Question No. 20 :-
https://leetcode.com/problems/valid-parentheses/description/
Solution Link :-
https://leetcode.com/problems/valid-parentheses/submissions/1399707177/

Description :-

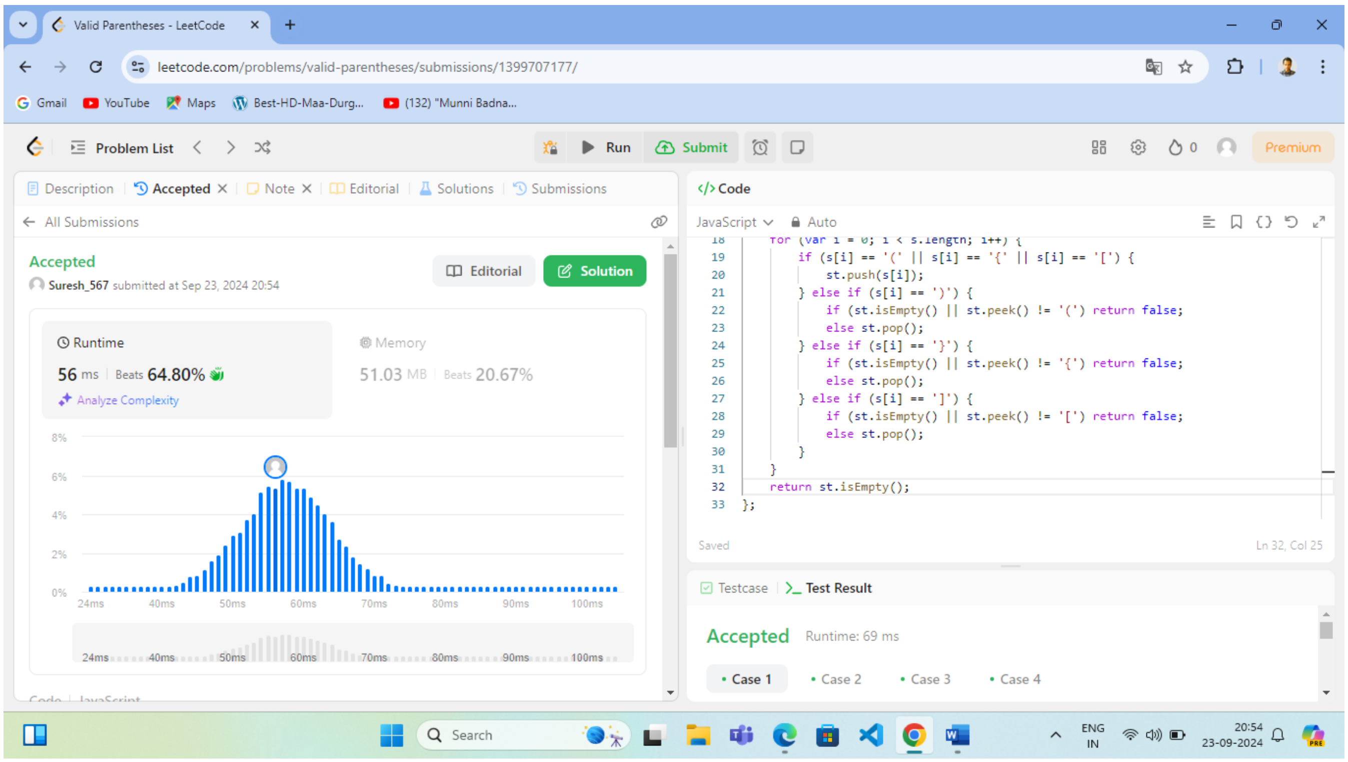
Time Complexity :- O(n)

This implementation also iterates over the string s once. The push, pop, peek, and isEmpty methods all operate in constant time, O(1). Therefore, the time complexity remains **O(n)**, where n is the length of the string s.

Space Complexity :- O(n)

The worst-case scenario is when the stack holds all the opening brackets, so the space complexity is **O(n)**.

Screenshot :-



Accepted
Suresh_567 submitted at Sep 23, 2024 20:54

Runtime 56 ms | Beats 64.80%
Memory 51.03 MB | Beats 20.67%

```
    for (var i = 0; i < s.length; i++) {
        if (s[i] == '(' || s[i] == '{' || s[i] == '[') {
            st.push(s[i]);
        } else if (s[i] == ')') {
            if (st.isEmpty() || st.peek() != '(') return false;
            else st.pop();
        } else if (s[i] == '}') {
            if (st.isEmpty() || st.peek() != '{') return false;
            else st.pop();
        } else if (s[i] == ']') {
            if (st.isEmpty() || st.peek() != '[') return false;
            else st.pop();
        }
    }
    return st.isEmpty();
};
```

Accepted Runtime: 69 ms

Question No. 496 :-

https://leetcode.com/problems/next-greater-element-i/description/

Solution Link :-

https://leetcode.com/problems/next-greater-element-i/submissions/1399718722/

Description :-

Time Complexity :- O(n+m)

The outer for loop iterates over all elements of nums2, making it **O(n)**, where n is the length of nums2.

The inner while loop can only run once for each element (each element is pushed and popped once from the stack). Thus, the while loop in total will also contribute **O(n)**.

The second loop iterates over nums1 and does constant-time lookups in the Map, making it **O(m)**, where m is the length of nums1.

Overall, the time complexity is **O(n + m)**, where n is the size of nums2 and m is the size of nums1.

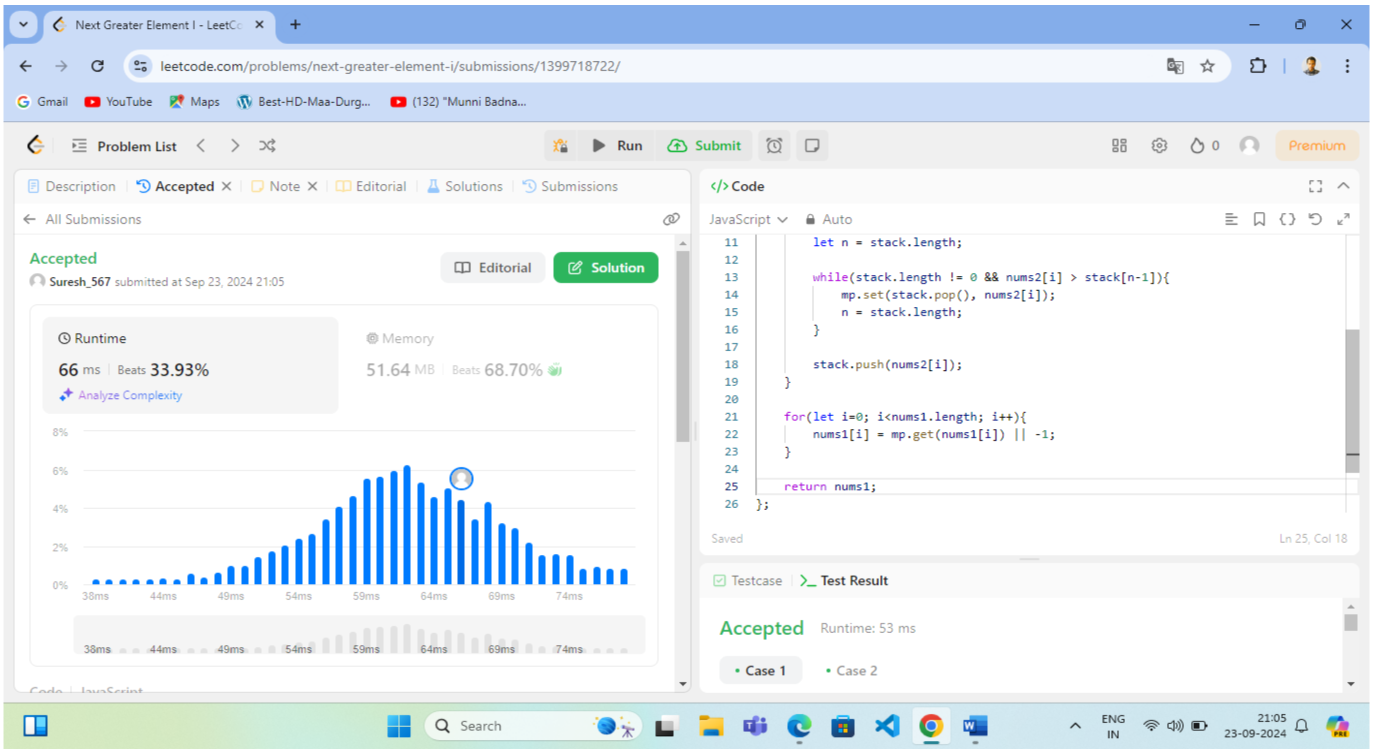
Space Complexity :- O(n)

The space complexity of the stack is **O(n)** in the worst case, as all elements of nums2 could be pushed onto the stack.

The Map stores a mapping for each element of nums2, which also takes **O(n)** space.

Therefore, the space complexity is **O(n)**.

Screenshot :-


Accepted

Suresh_567 submitted at Sep 23, 2024 21:05

Runtime 66 ms | Beats 33.93%

Memory 51.64 MB | Beats 68.70%

```
        let n = stack.length;

        while(stack.length != 0 && nums2[i] > stack[n-1]){
            mp.set(stack.pop(), nums2[i]);
            n = stack.length;
        }

        stack.push(nums2[i]);
    }

    for(let i=0; i<nums1.length; i++){
        nums1[i] = mp.get(nums1[i]) || -1;
    }

    return nums1;
};
```

Accepted Runtime: 53 ms

Question No. 1047 :-

https://leetcode.com/problems/remove-all-adjacent-duplicates-in-string/description/

Solution Link :-

https://leetcode.com/problems/remove-all-adjacent-duplicates-in-string/submissions/1399732919/

Description :-

Time Complexity :- O(n)

The loop iterates over each character of the string once, so the time complexity for the loop is **O(n)**, where n is the length of the string.

Stack operations like push and pop are done in constant time **O(1)**.

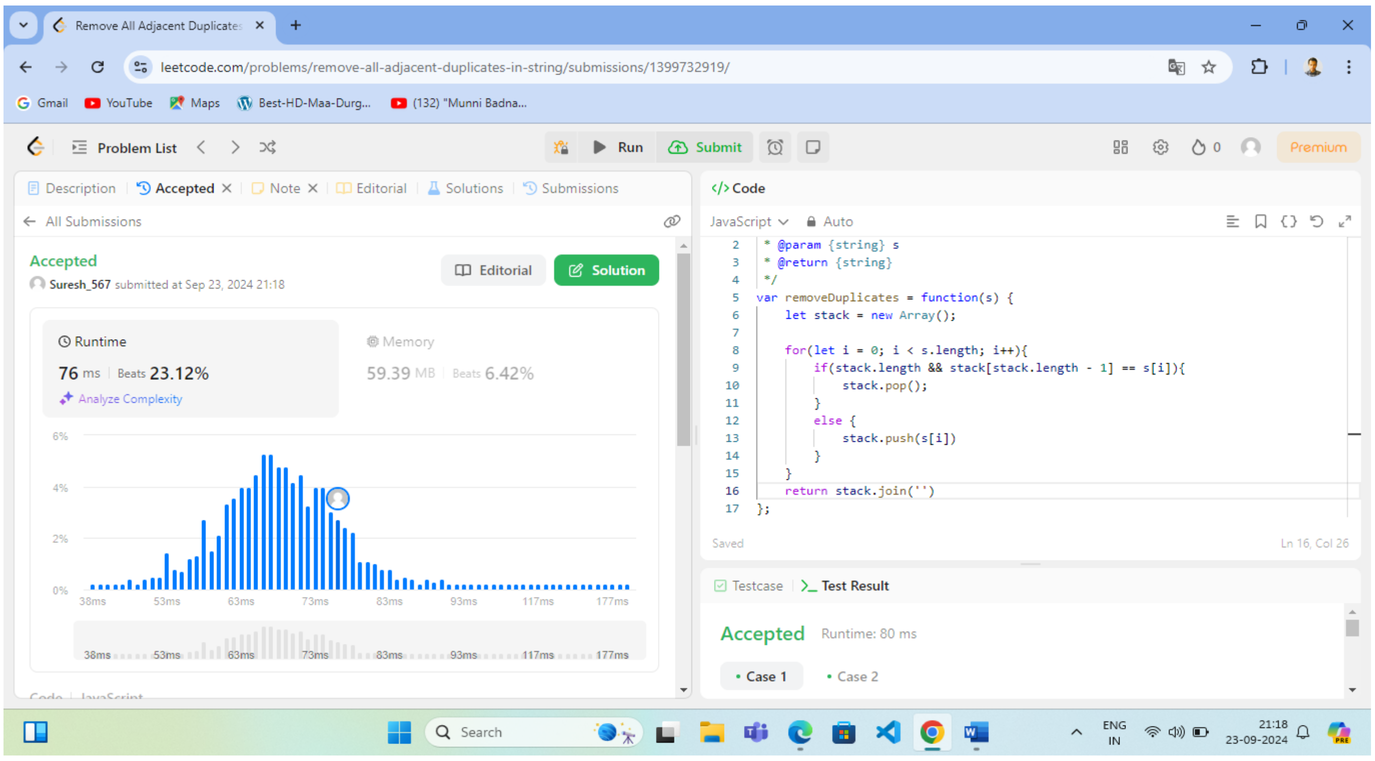
The join operation takes **O(n)** to concatenate the string.

Therefore, the total time complexity is **O(n)**.

Space Complexity :- O(n)

The space complexity is **O(n)** in the worst case, where no duplicate characters are found, and all characters are stored in the stack.

Screenshot :-


leetcode.com/problems/remove-all-adjacent-duplicates-in-string/submissions/1399732919/

Accepted
Suresh_567 submitted at Sep 23, 2024 21:18

Runtime: 76 ms | Beats 23.12%
Memory: 59.39 MB | Beats 6.42%

```javascript
 * @param {string} s
 * @return {string}
 */
var removeDuplicates = function(s) {
    let stack = new Array();

    for(let i = 0; i < s.length; i++){
        if(stack.length && stack[stack.length - 1] == s[i]){
            stack.pop();
        }
        else {
            stack.push(s[i])
        }
    }
    return stack.join('')
};
```

Accepted Runtime: 80 ms

Question No. 42 :-

https://leetcode.com/problems/trapping-rain-water/description/

Solution Link :-

https://leetcode.com/problems/trapping-rain-water/submissions/1399744726/

Description :-

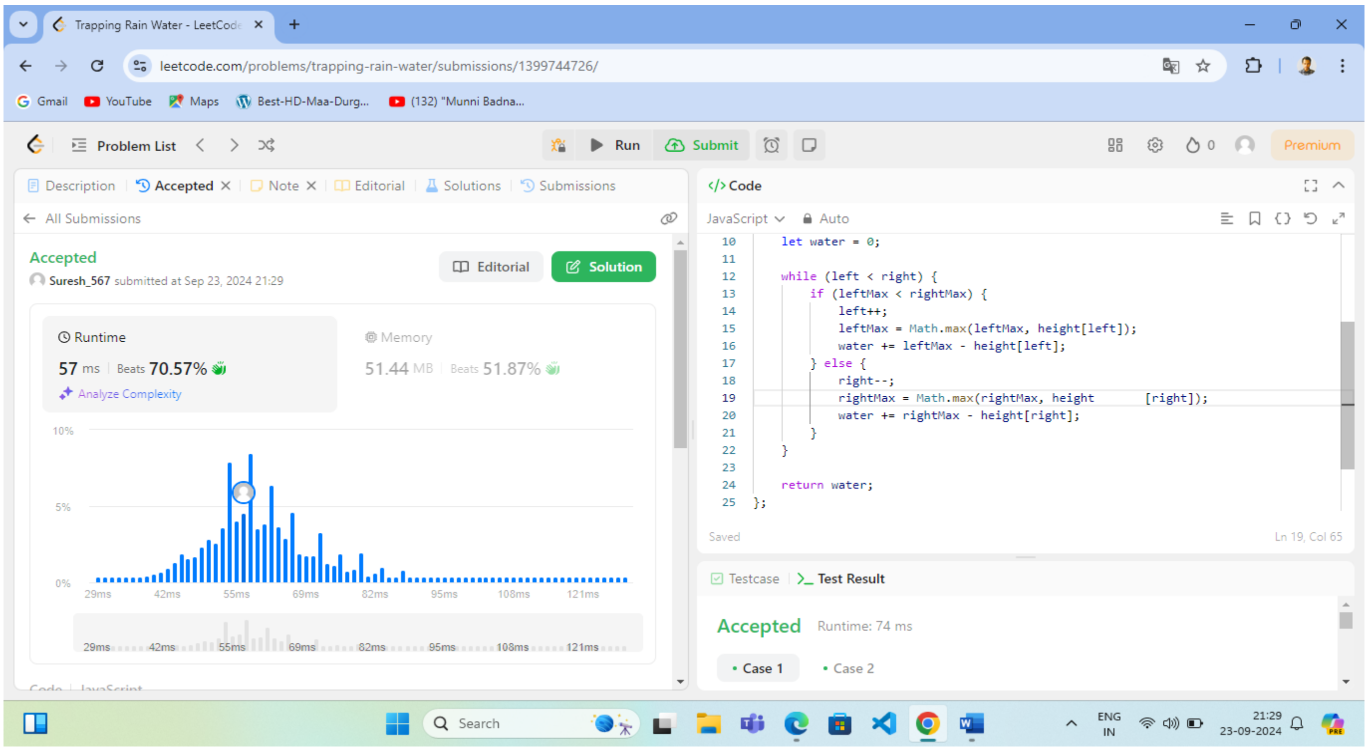
Time Complexity :- O(n)

This approach iterates over the array exactly once, moving pointers from both ends of the array toward the center. Each step involves constant time operations, making the time complexity **O(n)**, where n is the length of the height array.

Space Complexity :- O(1)

This approach uses a constant amount of extra space for the left, right, leftMax, rightMax, and water variables. Therefore, the space complexity is **O(1)**.

Screenshot :-


Accepted

Suresh_567 submitted at Sep 23, 2024 21:29

Runtime 57 ms | Beats 70.57%

Memory 51.44 MB | Beats 51.87%

```
    let water = 0;

    while (left < right) {
        if (leftMax < rightMax) {
            left++;
            leftMax = Math.max(leftMax, height[left]);
            water += leftMax - height[left];
        } else {
            right--;
            rightMax = Math.max(rightMax, height        [right]);
            water += rightMax - height[right];
        }
    }

    return water;
};
```

Accepted Runtime: 74 ms

Question No. 84:-
https://leetcode.com/problems/largest-rectangle-in-histogram/description/

Solution Link :-
https://leetcode.com/problems/largest-rectangle-in-histogram/submissions/1399755246/

Description :-

Time Complexity :- O(n)

The algorithm iterates over the heights array once in the main while loop, and during this iteration, each element is pushed onto the stack once and popped from the stack once. This gives an **O(n)** time complexity for the loop, where n is the length of the heights array.

The second while loop processes any remaining bars in the stack. Since each element can only be pushed and popped once, the overall contribution of this part is also **O(n)**.

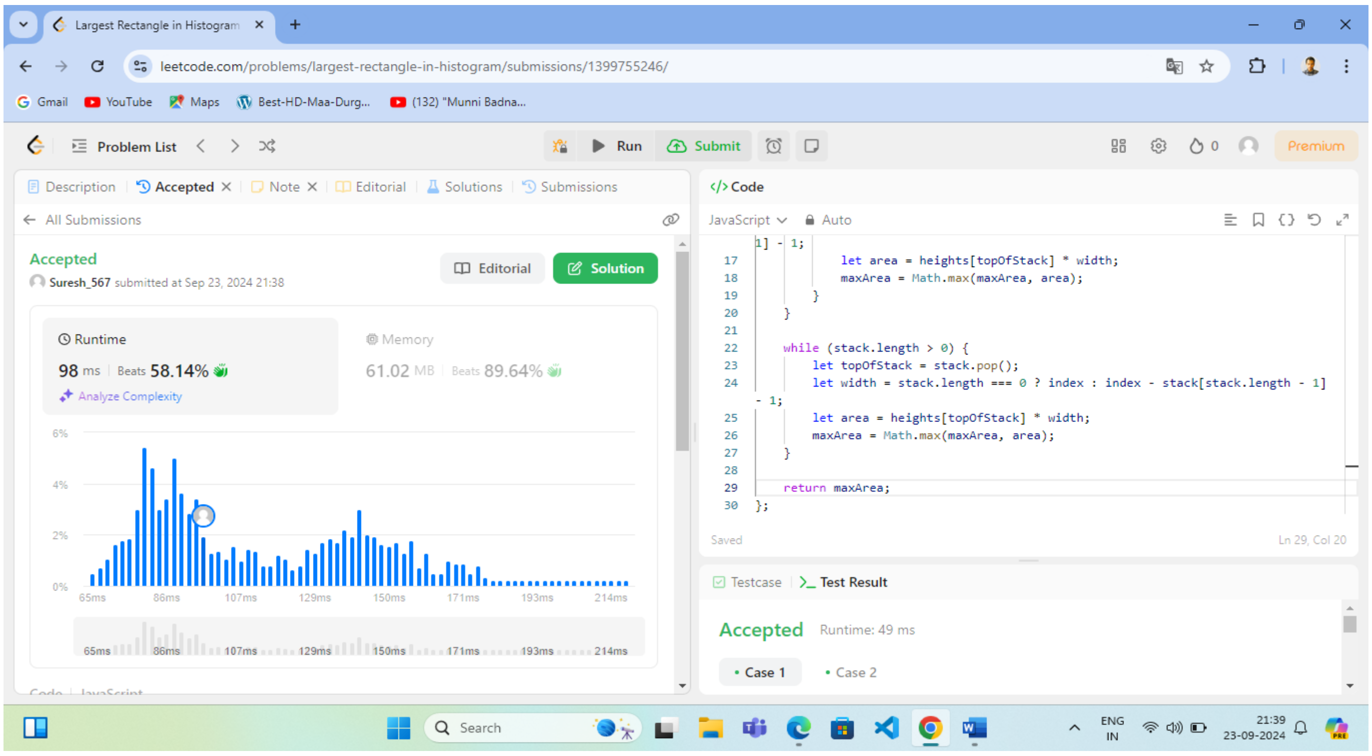
So, the overall time complexity is **O(n)**.

Space Complexity :- O(1)

The space complexity is determined by the space used by the stack. In the worst case, if the heights are increasing, every element in the heights array will be pushed onto the stack, leading to **O(n)** space usage.

So, the overall space complexity is **O(n)**.

Screenshot :-



Largest Rectangle in Histogram

leetcode.com/problems/largest-rectangle-in-histogram/submissions/1399755246/

Accepted

Suresh_567 submitted at Sep 23, 2024 21:38

Runtime

98 ms | Beats 58.14%

Memory

61.02 MB | Beats 89.64%

```
1] - 1;
            let area = heights[topOfStack] * width;
            maxArea = Math.max(maxArea, area);
        }
    }

    while (stack.length > 0) {
        let topOfStack = stack.pop();
        let width = stack.length === 0 ? index : index - stack[stack.length - 1]
- 1;
        let area = heights[topOfStack] * width;
        maxArea = Math.max(maxArea, area);
    }

    return maxArea;
};
```

Accepted Runtime: 49 ms

Case 1 · Case 2