LEETCODES

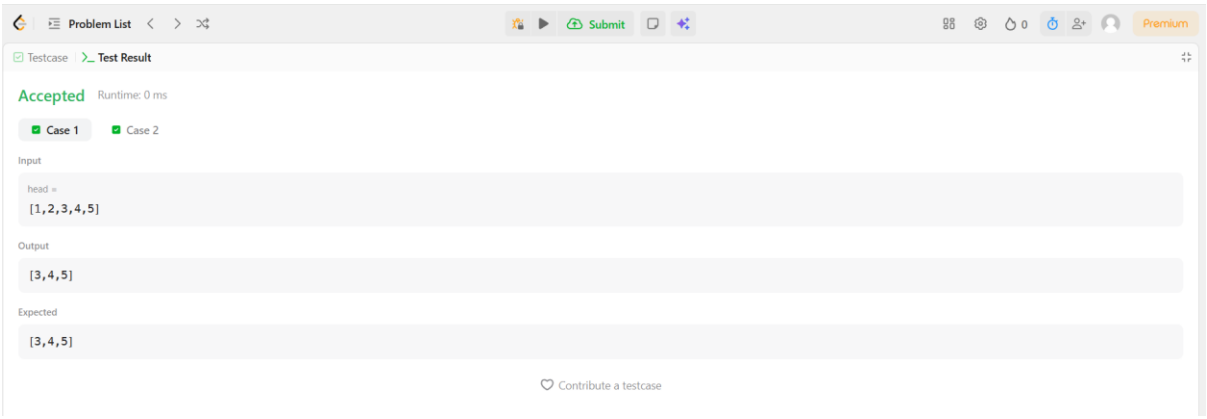
4b. Given the head of a singly linked list, return *the middle node of the linked list*.

If there are two middle nodes, return the second middle node.

</> Code

C ⌄ 🔒 Auto

```c

struct ListNode* middleNode(struct ListNode* head) {
    struct ListNode *slow = head;
    struct ListNode *fast = head;

    while (fast != NULL && fast->next != NULL) {
        slow = slow->next;
        fast = fast->next->next;
    }

    return slow;
}

```

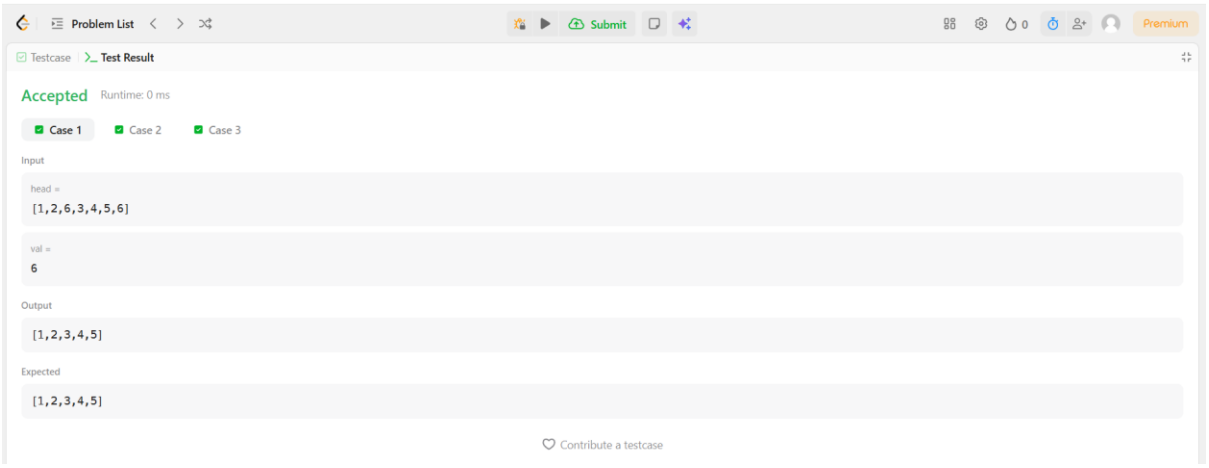


5bGiven the head of a linked list and an integer val, remove all the nodes of the linked list that has Node.val == val, and return *the new head*

Code

C Auto

```c
struct ListNode* removeElements(struct ListNode* head, int val) {
    // Remove nodes from beginning if needed
    while (head != NULL && head->val == val) {
        struct ListNode* temp = head;
        head = head->next;
        free(temp);
    }

    struct ListNode* curr = head;

    // Remove nodes from rest of the list
    while (curr != NULL && curr->next != NULL) {
        if (curr->next->val == val) {
            struct ListNode* temp = curr->next;
            curr->next = temp->next;
            free(temp);
        } else {
            curr = curr->next;
        }
    }

    return head;
}
```



7b. Given head, the head of a linked list, determine if the linked list has a cycle in it.

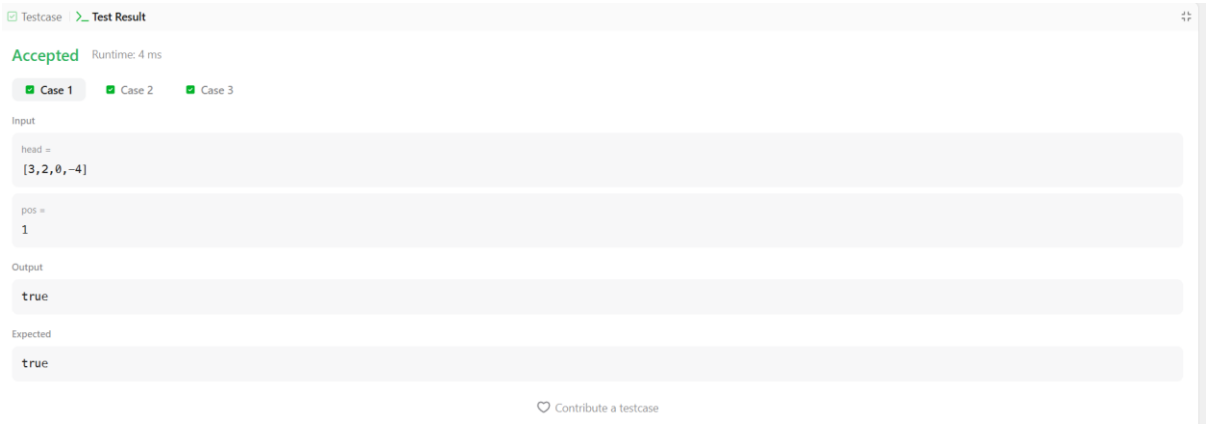
There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer. Internally, pos is used to denote the index of the node that tail's next pointer is connected to. Note that pos is not passed as a parameter.

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

Code

C Auto

```c
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     struct ListNode *next;
 * };
 */

bool hasCycle(struct ListNode *head) {
    struct ListNode *slow = head;
    struct ListNode *fast = head;

    while (fast != NULL && fast->next != NULL) {
        slow = slow->next;          // move 1 step
        fast = fast->next->next;    // move 2 steps

        if (slow == fast) {
            return true;            // cycle found
        }
    }

    return false;   // no cycle
}

```



8b. You are given two binary trees root1 and root2.

Imagine that when you put one of them to cover the other, some nodes of the two trees are overlapped while the others are not. You need to merge the two trees into a new binary tree. The merge rule is that if two nodes

overlap, then sum node values up as the new value of the merged node. Otherwise, the NOT null node will be used as the node of the new tree.

Return *the merged tree*.

Note: The merging process must start from the root nodes of both trees.