Vasanthan_T_DSA_Practice-7

19/11/2024

1. **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.

Example 1:

Input: nums = [1,2,3]

Output: [1,3,2]

Code:

```
class Solution {
    public void nextPermutation(int[] nums) {
        int n=nums.length;
        int ind1=-1;
```

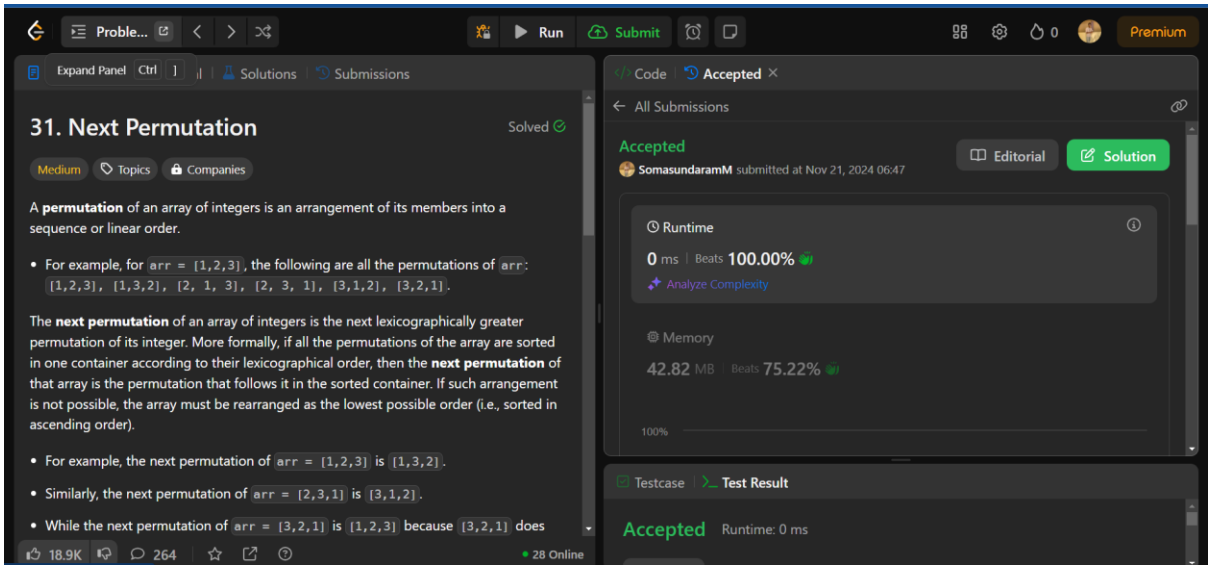
```
    int ind2=-1;
   for(int i=n-2;i>=0;i--){
       if(nums[i]<nums[i+1]){
           ind1=i;
           break;
       }
   }
   if(ind1==-1){
    reverseFunction(nums,0);
   }
   else{
    for(int i=n-1;i>ind1;i--){
       if(nums[i]>nums[ind1]){
           ind2=i;
           break;
       }
    }
    swap(nums,ind1,ind2);
    reverseFunction(nums,ind1+1);
   }
}
public void reverseFunction(int[] nums,int l){
    int i=l;
    int j=nums.length-1;
    while(i<=j){
       swap(nums,i,j);
       i++;
       j--;
```

```
        }

    }
    public void swap(int[] nums,int l,int r){

        int temp=nums[l];
        nums[l]=nums[r];
        nums[r]=temp;
    }
}
```

Output:



Time Complexity: O(n)

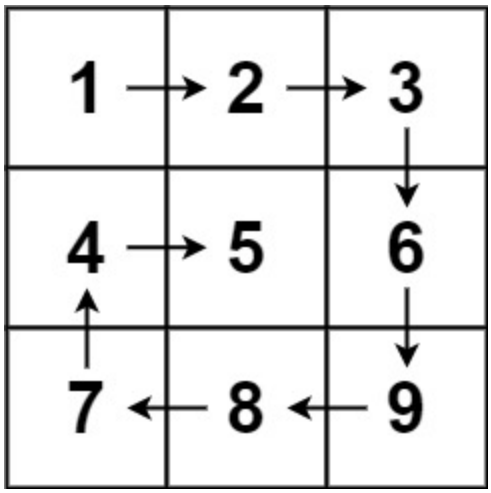
Space Complexity: O(1)

2.Spiral Matrix

1. 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]

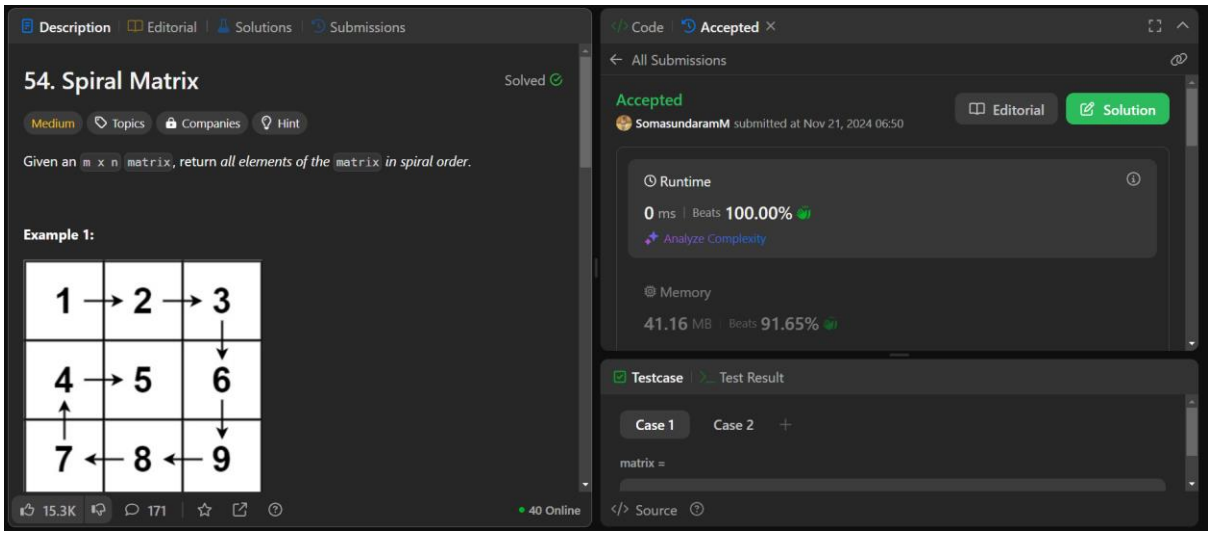
Code:

```
class Solution {
    public List<Integer> spiralOrder(int[][] matrix) {
        List<Integer> ans = new ArrayList<>();
        int l = 0, r = matrix[0].length - 1, t = 0, b = matrix.length - 1;
        while (t <= b && l <= r) {
            for (int i = l; i <= r; i++) {
                ans.add(matrix[t][i]);
            }
            t++;
            for (int i = t; i <= b; i++) {
                ans.add(matrix[i][r]);
            }
```

```
            r--;
            if (t <= b) {
                for (int i = r; i >= l; i--) {
                    ans.add(matrix[b][i]);
                }
                b--;
            }
            if (l<=r) {
                for (int i = b; i >= t; i--) {
                    ans.add(matrix[i][l]);
                }
                l++;
            }
        }
        return ans;

    }
}
```

Output:



Time Complexity: O(m*n)

Space Complexity: O(n)

3. Longest Substring without Repeating Character

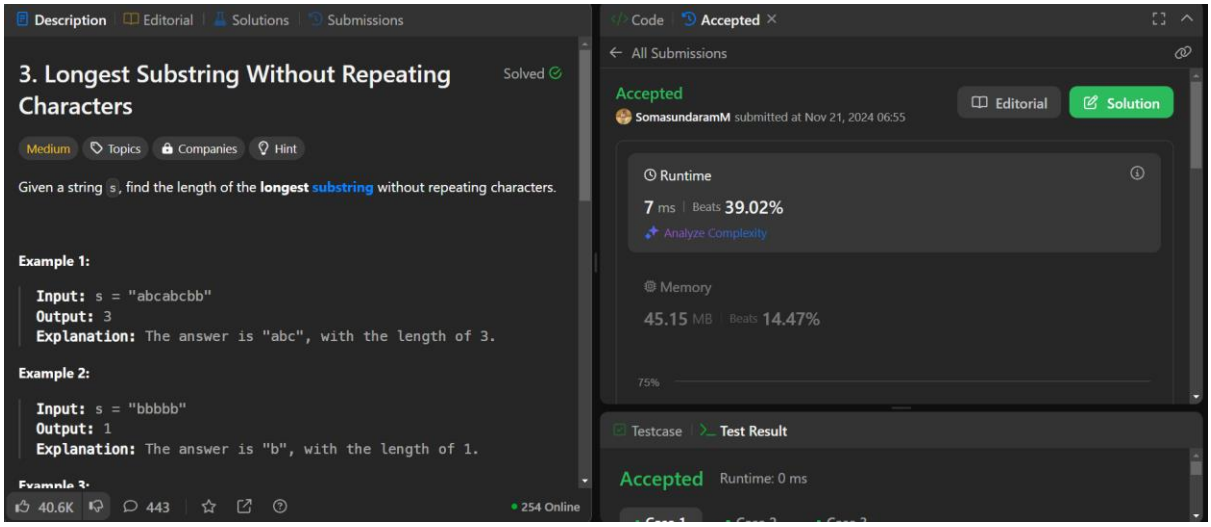
Given a string s, find the length of the **longest substring** without repeating characters.

Example 1:
Input: s = "abcabcbb"
Output: 3
Explanation: The answer is "abc", with the length of 3.

Solution:

```
class Solution {
   public int lengthOfLongestSubstring(String s) {
      int n=s.length();
      if(n==0 || n==1){
         return n;
      }
      int max=-1;
      boolean[] visited=new boolean[256];
      int left=0,right=0;
      while(right<n){
         while(visited[s.charAt(right)]){
            visited[s.charAt(left)]=false;
            left++;
         }
         visited[s.charAt(right)]=true;
         max=Math.max(max,right-left+1);
         right++;
      }
      return max;
   }
}
```

Output:



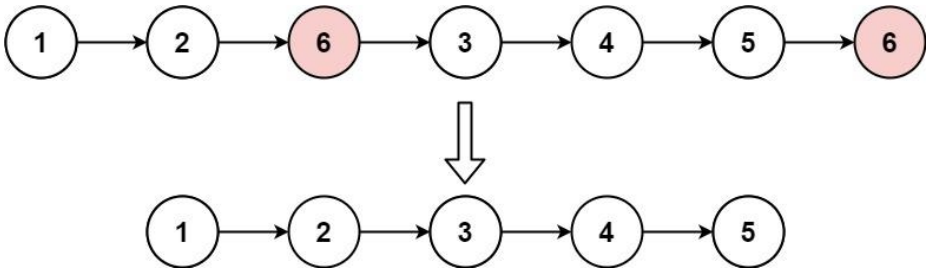
Time Complexity: O(n)

Space ComplexityO(1)

2. 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:



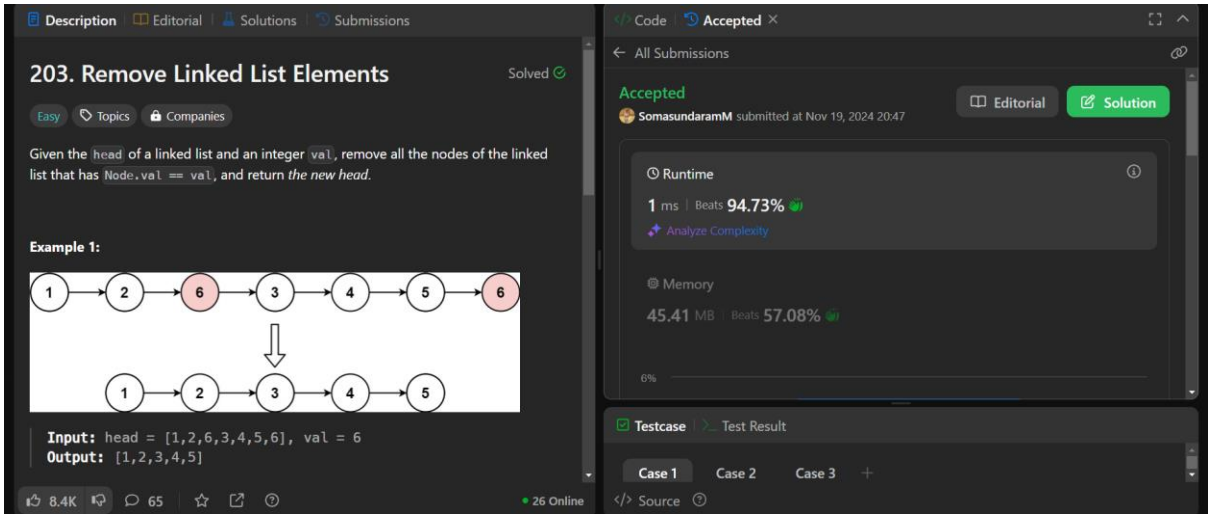
Input: head = [1,2,6,3,4,5,6], val = 6
Output: [1,2,3,4,5]

Solution:

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

```
 * }
 */
class Solution {
    public ListNode removeElements(ListNode head, int val) {
        if(head==null) return head;
        ListNode dummy=new ListNode(0);
        dummy.next=head;
        ListNode current=dummy;
        while(current.next!=null){
            if(current.next.val==val){
                current.next=current.next.next;
            }
            else current=current.next;
        }
        return dummy.next;
    }
```

Output:



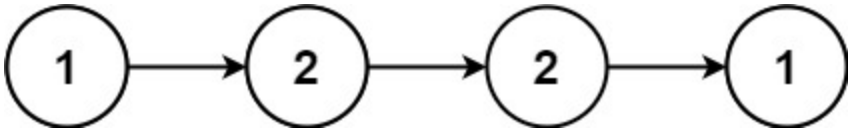
Time Complexity:O(n)

Space Complexity:O(1)

3. **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

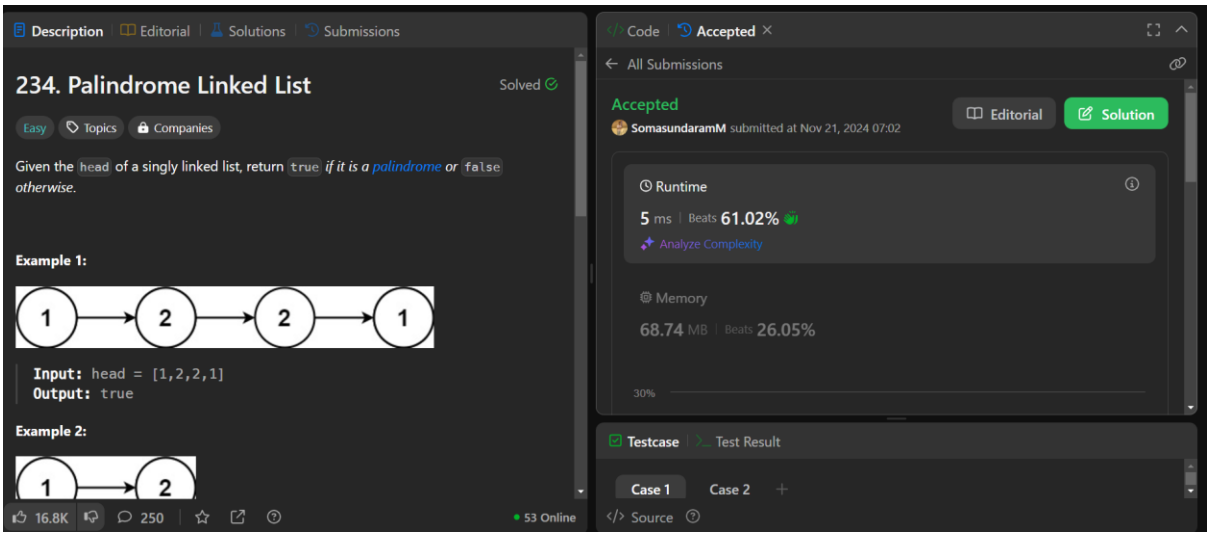
Code:

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution {
    public boolean isPalindrome(ListNode head) {
        ListNode slow=head,fast=head;
        while(fast.next!=null && fast.next.next!=null){
            slow=slow.next;
            fast=fast.next.next;
        }
        ListNode head2=reverse(slow.next);
        slow.next=null;
        while(head!=null && head2!=null){
            if(head.val!=head2.val){
                return false;
            }
            head=head.next;
            head2=head2.next;
        }
        return true;

    }
    static ListNode reverse(ListNode head){
        ListNode prev,next,curr;
        prev=null;
        curr=head;
```

```
        while(curr!=null){
            next=curr.next;
            curr.next=prev;
            prev=curr;
            curr=next;
        }
        return prev;
    }
}
```

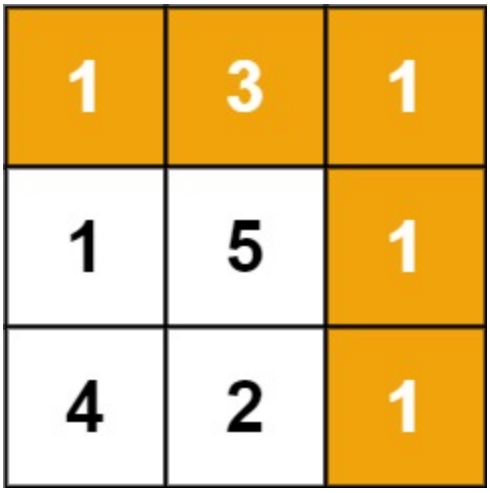
Output:



Time Complexity:O(n)
Space Complexity: O(1)

4. 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:

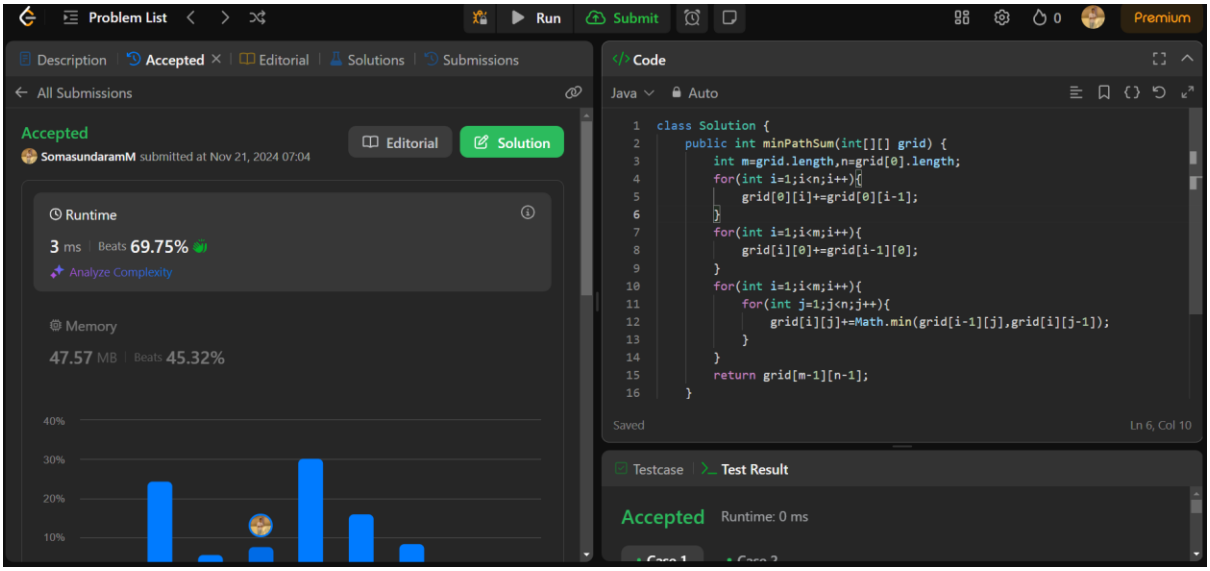


Input: grid = [[1,3,1],[1,5,1],[4,2,1]]
Output: 7
Explanation: Because the path 1 → 3 → 1 → 1 → 1 minimizes the sum.

Solution:

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

Output:



Time Complexity: O(n)

Space Complexity: O(1)