Data Structure Algorithm Assignment-3

Question: https://leetcode.com/problems/climbing-stairs/description/

Answer: https://leetcode.com/problems/climbing-stairs/submissions/1628585122

Description:

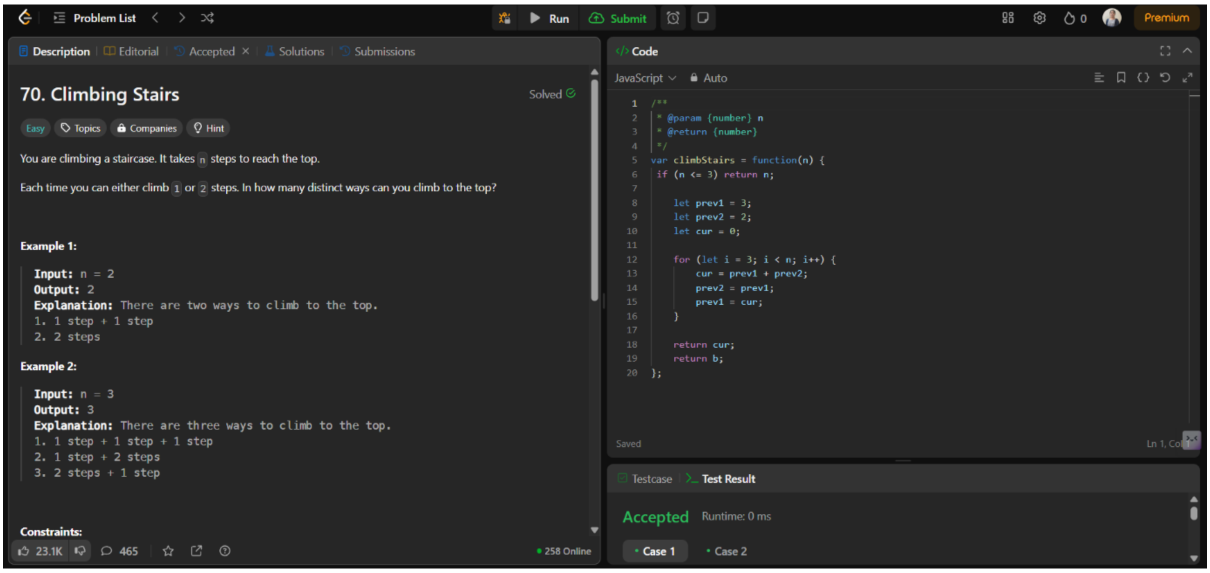
Time Complexity = O(n)

- You have a for loop that starts at i = 3 and runs until i < n, so the loop runs approximately n - 3 times.
- Each iteration does a constant amount of work (addition and assignment).
- Therefore, time complexity = O(n).

Space Complexity = O (1)

- You are only using a fixed number of variables (prev1, prev2, cur), no additional space grows with n.
- Thus, space complexity = O (1) (constant space).

Screenshot:



70. Climbing Stairs

Solved

You are climbing a staircase. It takes n steps to reach the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

Example 1:

```
Input: n = 2
Output: 2
Explanation: There are two ways to climb to the top.
1. 1 step + 1 step
2. 2 steps
```

Example 2:

```
Input: n = 3
Output: 3
Explanation: There are three ways to climb to the top.
1. 1 step + 1 step + 1 step
2. 1 step + 2 steps
3. 2 steps + 1 step
```

Constraints:

```javascript
/**
 * @param {number} n
 * @return {number}
 */
var climbStairs = function(n) {
 if (n <= 3) return n;

    let prev1 = 3;
    let prev2 = 2;
    let cur = 0;

    for (let i = 3; i < n; i++) {
        cur = prev1 + prev2;
        prev2 = prev1;
        prev1 = cur;
    }

    return cur;
    return b;
};
```

Saved

Ln 1, Col 1

Accepted Runtime: 0 ms

Question: https://leetcode.com/problems/merge-two-sorted-lists/description/

Answer:https://leetcode.com/problems/merge-two-sorted-lists/submissions/1628594172

Description:

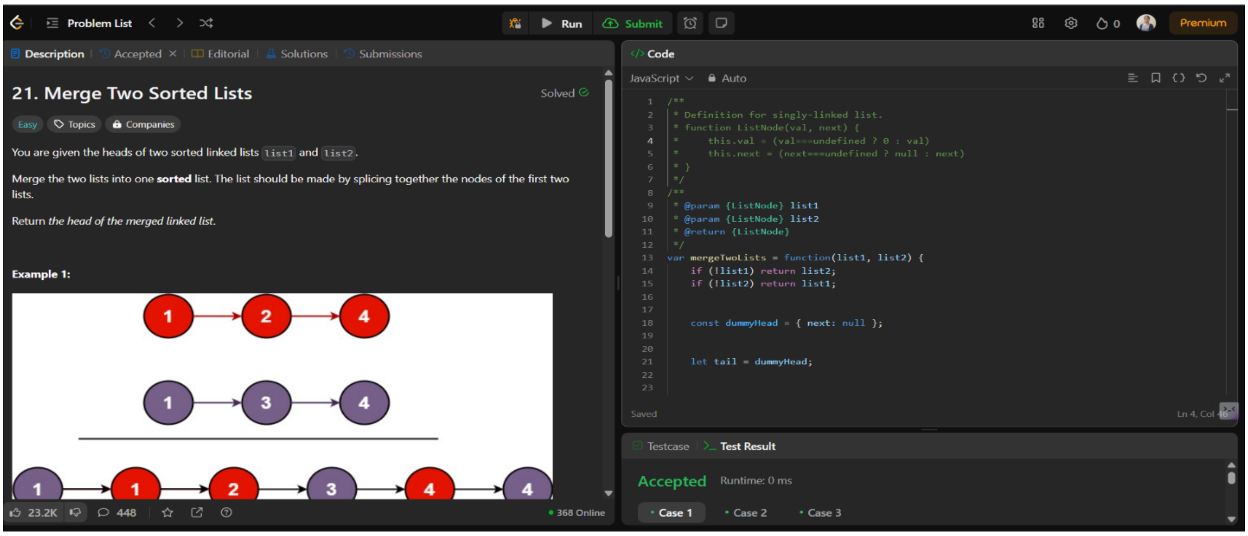
Time Complexity: O (n + m)

- Each node from list1 and list2 is visited exactly once.
- At each step, you either move list1 or list2 forward by one node.
- If n is the length of list1 and m is the length of list2, you do a total of n + m operations.

Space Complexity: O (1)

- You are using a few pointers (Dummy Head, tail, list1, list2), but you are not creating any new nodes — you are just rearranging existing nodes.
- extra space used is constant, regardless of the input size.

Screenshot:



21. Merge Two Sorted Lists

You are given the heads of two sorted linked lists `list1` and `list2`.

Merge the two lists into one **sorted** list. The list should be made by splicing together the nodes of the first two lists.

Return *the head of the merged linked list.*

Example 1:

```
/**
 * Definition for singly-linked list.
 * function ListNode(val, next) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.next = (next===undefined ? null : next)
 * }
 */
/**
 * @param {ListNode} list1
 * @param {ListNode} list2
 * @return {ListNode}
 */
var mergeTwoLists = function(list1, list2) {
    if (!list1) return list2;
    if (!list2) return list1;


    const dummyHead = { next: null };


    let tail = dummyHead;


```

Accepted Runtime: 0 ms

Question: https://leetcode.com/problems/palindrome-linked-list/description/

Answer: https://leetcode.com/problems/palindrome-linked-list/submissions/1628597648

Description:

Time Complexity: O(n)

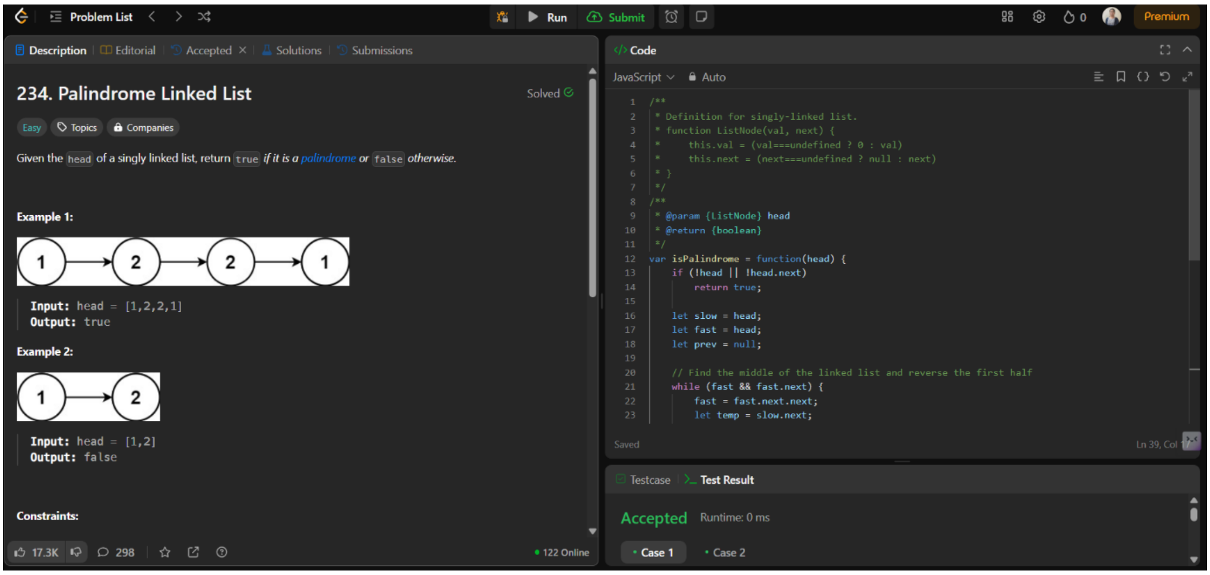
- The while loop (finding middle + reversing first half) goes through about half of the list — O(n/2) operations.
- The second while loop (comparing two halves) also goes through about half — another O(n/2) operations.

Total time = O(n/2) + O(n/2) = O(n).

Space Complexity: O (1)

- You're only using a few pointers (slow, fast, preview, next) — all are constant-sized variables.
- No extra space proportional to input size (no array, no recursion stack).
- So, the space complexity is: O (1)

Screenshot:



234. 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
```

Constraints:

```
/**
 * Definition for singly-linked list.
 * function ListNode(val, next) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.next = (next===undefined ? null : next)
 * }
 */
/**
 * @param {ListNode} head
 * @return {boolean}
 */
var isPalindrome = function(head) {
    if (!head || !head.next)
        return true;

    let slow = head;
    let fast = head;
    let prev = null;

    // Find the middle of the linked list and reverse the first half
    while (fast && fast.next) {
        fast = fast.next.next;
        let temp = slow.next;
```

Accepted Runtime: 0 ms

Question: https://leetcode.com/problems/palindrome-linked-list/description/

Answer: https://leetcode.com/problems/linked-list-cycle/submissions/1628599176

Description:

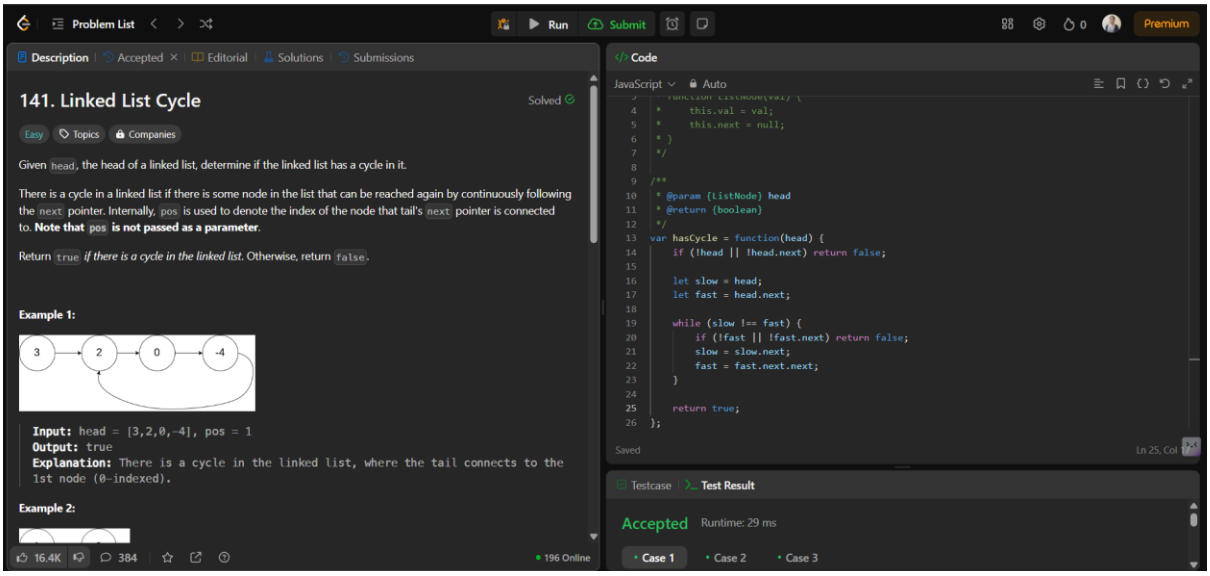
Time Complexity: O(n)

- In the worst case, the fast pointer moves through all the nodes of the linked list.
- If there is no cycle, fast will reach the end (null), visiting each node once → O(n).
- If there is a cycle, the fast and slow pointers will meet somewhere inside the cycle
- The distance before entering the cycle is at most n, and once inside, the meeting happens in at most n steps → O(n)

Space Complexity: O (1)

- Only two pointers (fast and slow) are used.
- No extra data structures are needed (no arrays, hash maps, etc.).
- Final space complexity: O (1) (constant space).

Screenshot:


Problem List
Run
Submit
Premium
Description | Accepted | Editorial | Solutions | Submissions
Code
JavaScript
Auto

141. Linked List Cycle

Solved

Given `head`, the head of a linked list, determine if the linked list has a cycle in it.

There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the `next` pointer. Internally, `pos` is used to denote the index of the node that tail's `next` pointer is connected to. **Note that `pos` is not passed as a parameter.**

Return `true` *if there is a cycle in the linked list*. Otherwise, return `false`.

Example 1:

```
Input: head = [3,2,0,-4], pos = 1
Output: true
Explanation: There is a cycle in the linked list, where the tail connects to the
1st node (0-indexed).
```

Example 2:

```
 * function ListNode(val) {
 *     this.val = val;
 *     this.next = null;
 * }
 */

/**
 * @param {ListNode} head
 * @return {boolean}
 */
var hasCycle = function(head) {
    if (!head || !head.next) return false;

    let slow = head;
    let fast = head.next;

    while (slow !== fast) {
        if (!fast || !fast.next) return false;
        slow = slow.next;
        fast = fast.next.next;
    }

    return true;
};
```

Saved

Ln 25, Col 17

Accepted Runtime: 29 ms

Question: https://leetcode.com/problems/linked-list-cycle/description/

Answer:https://leetcode.com/problems/remove-nth-node-from-end-of-list/submissions/1628603281

Description:

Time Complexity: O(L)

O(L) — where L is the length of the linked list.

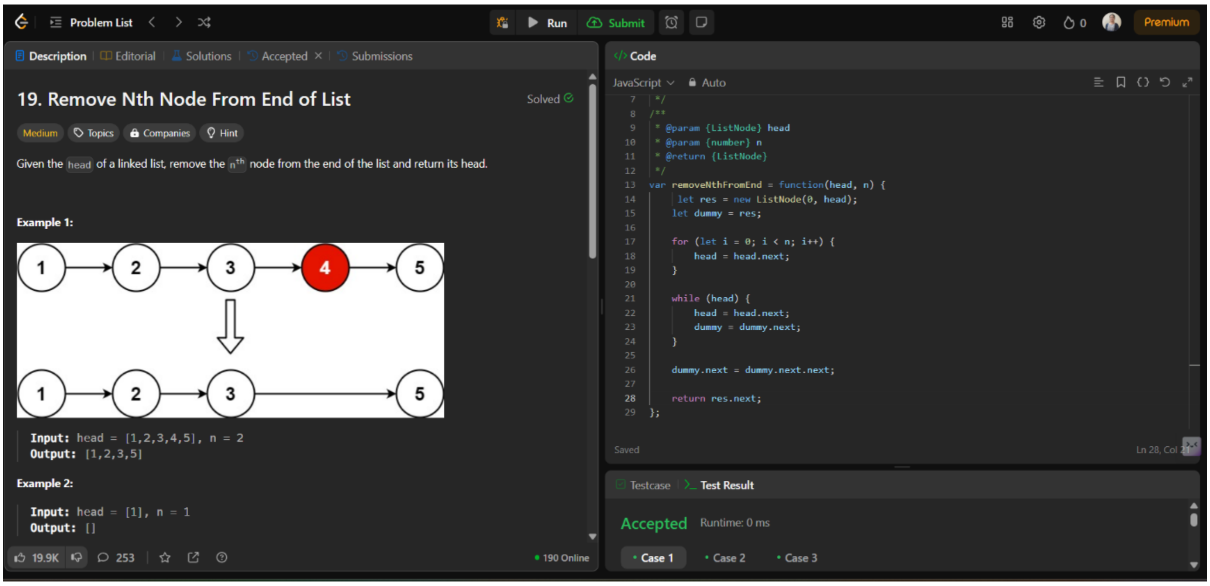
- The function first advances the head pointer by n steps — O(n).
- Then it continues to traverse the rest of the list along with dummy — up to O (L - n) steps.
- Therefore, the total number of operations is proportional to the length of the list: O(L).

Space Complexity: O (1)

O (1) — constant space.

- No extra data structures are used that grow with input size.
- Only a few pointers (res, dummy, and head) are used.

Screenshot:



19. Remove Nth Node From End of List

Given the head of a linked list, remove the n^{th} node from the end of the list and return its head.

Example 1:

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

Example 2:

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

```
 */
/**
 * @param {ListNode} head
 * @param {number} n
 * @return {ListNode}
 */
var removeNthFromEnd = function(head, n) {
     let res = new ListNode(0, head);
    let dummy = res;

    for (let i = 0; i < n; i++) {
        head = head.next;
    }

    while (head) {
        head = head.next;
        dummy = dummy.next;
    }

    dummy.next = dummy.next.next;

    return res.next;
};
```

Accepted Runtime: 0 ms

Question: https://leetcode.com/problems/remove-nth-node-from-end-of list/description/

Answer: https://leetcode.com/problems/powx-n/submissions/1628605732

Description:

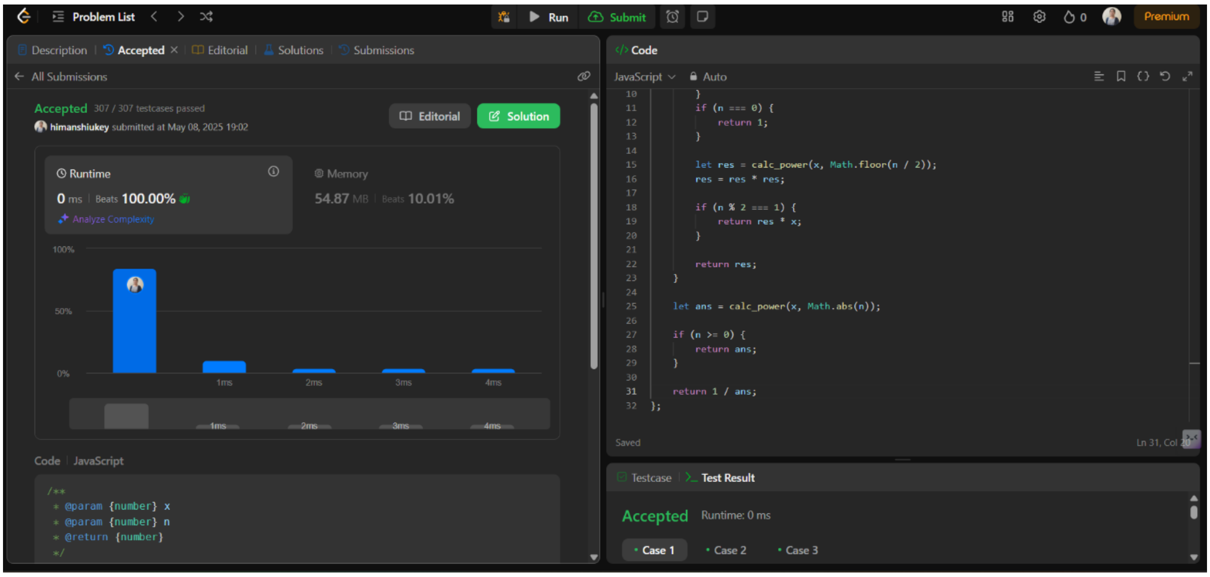
Time Complexity: O (log n)

- The algorithm reduces the exponent n by half in each recursive call (or iteration).
- This is typical of fast exponentiation, where:
- If n is even, it computes pow (x * x, n / 2)
- If n is odd, it computes x * pow (x * x, (n - 1) / 2)
- Hence, the number of steps is proportional to log n.

Space Complexity:

- If implemented recursively: O (log n) — due to the call stack depth.
- If implemented iteratively: O (1) — only a few variables are used (no recursive stack).

Screenshot:



Accepted 307 / 307 testcases passed

himanshiukey submitted at May 08, 2025 19:02

Runtime 0 ms | Beats 100.00%

Memory 54.87 MB | Beats 10.01%

Code | JavaScript

```
/**
 * @param {number} x
 * @param {number} n
 * @return {number}
 */
```

```
        }
        if (n === 0) {
            return 1;
        }

        let res = calc_power(x, Math.floor(n / 2));
        res = res * res;

        if (n % 2 === 1) {
            return res * x;
        }

        return res;
    }

    let ans = calc_power(x, Math.abs(n));

    if (n >= 0) {
        return ans;
    }

    return 1 / ans;
};
```

Accepted Runtime: 0 ms

Question: https://leetcode.com/problems/powx-n/description/

Answer:https://leetcode.com/problems/delete-node-in-a-linked-list/submissions/1628607345

Description:

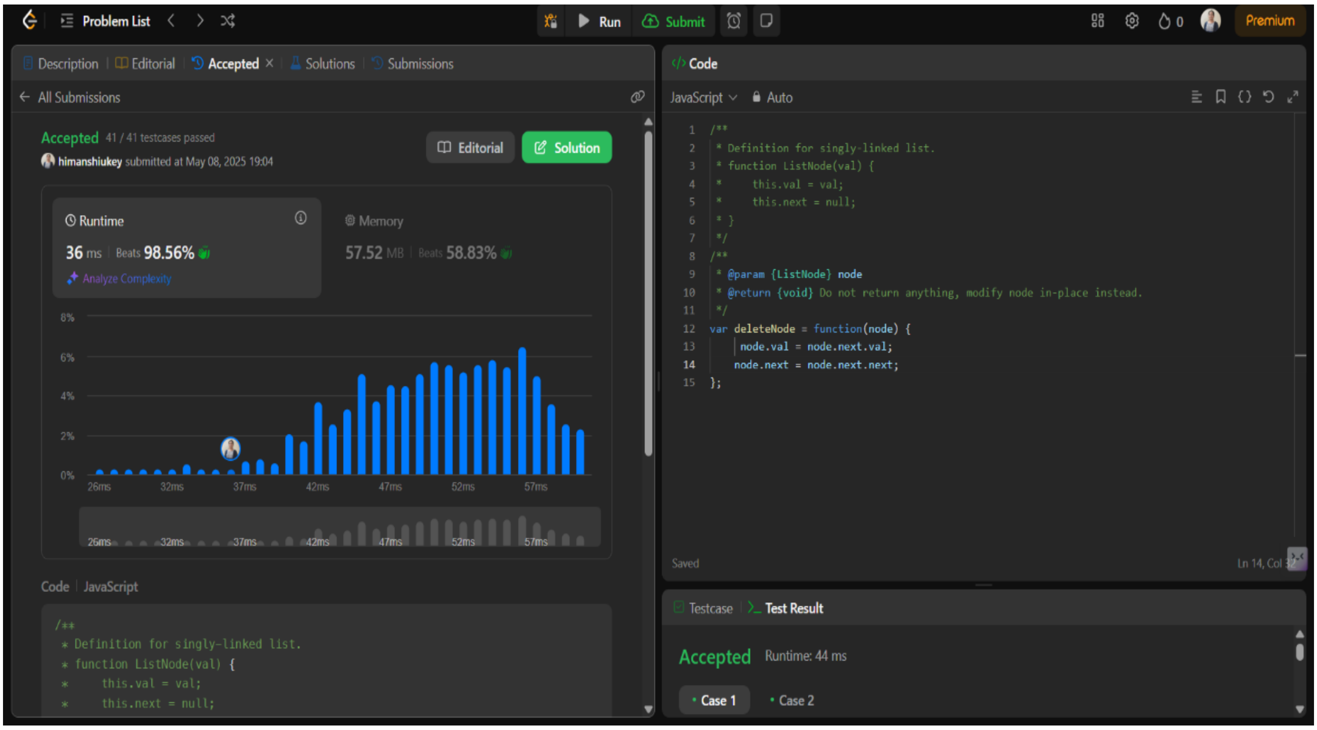
Time Complexity: 0(1)

- constant time.
- It only accesses and modifies two nodes.

Space Complexity: O (1)

- No extra space use

Screenshot:



Accepted 41 / 41 testcases passed

himanshiukey submitted at May 08, 2025 19:04

36 ms | Beats 98.56%

57.52 MB | Beats 58.83%

Code | JavaScript

```
/**
 * Definition for singly-linked list.
 * function ListNode(val) {
 *     this.val = val;
 *     this.next = null;
```

```
/**
 * Definition for singly-linked list.
 * function ListNode(val) {
 *     this.val = val;
 *     this.next = null;
 * }
 */
/**
 * @param {ListNode} node
 * @return {void} Do not return anything, modify node in-place instead.
 */
var deleteNode = function(node) {
    node.val = node.next.val;
    node.next = node.next.next;
};
```

Accepted Runtime: 44 ms

Thank You For This Assignment