Name:Sudheep S Dept:CSE Reg No:22CS171 Date:19.11.24

1.Next Permutation

class Solution {

    public void nextPermutation(int[] nums) {

        int ind1=-1;

        int ind2=-1;

        for(int i=nums.length-2;i>=0;i--){

            if(nums[i]<nums[i+1]){

                ind1=i;

                break;

            }

        }

        if(ind1==-1){

            reverse(nums,0);

        }

        else{

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

                if(nums[i]>nums[ind1]){

                    ind2=i;

                    break;

                }

            }

            swap(nums,ind1,ind2);

            reverse(nums,ind1+1);

        }

    }

    void swap(int[] nums,int i,int j){

        int temp=nums[i];

        nums[i]=nums[j];

        nums[j]=temp;

    }

    void reverse(int[] nums,int start){

        int i=start;

        int j=nums.length-1;

        while(i<j){

            swap(nums,i,j);

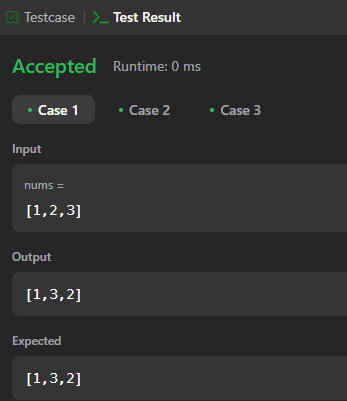
            i++;

            j--;

        }

    }

}



2.Spiral Matrix

class Solution {

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

        int rows = matrix.length;

        int cols = matrix[0].length;

        int x = 0;

        int y = 0;

        int dx = 1;

        int dy = 0;

        List<Integer> res = new ArrayList<>();

        for (int i = 0; i < rows \* cols; i++) {

            res.add(matrix[y][x]);

            matrix[y][x] = -101;

            if (!(0 <= x + dx && x + dx < cols && 0 <= y + dy && y + dy < rows) || matrix[y+dy][x+dx] == -101) {

                int temp = dx;

                dx = -dy;

                dy = temp;

            }

            x += dx;

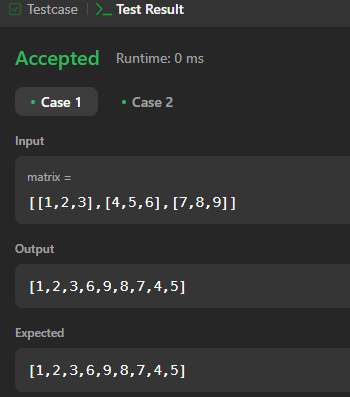
            y += dy;

        }

        return res;

    }

}



3.Longest Substring without repeating character

class Solution {

    public int lengthOfLongestSubstring(String s) {

        int left = 0;

        int maxLength = 0;

        HashSet<Character> charSet = new HashSet<>();

        for (int right = 0; right < s.length(); right++) {

            while (charSet.contains(s.charAt(right))) {

                charSet.remove(s.charAt(left));

                left++;

            }

            charSet.add(s.charAt(right));

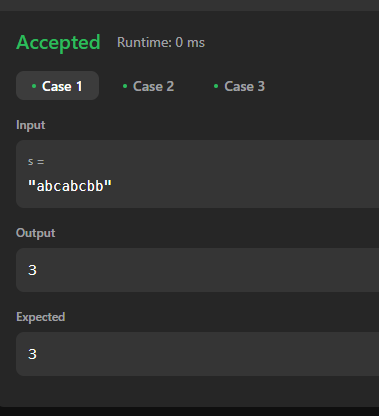
            maxLength = Math.max(maxLength, right - left + 1);

        }

        return maxLength;

    }

}



4.Remove Linked List element

class Solution {

    public ListNode removeElements(ListNode head, int val) {

        ListNode ans = new ListNode(0, head);

        ListNode dummy = ans;

        while (dummy != null) {

            while (dummy.next != null && dummy.next.val == val) {

                dummy.next = dummy.next.next;

            }

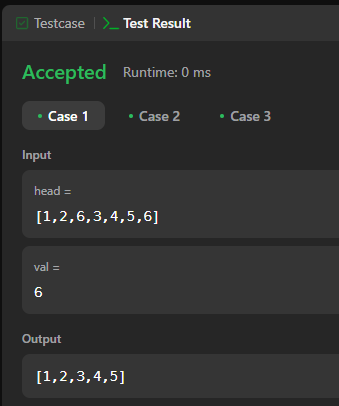
            dummy = dummy.next;

        }

        return ans.next;

    }

}



5.Palindrome Linked List

class Solution {

    public boolean isPalindrome(ListNode head) {

        List<Integer> list = new ArrayList();

        while(head != null) {

            list.add(head.val);

            head = head.next;

        }

        int left = 0;

        int right = list.size()-1;

        while(left < right && list.get(left) == list.get(right)) {

            left++;

            right--;

        }

        return left >= right;

    }

}



6.Minimum path sum

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]);

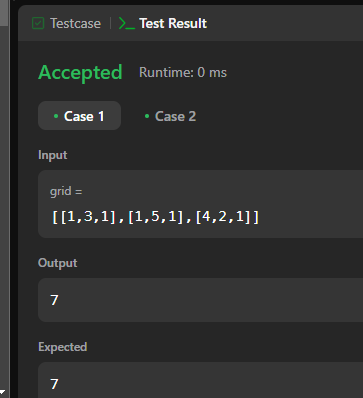
            }

        }

        return grid[m - 1][n - 1];

    }

}



7.Validate Binary Search Tree

/\*\*

 \* 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 {

    Stack<Integer> stack = new Stack();

    public boolean isValidBST(TreeNode root) {

        fillStack(root);

        while(!stack.isEmpty()){

            int ele = stack.pop();

            if(stack.isEmpty()){

                return true;

            }

            if(ele<=stack.peek()){

                return false;

            }

        }

        return true;

    }

    public void fillStack(TreeNode root){

        if(root !=null){

           fillStack(root.left);

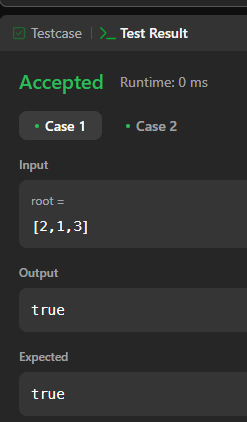
           stack.push(root.val);

           fillStack(root.right);

        }

    }

}



8.Word Ladder

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.offer(beginWord);

        Set<String> visited = new HashSet<>();

        visited.add(beginWord);

        int length = 1;

        while (!queue.isEmpty()) {

            int levelSize = queue.size();

            for (int i = 0; i < levelSize; i++) {

                String currentWord = queue.poll();

                if (currentWord.equals(endWord)) return length;

                for (String neighbor : getNeighbors(currentWord, wordSet)) {

                    if (!visited.contains(neighbor)) {

                        visited.add(neighbor);

                        queue.offer(neighbor);

                    }

                }

            }

            length++;

        }

        return 0;

    }

    private List<String> getNeighbors(String word, Set<String> wordSet) {

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

        char[] wordChars = word.toCharArray();

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

            char originalChar = wordChars[i];

            for (char c = 'a'; c <= 'z'; c++) {

                if (c == originalChar) continue;

                wordChars[i] = c;

                String transformedWord = new String(wordChars);

                if (wordSet.contains(transformedWord)) {

                    neighbors.add(transformedWord);

                }

            }

            wordChars[i] = originalChar;

        }

        return neighbors;

    }

}

