    }
}
```