

# DSA CODING PRACTICE – 8

## 1.Spiral matrix

The screenshot displays a coding interface for the 'Spiral Matrix' problem. On the left, the 'Accepted' status is shown with a submission by 'Jency R' at Nov 19, 2024 13:47. The runtime is 0 ms, beating 100.00% of submissions, and the memory usage is 41.38 MB, beating 65.27%. The code on the right is a Java solution that traverses the matrix in a spiral order, starting from the top-left and moving right, then down, then left, and finally up, repeating the cycle.

```
1 class Solution {
2     public List<Integer> spiralOrder(int[][] matrix) {
3         List<Integer> res = new ArrayList<>();
4         if(matrix == null || matrix.length == 0){
5             return res;
6         }
7         int l = 0, t = 0;
8         int r = matrix[0].length - 1, b = matrix.length - 1;
9
10        while(t <= b && l <= r){
11            // Traverse from left to right
12            for(int i = l; i <= r; i++){
13                res.add(matrix[t][i]);
14            }
15            t++;
16
17            // Traverse from top to bottom
18            for(int i = t; i <= b; i++){
19                res.add(matrix[i][r]);
20            }
21        }
22    }
23 }
```

## 2. Longest substring without repeating characters

The screenshot displays a coding interface for the 'Longest Substring Without Repeating Characters' problem. On the left, the 'Accepted' status is shown with a submission by 'Jency R' at Nov 19, 2024 18:07. The runtime is 5 ms, beating 87.24% of submissions, and the memory usage is 44.46 MB, beating 68.04%. The code on the right is a Java solution that uses a sliding window approach with a HashMap to track the indices of characters in the current substring. It iterates through the string, updating the start and end of the window to maintain a substring with all unique characters.

```
1 class Solution {
2     public static int lengthOfLongestSubstring(String s) {
3         Map<Character, Integer> charIndexMap = new HashMap<>();
4         int start = 0;
5         int maxLength = 0;
6
7         for (int end = 0; end < s.length(); end++) {
8             char currentChar = s.charAt(end);
9
10            if (charIndexMap.containsKey(currentChar) && charIndexMap.get(currentChar) >= start) {
11                start = charIndexMap.get(currentChar) + 1;
12            }
13
14            charIndexMap.put(currentChar, end);
15            int currentLength = end - start + 1;
16            maxLength = Math.max(maxLength, currentLength);
17        }
18        return maxLength;
19    }
20 }
```

### 3. Remove linked list elements

The screenshot shows a LeetCode submission for the problem "Remove Elements". The submission is accepted and was made by user "Jency R" on Nov 19, 2024, at 18:27. The performance metrics are 1 ms runtime, beating 94.73% of solutions, and 45.73 MB memory, beating 22.34% of solutions. The code is a Java solution that removes elements from a linked list.

```
10 //
11 class Solution {
12     public ListNode removeElements(ListNode head, int val) {
13         ListNode ans = new ListNode(0, head);
14         ListNode d = ans;
15         while(d != null) {
16             while(d.next != null && d.next.val == val) {
17                 d.next = d.next.next;
18             }
19             d = d.next;
20         }
21         return ans.next;
22     }
23 }
24
25 }
```

### 4. Palindrome linked list

The screenshot shows a LeetCode submission for the problem "Palindrome Linked List". The submission is accepted and was made by user "Jency R" on Nov 19, 2024, at 18:12. The performance metrics are 8 ms runtime, beating 31.06% of solutions, and 56.24 MB memory, beating 91.46% of solutions. The code is a Java solution that checks if a linked list is a palindrome.

```
4 class Solution {
5     public boolean isPalindrome(ListNode head) {
6         List<Integer> vals = new ArrayList<>();
7         ListNode currentNode = head;
8         while (currentNode != null) {
9             vals.add(currentNode.val);
10            currentNode = currentNode.next;
11        }
12        int start = 0;
13        int end = vals.size() - 1;
14        while (start < end) {
15            if (!vals.get(start).equals(vals.get(end))) {
16                return false;
17            }
18            start++;
19            end--;
20        }
21        return true;
22    }
23 }
```

## 5. Validate binary search tree

The screenshot displays a LeetCode submission interface for the problem "Validate Binary Search Tree". The submission is marked as "Accepted" and was submitted at Nov. The performance metrics show a runtime of 0 ms, beating 100.00% of submissions, and a memory usage of 44.31 MB, beating 21.99% of submissions. The code is written in Java and implements a recursive helper method to validate the BST.

**Runtime:** 0 ms | Beats 100.00%  
[Analyze Complexity](#)

**Memory:** 44.31 MB | Beats 21.99%

```
class Solution {
    public boolean isValidBST(TreeNode root) {
        return isValidBSTHelper(root, null, null);
    }

    private boolean isValidBSTHelper(TreeNode node, Integer min, Integer max) {
        if (node == null) {
            return true;
        }
        if ((min != null && node.val <= min) || (max != null && node.val >= max)) {
            return false;
        }
        return isValidBSTHelper(node.left, min, node.val) && isValidBSTHelper(node.right, node.val, max);
    }
}
```

Saved | Ln 21, Col 80

Test Result | Testcase