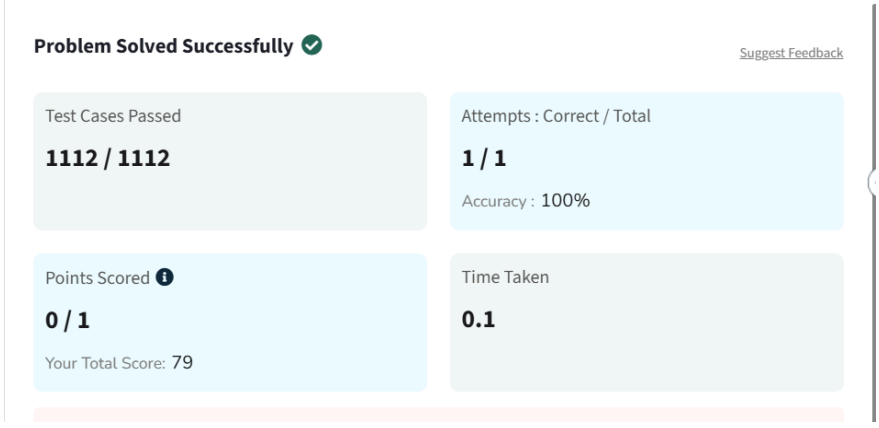
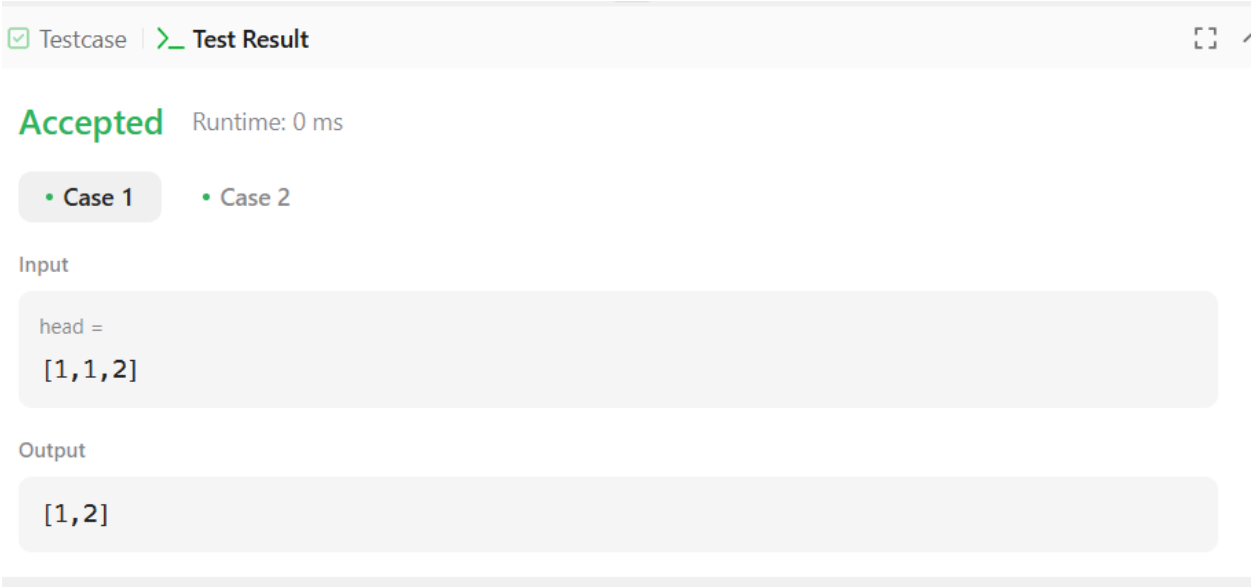
# 1. Print Linked List

```
/*
struct Node {
    int data;
    struct Node* next;

    Node(int x) {
        data = x;
        next = nullptr;
    }
};
*/
/*
    Print elements of a linked list on console
    Head pointer input could be NULL as well for empty list
*/

class Solution {
  public:
    // Function to display the elements of a linked list in same line
    void printList(Node *head) {
        struct Node*ptr;
        ptr=head;
        while(ptr!=NULL){
            cout<<ptr->data<<" ";
            ptr=ptr->next;
        }
    }
};
```



## 2.Remove duplicates from a sorted list

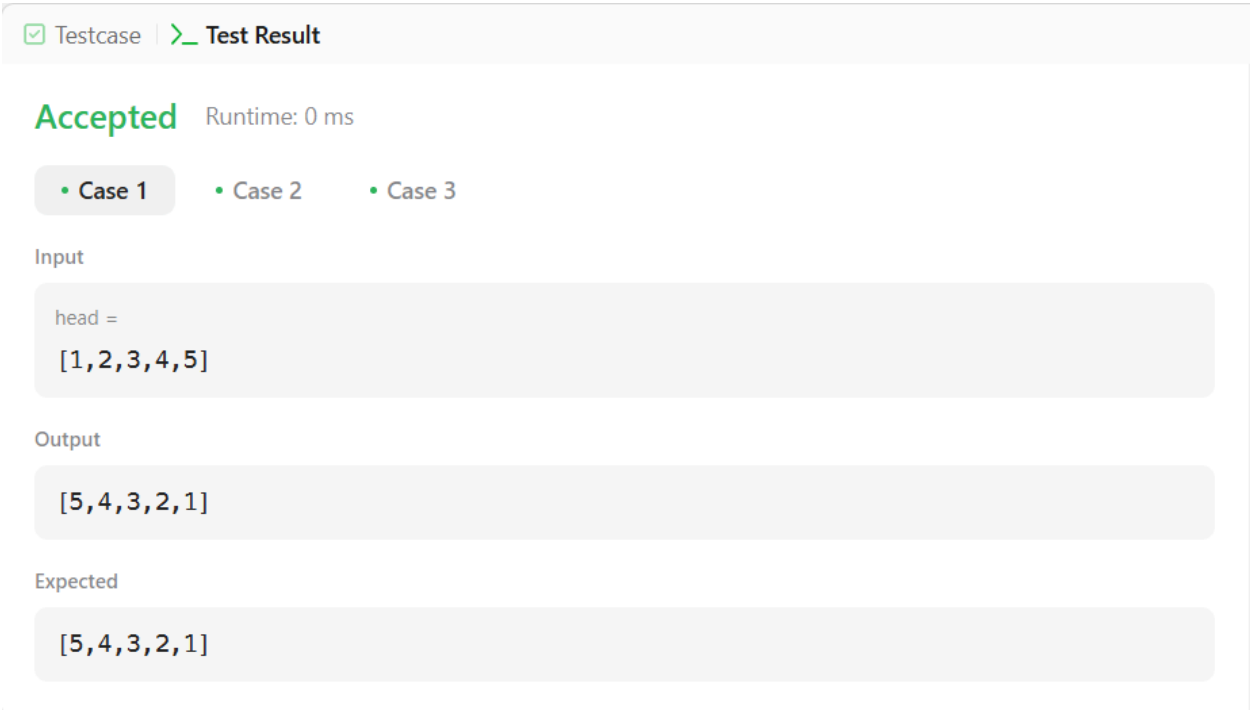
```
class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head) {
        ListNode* current = head;
        while (current && current->next) {
            if (current->val == current->next->val) {
                current->next = current->next->next;
            } else {
                current = current->next;
            }
        }
        return head;
    }
};
```




## 3.Reverse a linked list

```
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        // Initialize pointers
        ListNode* prev = nullptr; // Previous node starts as NULL
        ListNode* next = nullptr; // Next node
        ListNode* curr = head;    // Current node starts at the head

        // Traverse the list
        while (curr != nullptr) {
```

```
            // Save the next node
            next = curr->next;

            // Reverse the link
            curr->next = prev;

            // Move pointers forward
            prev = curr; // Move prev to the current node
            curr = next; // Move curr to the next node
        }

        // prev is now the new head of the reversed list
        return prev;
    }
};
```

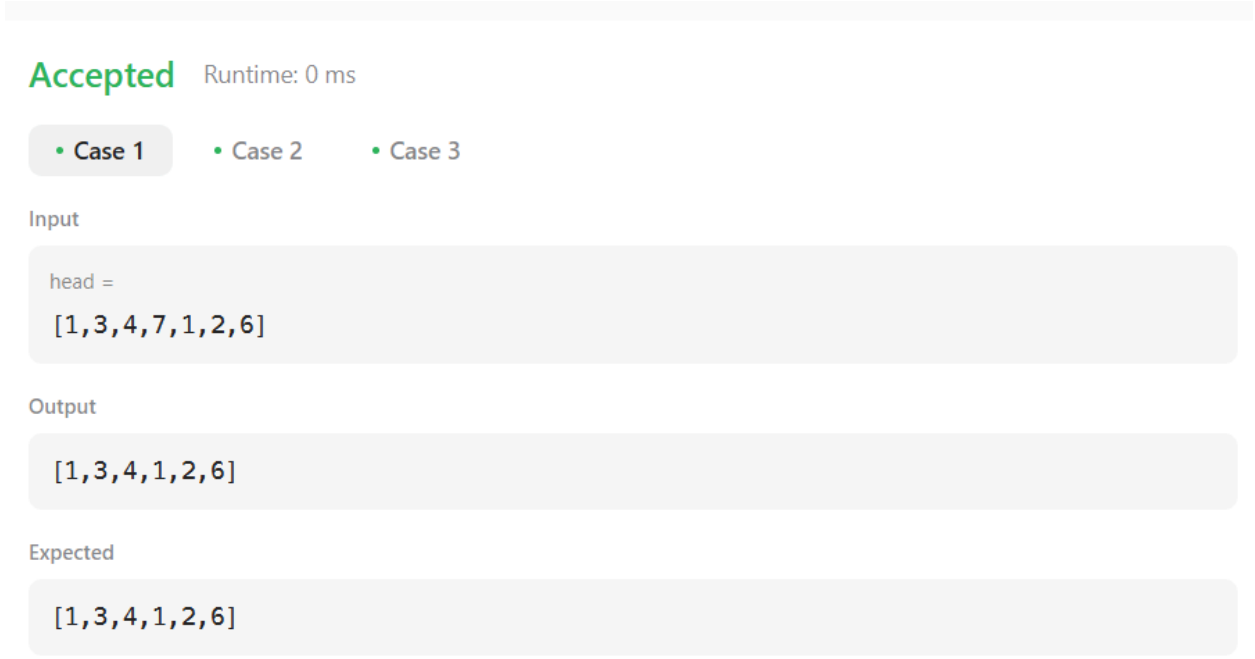



## 4.Delete middle node of a list:

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
```

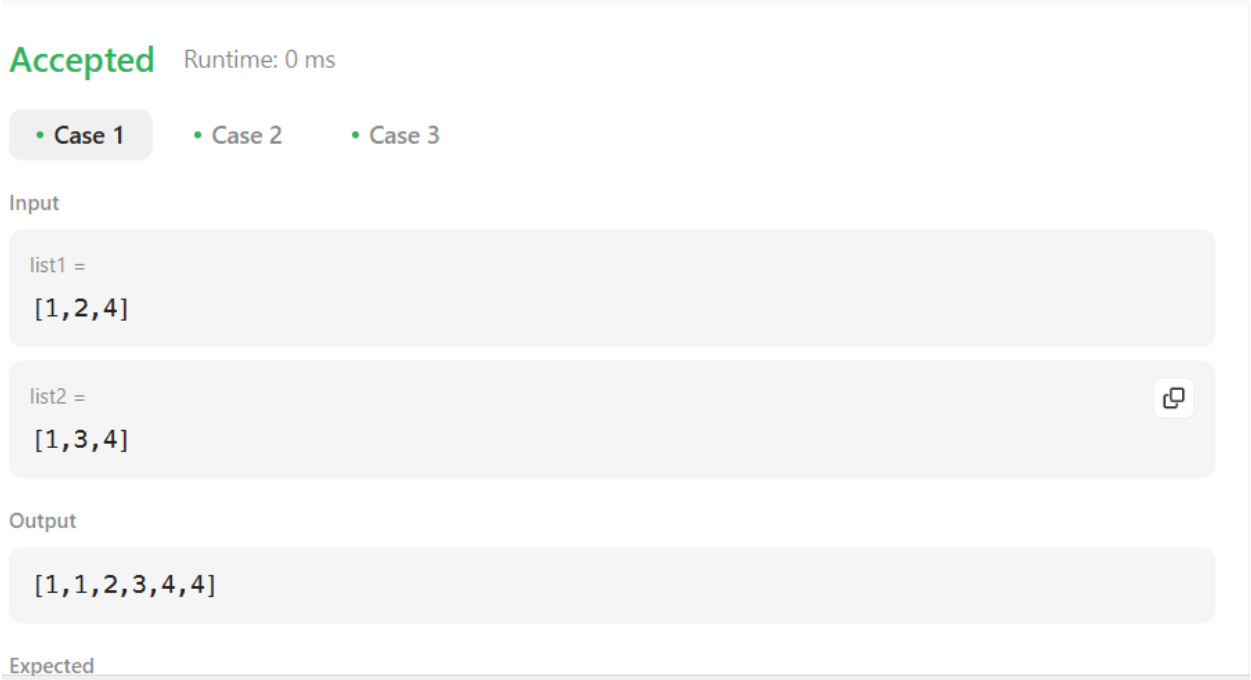
```
 * };
 */
class Solution {
public:
    ListNode* deleteMiddle(ListNode* head) {
        // ..............................................................
        // .....................Optimal Approach.....................
        // ..............................................................
        if(!head || !head->next) return nullptr;
        ListNode*slow = head;
        ListNode*fast = head->next->next;
        while(fast !=nullptr && fast->next != nullptr){
            fast = fast->next->next;
            slow = slow->next;
        }
        ListNode*delnode = slow->next;
        slow->next =slow->next->next;
        delete delnode;
        return head;
        // ......... Space Complexity : O(1).....................
        // ......... Time Complexity: O(N + N/2) .....................

        // ..................................................................
        // .....................Brute Force Approach.....................
        // ..................................................................
        // if(!head || !head->next) return nullptr;
        // int count = 0;
        // ListNode*temp=head;
        // while(temp!=nullptr){
            // count++;
            // temp=temp->next;
        // }
        // count = count/2;
        // temp = head;
        // while(temp != nullptr){
            // count--;
            // if(count == 0){
                // ListNode*delmid=temp->next;
                // temp->next = temp->next->next;
                // delete delmid;
                // break;
            // }
            // temp = temp->next;
        // }
        // return temp;
    // ..................... Time Complexity: O(N + N/2) ..............
```

```
    // ..... Space Complexity : O(1) .....
    }

};
```



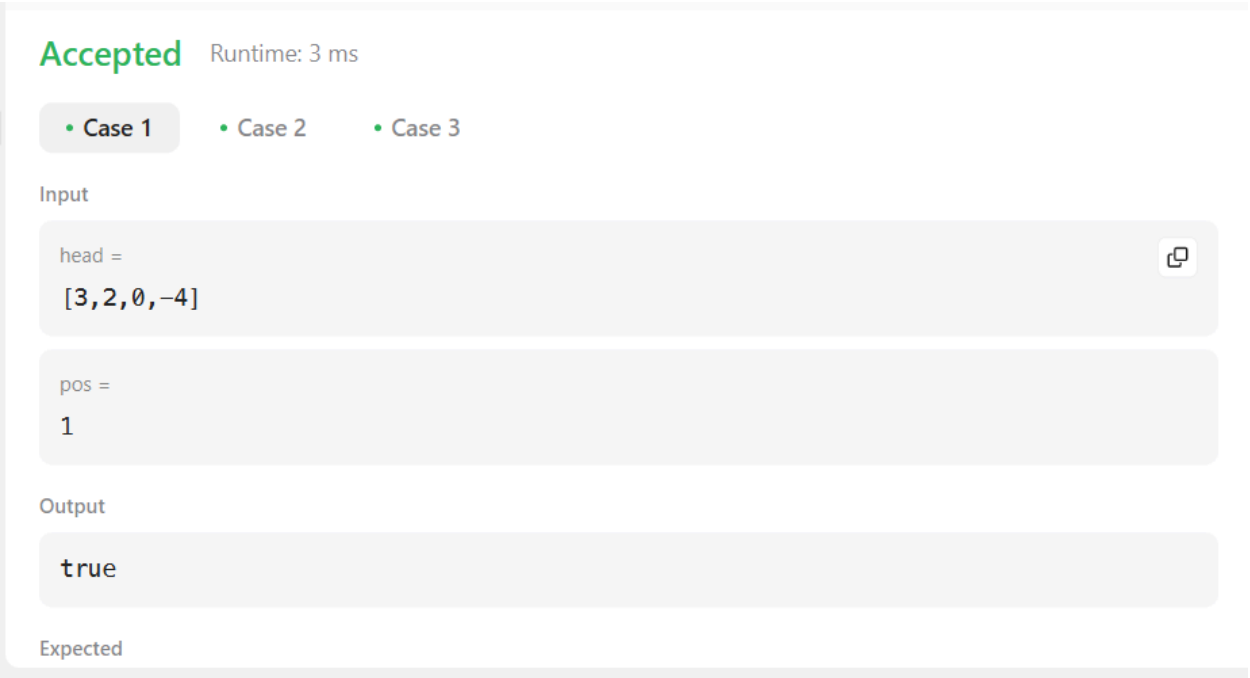

## 5.Merge two sorted linked lists:

```
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* list1, ListNode* list2) {
        if(list1 == NULL || list2 == NULL){
            return list1 == NULL ? list2 : list1;
        }

        if(list1->val <= list2->val){
            list1->next =  mergeTwoLists(list1->next, list2);
            return list1;
        }
        else{
            list2->next =  mergeTwoLists(list1, list2->next);
            return list2;
        }
    }
};
```




## 6.Detect a cycle in a linked list:

```
class Solution {
public:
    bool hasCycle(ListNode* head) {
        if (head == NULL || head->next == NULL) {
            return false;
        }
        ListNode* slow = head;
        ListNode* fast = head->next;
        while (fast != slow) {
            if (fast->next == NULL || fast->next->next == NULL) {
                return false;
            }
            slow = slow->next;
            fast = fast->next->next;
        }
        return true;
    }
};
```




# 7. Rotate list

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* rotateRight(ListNode* head, int k) {
        // base condition
        if(head==NULL || head->next==NULL || k==0) return head;

        ListNode* curr=head;
        int count=1;
        while(curr->next!=NULL){
            curr=curr->next;
            count++;
        }
        curr->next=head;
        k=count-(k%count);
        while(k-->0){
```

```
            curr=curr->next;
        }
        head=curr->next;
        curr->next=NULL;  // curr points to tail node sorta

        return head;

    }
};
```

head =

```
[1,2,3,4,5]
```

k =

```
2
```

Output

```
[4,5,1,2,3]
```

Expected

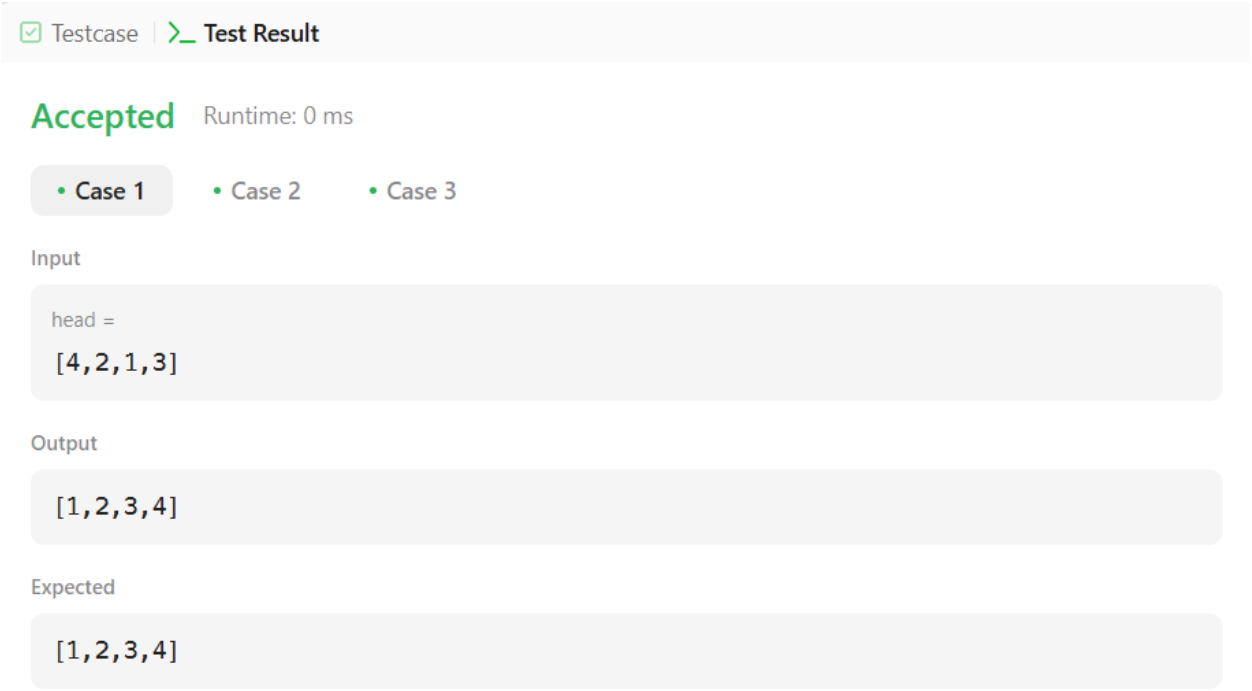
```
[4,5,1,2,3]
```

Contribute a testcase

# 8. Sort List:

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* sortList(ListNode* head) {
        if(head==NULL) return NULL;
        ListNode *damy=NULL,*current=NULL;
```

```
        vector<int>v;
        while(head){
            v.push_back(head->val);
            head=head->next;
        }
        sort(v.begin(),v.end());
        for(auto u:v){
            ListNode *tem = new ListNode(u);
            if(damy==NULL){
                damy=tem;
                current=tem;
            }
            else{
                current->next=tem;
                current=current->next;
            }
        }
        return damy;
    }
};
```



# 9.Merge k sorted lists

```
class Solution {
public:
    ListNode* mergeKLists(vector<ListNode*>& lists) {
        if (lists.empty()) {
            return nullptr;
        }
        return mergeKListsHelper(lists, 0, lists.size() - 1);
    }

    ListNode* mergeKListsHelper(vector<ListNode*>& lists, int start, int end) {
        if (start == end) {
            return lists[start];
        }
        if (start + 1 == end) {
            return merge(lists[start], lists[end]);
        }
        int mid = start + (end - start) / 2;
        ListNode* left = mergeKListsHelper(lists, start, mid);
        ListNode* right = mergeKListsHelper(lists, mid + 1, end);
        return merge(left, right);
    }

    ListNode* merge(ListNode* l1, ListNode* l2) {
        ListNode* dummy = new ListNode(0);
        ListNode* curr = dummy;

        while (l1 && l2) {
            if (l1->val < l2->val) {
                curr->next = l1;
                l1 = l1->next;
            } else {
                curr->next = l2;
                l2 = l2->next;
            }
            curr = curr->next;
        }

        curr->next = l1 ? l1 : l2;

        return dummy->next;
    }
};
```

## Accepted Runtime: 0 ms

• Case 1 • Case 2 • Case 3

Input

lists =

```
[[1,4,5],[1,3,4],[2,6]]
```

Output

```
[1,1,2,3,4,4,5,6]
```

Expected

```
[1,1,2,3,4,4,5,6]
```