Coding Practice-7

1.Minimum Path Sum

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

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

Example 1:


```
Input: grid = [[1,3,1],[1,5,1],[4,2,1]]
Output: 7
Explanation: Because the path 1 → 3 → 1 → 1 → 1 minimizes the sum.
```

Code:

```
class Solution {
    public int minPathSum(int[][] grid) {
        int m = grid.length, n = grid[0].length;
        for (int j = 1; j < n; j++) {
            grid[0][j] += grid[0][j - 1];
        }
```

```
        for (int i = 1; i < m; i++) {
            grid[i][0] += grid[i - 1][0];
        }
        for (int i = 1; i < m; i++) {
            for (int j = 1; j < n; j++) {
                grid[i][j] += Math.min(grid[i - 1][j], grid[i][j - 1]);
            }
        }
        return grid[m - 1][n - 1];
    }
}
public class Main {
    public static void main(String[] args) {
        Solution solution = new Solution();
        int[][] grid = {
            {1, 3, 1},
            {1, 5, 1},
            {4, 2, 1}
        };
        int result = solution.minPathSum(grid);
        System.out.println("Minimum Path Sum: " + result);
    }
}
```

Output: Minimum Path Sum: 7

Time complexity: O(n*m)

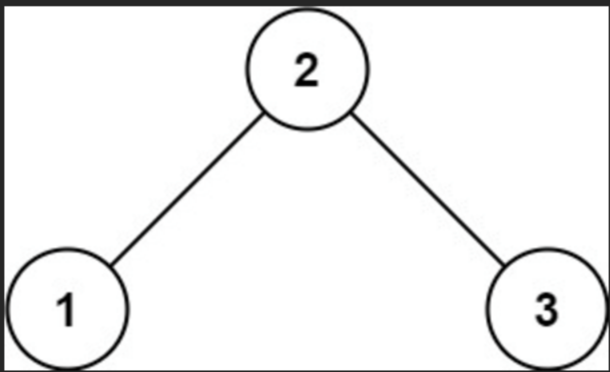
2. Validate binary search tree

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
```

Code:

```
class Solution {
    public boolean isValidBST(TreeNode root) {
        return validate(root, Long.MIN_VALUE, Long.MAX_VALUE);
    }
    private boolean validate(TreeNode node, long low, long high) {
        if (node == null) return true;
        if (node.val <= low || node.val >= high) return false;
        return validate(node.left, low, node.val) && validate(node.right, node.val, high);
    }
    public static void main(String[] args) {
        TreeNode root = new TreeNode(2, new TreeNode(1), new TreeNode(3));
        Solution sol = new Solution();
```

```
        System.out.println(sol.isValidBST(root));
    }
}
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;
    }
}
```

Output: true

Time Complexity:O(n)

3. Word Ladder

A **transformation sequence** from word `beginWord` to word `endWord` using a dictionary `wordList` is a sequence of words `beginWord -> s1 -> s2 -> ... -> sk` such that:

- Every adjacent pair of words differs by a single letter.
- Every `si` for `1 <= i <= k` is in `wordList`. Note that `beginWord` does not need to be in `wordList`.
- `sk == endWord`

Given two words, `beginWord` and `endWord`, and a dictionary `wordList`, return *the **number of words** in the **shortest transformation sequence** from* `beginWord` *to* `endWord`*, or* `0` *if no such sequence exists.*

Example 1:

> **Input:** beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log","cog"]
> **Output:** 5
> **Explanation:** One shortest transformation sequence is "hit" -> "hot" -> "dot" -> "dog" -> cog", which is 5 words long.

Code:

```
import java.util.*;
class Solution {
    public int ladderLength(String beginWord, String endWord, List<String> wordList) {
        Set<String> wordSet = new HashSet<>(wordList);
        if (!wordSet.contains(endWord)) return 0;
        Queue<String> queue = new LinkedList<>();
        queue.add(beginWord);
        int level = 1;
        while (!queue.isEmpty()) {
            int size = queue.size();
            for (int i = 0; i < size; i++) {
                String currentWord = queue.poll();
                char[] wordChars = currentWord.toCharArray();
                for (int j = 0; j < wordChars.length; j++) {
                    char originalChar = wordChars[j];
                    for (char c = 'a'; c <= 'z'; c++) {
                        if (c == originalChar) continue;
                        wordChars[j] = c;
                        String nextWord = new String(wordChars);
                        if (nextWord.equals(endWord)) return level + 1;
                        if (wordSet.contains(nextWord)) {
                            queue.add(nextWord);
                            wordSet.remove(nextWord);
                        }
                    }
                    wordChars[j] = originalChar;
                }
            }
            level++;
```

```
        }
        return 0;
    }
    public static void main(String[] args) {
        Solution solution = new Solution();
        String beginWord = "hit";
        String endWord = "cog";
        List<String> wordList = Arrays.asList("hot", "dot", "dog", "lot", "log", "cog");
        System.out.println(solution.ladderLength(beginWord, endWord, wordList));
    }
}
```

Output: 5

Time complexity: O(n×m^2)

4.Word Ladder II

A **transformation sequence** from word `beginWord` to word `endWord` using a dictionary `wordList` is a sequence of words `beginWord -> s1 -> s2 -> ... -> sk` such that:

- Every adjacent pair of words differs by a single letter.
- Every s_i for `1 <= i <= k` is in `wordList`. Note that `beginWord` does not need to be in `wordList`.
- s_k `== endWord`

Given two words, `beginWord` and `endWord`, and a dictionary `wordList`, return *all the* ***shortest transformation sequences*** *from* `beginWord` *to* `endWord`*, or an empty list if no such sequence exists. Each sequence should be returned as a list of the words* `[beginWord, s1, s2, ..., sk]`.

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"]]
Explanation: There are 2 shortest transformation sequences:
"hit" -> "hot" -> "dot" -> "dog" -> "cog"
"hit" -> "hot" -> "lot" -> "log" -> "cog"
```

Code:

```
import java.util.*;
class Solution {
    String b;
    HashMap<String, Integer> mpp;
    List<List<String>> ans;
    private void dfs(String word, List<String> seq) {
        if (word.equals(b)) {
            List<String> dup = new ArrayList<>(seq);
            Collections.reverse(dup);
            ans.add(dup);
            return;
        }
        int steps = mpp.get(word);
        int sz = word.length();
        for (int i = 0; i < sz; i++) {
            for (char ch = 'a'; ch <= 'z'; ch++) {
                char[] replacedCharArray = word.toCharArray();
                replacedCharArray[i] = ch;
                String replacedWord = new String(replacedCharArray);
                if (mpp.containsKey(replacedWord) && mpp.get(replacedWord) + 1 == steps) {
                    seq.add(replacedWord);
                    dfs(replacedWord, seq);
                    seq.remove(seq.size() - 1);
                }
            }
        }
    }

    public List<List<String>> findLadders(String beginWord, String endWord, List<String>
wordList) {
```

```java
Set<String> st = new HashSet<>();
for (String word : wordList) {
    st.add(word);
}
Queue<String> q = new LinkedList<>();
b = beginWord;
q.add(beginWord);
mpp = new HashMap<>();
mpp.put(beginWord, 1);
st.remove(beginWord);
while (!q.isEmpty()) {
    String word = q.poll();
    int steps = mpp.get(word);
    for (int i = 0; i < word.length(); i++) {
        for (char ch = 'a'; ch <= 'z'; ch++) {
            char[] replacedCharArray = word.toCharArray();
            replacedCharArray[i] = ch;
            String replacedWord = new String(replacedCharArray);
            if (st.contains(replacedWord)) {
                q.add(replacedWord);
                st.remove(replacedWord);
                mpp.put(replacedWord, steps + 1);
            }
        }
    }
}
ans = new ArrayList<>();
if (mpp.containsKey(endWord)) {
    List<String> seq = new ArrayList<>();
    seq.add(endWord);
```

```
            dfs(endWord, seq);
        }
        return ans;
    }
    public static void main(String[] args) {
        Solution solution = new Solution();
        String beginWord = "hit";
        String endWord = "cog";
        List<String> wordList = Arrays.asList("hot", "dot", "dog", "lot", "log", "cog");
        List<List<String>> ladders = solution.findLadders(beginWord, endWord, wordList);
        System.out.println("All shortest transformation sequences:");
        for (List<String> ladder : ladders) {
            System.out.println(ladder);
        }
    }
}
```

Output:

[hit, hot, dot, dog, cog]

[hit, hot, lot, log, cog]

Time complexity: O(n·m+p·m)

5.Course Schedule

There are a total of `numCourses` courses you have to take, labeled from `0` to `numCourses - 1`. You are given an array `prerequisites` where `prerequisites[i] = [`a_i`, `b_i`]` indicates that you **must** take course b_i first if you want to take course a_i.

- For example, the pair `[0, 1]`, indicates that to take course `0` you have to first take course `1`.

Return `true` if you can finish all courses. Otherwise, return `false`.

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.
```

Code:

```
class Solution {
    public boolean canFinish(int numCourses, int[][] prerequisites) {
        List<List<Integer>> graph = new ArrayList<>();
        int[] inDegree = new int[numCourses];
        for (int i = 0; i < numCourses; i++) {
            graph.add(new ArrayList<>());
        }
        for (int[] prereq : prerequisites) {
            graph.get(prereq[1]).add(prereq[0]);
            inDegree[prereq[0]]++;
        }
        Queue<Integer> queue = new LinkedList<>();
        for (int i = 0; i < numCourses; i++) {
            if (inDegree[i] == 0) {
```

```
                queue.add(i);
            }
        }
        int count = 0;
        while (!queue.isEmpty()) {
            int course = queue.poll();
            count++;
            for (int neighbor : graph.get(course)) {
                inDegree[neighbor]--;
                if (inDegree[neighbor] == 0) {
                    queue.add(neighbor);
                }
            }
        }
        return count == numCourses;
    }
    public static void main(String[] args) {
        Solution sol = new Solution();
        int numCourses = 4;
        int[][] prerequisites = {{1, 0}, {2, 1}, {3, 2}};
        System.out.println(sol.canFinish(numCourses, prerequisites)); // Output: true
    }
}
```

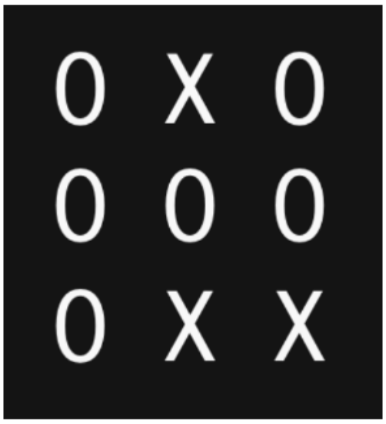
Output: true

Time complexity:O(n^2)

6. Design tic tac toe

A Tic-Tac-Toe board of size 3X3 is given after all the moves are played, i.e., all nine spots are filled. Find out if the given board is valid, i.e., is it possible to reach this board position after a set of moves or not.
Note that every arbitrarily filled grid of 9 spaces isn't valid, e.g., a grid filled with 3 **X** and 6 **O** isn't a valid situation because each player needs to take alternate turns.

Note: The game starts with X



Examples:

Input:
board[] = {'X', 'X', 'O',
'O', 'O', 'X',
'X', 'O', 'X'};
Output: Valid
Explanation: This is a valid board.

Input:
board[] = {'O', 'X', 'X',
'O', 'X', 'X',
'O', 'O', 'X'};
Output: Invalid
Explanation: Both X and O cannot win.

Code:

```
class Solution {
    int[][] win = {{0, 1, 2}, {3, 4, 5}, {6, 7, 8}, {0, 3, 6}, {1, 4, 7}, {2, 5, 8}, {0, 4, 8}, {2, 4, 6}}
```

```
public int isCWin(char[] board, char c) {
    int count = 0;
    for (int i = 0; i < 8; i++) {
        if (board[win[i][0]] == c && board[win[i][1]] == c && board[win[i][2]] == c) {
            count++;
        }
    }
    return count;
}
public boolean isValid(char[] board) {
    int xCount = 0, oCount = 0;
    for (char c : board) {
        if (c == 'X') {
            xCount++;
        } else if (c == 'O') {
            oCount++;
        }
    }
    int cx = isCWin(board, 'X');
    int co = isCWin(board, 'O');
    if (xCount != oCount + 1) {
        return false;
    }
    if (co == 1 && cx == 0) {
        return true;
    }
    if (cx == 1 && co == 0) {
        return true;
    }
    if (cx == 0 && co == 0) {
```

```
            return true;
        }
        return false;
    }
    public static void main(String[] args) {
        Solution solution = new Solution();
        char[] board = {'X', 'X', 'X', 'O', 'O', ' ', ' ', ' ', ' '};
        System.out.println(solution.isValid(board));
    }
}
```

Output: Invalid

Time Complexity: O(1)

7. Next permutation

A **permutation** of an array of integers is an arrangement of its members into a sequence or linear order.

- For example, for `arr = [1,2,3]`, the following are all the permutations of `arr`: `[1,2,3], [1,3,2], [2, 1, 3], [2, 3, 1], [3,1,2], [3,2,1]`.

The **next permutation** of an array of integers is the next lexicographically greater permutation of its integer. More formally, if all the permutations of the array are sorted in one container according to their lexicographical order, then the **next permutation** of that array is the permutation that follows it in the sorted container. If such arrangement is not possible, the array must be rearranged as the lowest possible order (i.e., sorted in ascending order).

- For example, the next permutation of `arr = [1,2,3]` is `[1,3,2]`.
- Similarly, the next permutation of `arr = [2,3,1]` is `[3,1,2]`.
- While the next permutation of `arr = [3,2,1]` is `[1,2,3]` because `[3,2,1]` does not have a lexicographical larger rearrangement.

Given an array of integers `nums`, *find the next permutation of* `nums`.

The replacement must be **in place** and use only constant extra memory.

Code:

```
class Solution {
```

```
public void nextPermutation(int[] nums) {
    int n = nums.length;
    int i = n - 2;
    while (i >= 0 && nums[i] >= nums[i + 1]) {
        i--;
    }
    if (i >= 0) {
        int j = n - 1;
        while (j >= 0 && nums[j] <= nums[i]) {
            j--;
        }
        swap(nums, i, j);
    }
    reverse(nums, i + 1, n - 1);
}
private void swap(int[] nums, int i, int j) {
    int temp = nums[i];
    nums[i] = nums[j];
    nums[j] = temp;
}
private void reverse(int[] nums, int start, int end) {
    while (start < end) {
        swap(nums, start, end);
        start++;
        end--;
    }
}
public static void main(String[] args) {
    Solution solution = new Solution();
    int[] nums = {1, 2, 3};
```

```
        solution.nextPermutation(nums);
        System.out.println(Arrays.toString(nums));
    }
}
```

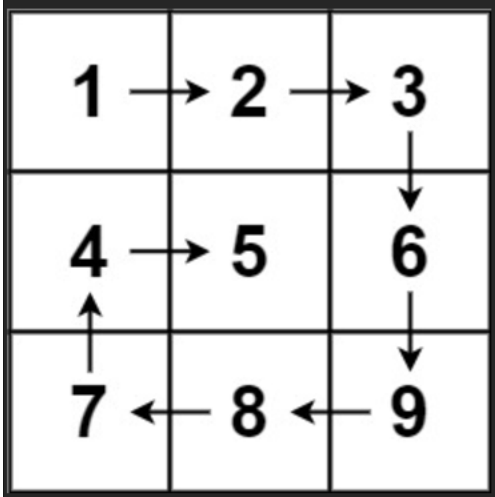
Output: [1, 3, 2]

Time Complexity:O(n)

8. Spiral matrix

Given an `m x n` `matrix`, return *all elements of the* `matrix` *in spiral order*.

Example 1:




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

Code:

```
import java.util.ArrayList;
import java.util.List;
class Solution {
    public List<Integer> spiralOrder(int[][] matrix) {
        List<Integer> result = new ArrayList<>();
        if (matrix == null || matrix.length == 0) return result;

        int m = matrix.length;
```

```
        int n = matrix[0].length;
        int top = 0, bot = m - 1, l = 0, r = n - 1;
        while (top <= bot && l <= r) {
            for (int i = l; i <= r; i++) {
                result.add(matrix[top][i]);
            }
            top++;
            for (int i = top; i <= bot; i++) {
                result.add(matrix[i][r]);
            }
            r--;
            if (top <= bot) {
                for (int i = r; i >= l; i--) {
                    result.add(matrix[bot][i]);
                }
                bot--;
            }
            if (l <= r) {
                for (int i = bot; i >= top; i--) {
                    result.add(matrix[i][l]);
                }
                l++;
            }
        }
        return result;
    }
    public static void main(String[] args) {
        Solution solution = new Solution();
        int[][] matrix = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
        List<Integer> result = solution.spiralOrder(matrix);
```

```
        System.out.println(result);
    }
}
```

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

Time Complexity:O(m*n)

9. 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.
```

Code:

```
import java.util.HashSet;
import java.util.Set;
class Solution {
    public int lengthOfLongestSubstring(String s) {
        Set<Character> set = new HashSet<>();
        int maxLength = 0;
        int left = 0;
        for (int right = 0; right < s.length(); right++) {
```

```
            if (!set.contains(s.charAt(right))) {
                set.add(s.charAt(right));
                maxLength = Math.max(maxLength, right - left + 1);
            } else {
                while (s.charAt(left) != s.charAt(right)) {
                    set.remove(s.charAt(left));
                    left++;
                }
                set.remove(s.charAt(left));
                left++;
                set.add(s.charAt(right));
            }
        }
        return maxLength;
    }
    public static void main(String[] args) {
        Solution solution = new Solution();
        String s = "abcabcbb";
        System.out.println(solution.lengthOfLongestSubstring(s));
    }
}
```

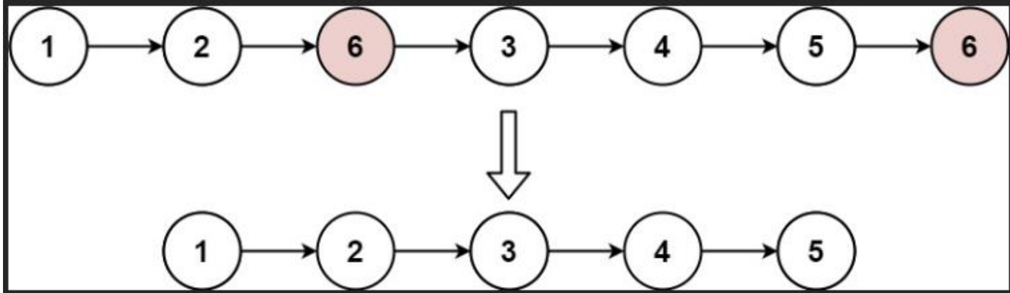
Output:3

Time Complexity:O(n)

10. Remove linked list elements

Given the `head` of a linked list and an integer `val`, remove all the nodes of the linked list that has `Node.val == val`, and return *the new head*.

Example 1:




```
Input: head = [1,2,6,3,4,5,6], val = 6
Output: [1,2,3,4,5]
```

Example 2:

```
Input: head = [], val = 1
Output: []
```

Code:

```
class Solution {
    public ListNode removeElements(ListNode head, int val) {
        ListNode temp = null;
        ListNode temp2 = head;

        while(temp2 != null){
            if(temp2.val == val){
                if(temp == null){
                    if(head.next == null){
                        return null;
                    }
                    head = head.next;
                    temp2 = head;
                }
                else if(temp2.next == null){
```

```
                temp.next = null;
                break;
            }
            else{
                temp2 = temp2.next;
                temp.next = temp2;
            }
        }
        else{
            temp = temp2;
            temp2 = temp.next;
        }
    }
    return head;
}
public static void main(String[] args) {
    Solution solution = new Solution();
    ListNode head = new ListNode(1);
    head.next = new ListNode(2);
    head.next.next = new ListNode(6);
    head.next.next.next = new ListNode(3);
    head.next.next.next.next = new ListNode(4);
    head.next.next.next.next.next = new ListNode(5);
    head.next.next.next.next.next.next = new ListNode(6);
    int val = 6;
    ListNode result = solution.removeElements(head, val);
    while (result != null) {
        System.out.print(result.val + " ");
        result = result.next;
    }
```

```
    }
}
class ListNode {
    int val;
    ListNode next;
    ListNode(int x) {
        val = x;
        next = null;
    }
}
```

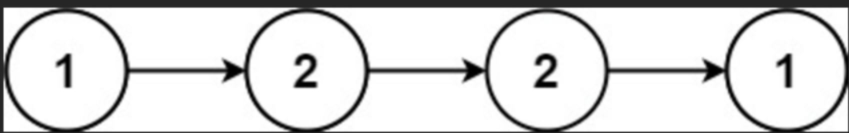
Output: [1, 2, 3, 4, 5]

Time Complexity:O(n)

11. Palindrome linked list

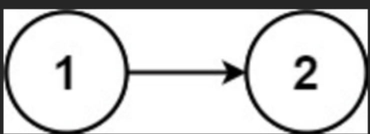
Given the `head` of a singly linked list, return `true` *if it is a palindrome or* `false` *otherwise.*

Example 1:



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

Example 2:



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

Code:

```
class Solution {
    public boolean isPalindrome(ListNode head) {
```

```
        List<Integer> list = new ArrayList<>();
        while (head != null) {
            list.add(head.val);
            head = head.next;
        }
        int l = 0;
        int r = list.size() - 1;
        while (l < r && list.get(l).equals(list.get(r))) {
            l++;
            r--;
        }
        return l >= r;
    }
    public static void main(String[] args) {
        Solution solution = new Solution();
        ListNode head = new ListNode(1);
        head.next = new ListNode(2);
        head.next.next = new ListNode(2);
        head.next.next.next = new ListNode(1);
        boolean result = solution.isPalindrome(head);
        System.out.println(result);
    }
}
class ListNode {
    int val;
    ListNode next;
    ListNode() {}
    ListNode(int val) { this.val = val; }
    ListNode(int val, ListNode next) { this.val = val; this.next = next; }
}
```

Output:true

Time Complexity:O(n)