Assignment-3

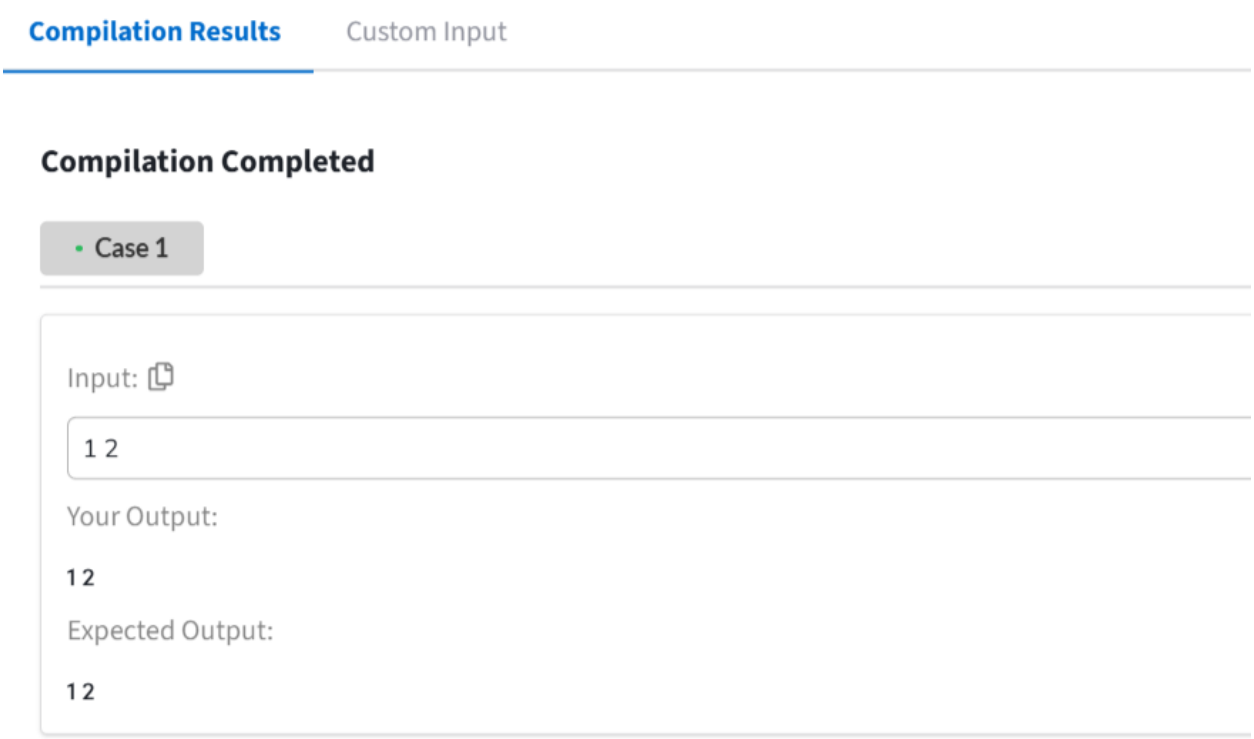
Student Name: Lakhan Singh
UID: 22BCS12194
Branch: BE-CSE
Section/Group: 608-B
Semester: 6th
Date of Performance: 6/02/25
Subject Name: Advance Programming Lab-2
Subject Code: 22CSP-351

1. Aim: Print Linked List.

Code:

```
class Solution {
  public:
    // Function to display the elements of a linked list in the same line
    void printList(Node *head) {
        Node* current = head;
        while (current != nullptr) {
            cout << current->data;
            if (current->next != nullptr) {
                cout << " "; // Print space only between elements
            }
            current = current->next;
        }
    }
};
```

Output:



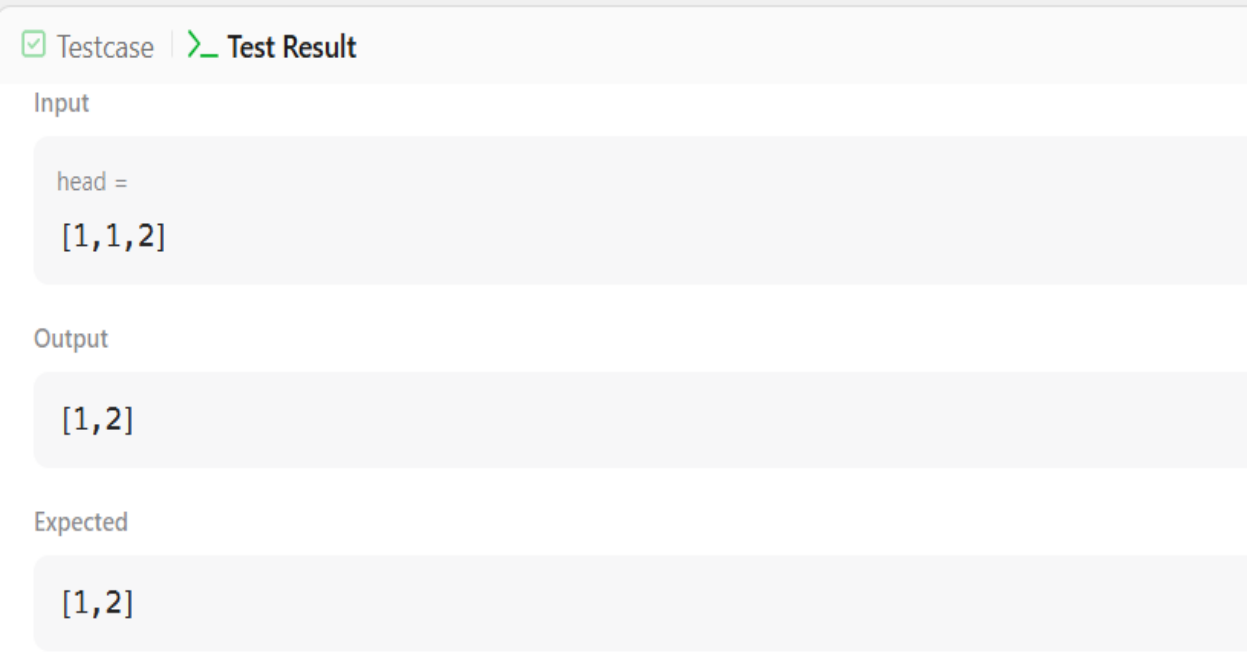
2. Aim: Remove duplicates from a sorted list.

Code:

```
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; // Skip duplicate node
```

```
            } else {
                current = current->next; // Move to next node
            }
        }
        return head;
    }
};
```

Output:

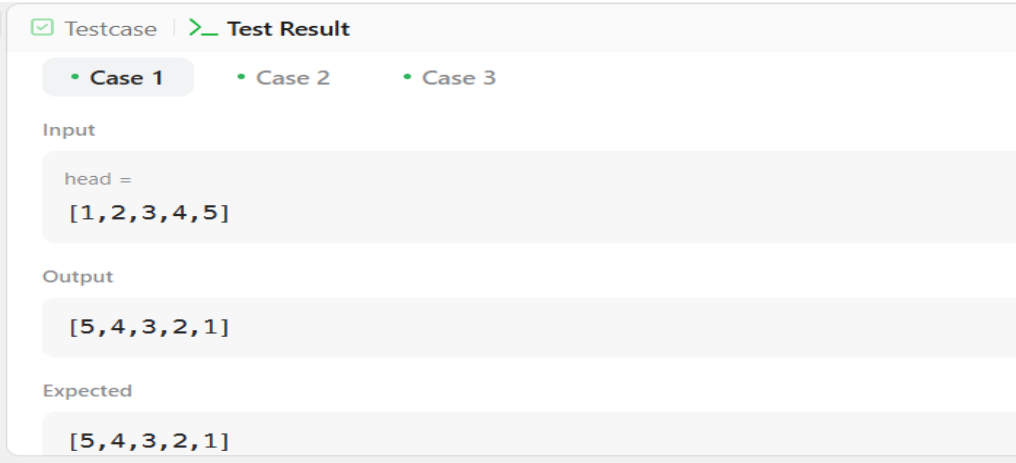


3. Aim: Reverse a linked list.

Code:

```
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        ListNode* prev = nullptr;
        ListNode* current = head;
        while (current) {
            ListNode* nextNode = current->next; // Store next node
            current->next = prev; // Reverse the link
            prev = current; // Move prev forward
            current = nextNode; // Move current forward
        }
        return prev; // New head of the reversed list
    }
};
```

Output:



4. Aim: Delete middle node of a list.

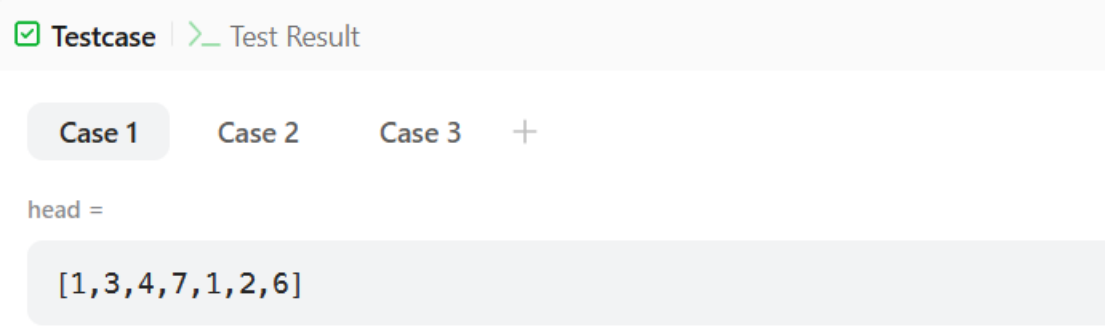
Code:

```
class Solution {
public:
    ListNode* deleteMiddle(ListNode* head) {
        if (!head || !head->next) return nullptr;
        int count = 0;
        ListNode* temp = head;
        while (temp) {
            count++;
            temp = temp->next;
        }
        int middle = count / 2;

        temp = head;
        for (int i = 0; i < middle - 1; i++) {
            temp = temp->next;
        }
        temp->next = temp->next->next;

        return head;
    }
};
```

Output:



5. Aim: Merge two sorted linked lists.

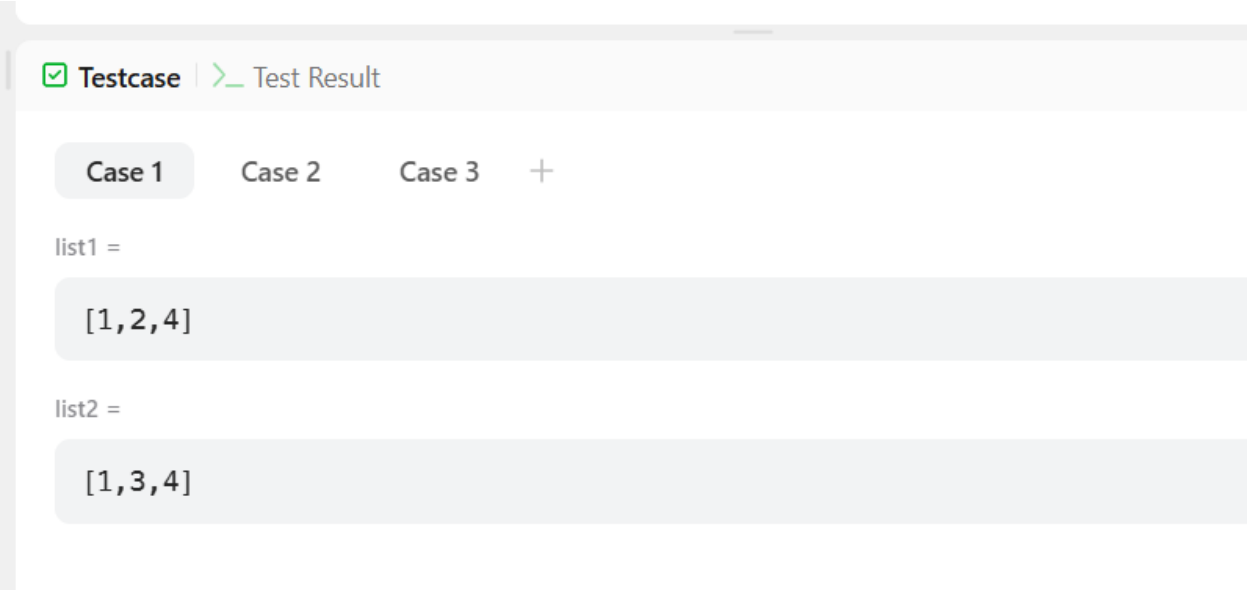
Code:

```
 class Solution {
public:
    ListNode* mergeTwoLists(ListNode* list1, ListNode* list2) {
        // Create a dummy node to simplify edge cases
        ListNode dummy(0);
        ListNode* tail = &dummy;
        // Traverse both lists and merge them in sorted order
        while (list1 && list2) {
            if (list1->val < list2->val) {
                tail->next = list1;
                list1 = list1->next;
            } else {
```

```
            tail->next = list2;
            list2 = list2->next;
        }
        tail = tail->next;
    }
    // Attach any remaining nodes from either list
    tail->next = list1 ? list1 : list2;

    return dummy.next; // Return the merged list starting from the first actual node
  }
};
```

Output:



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

Code:

```
class Solution {
public:
    bool hasCycle(ListNode *head) {
        ListNode *slow = head, *fast = head;
        while (fast && fast->next) {  // Ensure fast and fast->next are not null
            slow = slow->next;        // Move slow one step
            fast = fast->next->next;  // Move fast two steps
            if (slow == fast) {       // If they meet, cycle exists
                return true;
            } }
        return false;  // No cycle
    }
};
```

Output:

7. Aim: Rotate a list.

Code:

```
class Solution {
public:
    ListNode* rotateRight(ListNode* head, int k) {
        if (!head || !head->next || k == 0) return head; // Edge cases: empty list or no rotation needed

        // Step 1: Find the length of the list
        ListNode* temp = head;
        int length = 1;  // Start counting from 1 because we're already at head
        while (temp->next) {
            temp = temp->next;
            length++;
        }

        // Step 2: Optimize k
        k = k % length;
        if (k == 0) return head; // If k is a multiple of length, no change needed

        // Step 3: Find the new tail (at position length - k - 1)
        temp->next = head; // Connect tail to head to make a circular list
        temp = head;
        for (int i = 0; i < length - k - 1; i++) {
```

```
            temp = temp->next;
        }

        // Step 4: Break the cycle and set the new head
        head = temp->next;
        temp->next = nullptr;

        return head;
    }
};
```

Output:

8. Aim: Sort List.

Code:

```
class Solution {
public:
    ListNode* sortList(ListNode* head) {
        if (!head || !head->next) return head;

        // Step 1: Find the middle of the list
        ListNode* slow = head, *fast = head->next;
        while (fast && fast->next) {
            slow = slow->next;
            fast = fast->next->next;
        }

        // Step 2: Split the list into two halves
        ListNode* mid = slow->next;
        slow->next = nullptr;

        // Step 3: Recursively sort both halves
        ListNode* left = sortList(head);
        ListNode* right = sortList(mid);

        // Step 4: Merge the sorted halves
        return merge(left, right);
```

```
    }

private:
    ListNode* merge(ListNode* l1, ListNode* l2) {
        ListNode dummy(0);
        ListNode* tail = &dummy;

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

        // Append the remaining elements
        tail->next = l1 ? l1 : l2;

        return dummy.next;
    }
};
```

Output:

9. Aim: Merge k sorted lists.

Code:

```
class Solution {
public:
    struct Compare {
        bool operator()(ListNode* a, ListNode* b) {
            return a->val > b->val;  // Min-heap based on node value
        }
    };

    ListNode* mergeKLists(vector<ListNode*>& lists) {
```

```
priority_queue<ListNode*, vector<ListNode*>, Compare> minHeap;

// Push the head of each linked list into the heap
for (ListNode* list : lists) {
    if (list) minHeap.push(list);
}

ListNode dummy(0);  // Dummy node to simplify merging
ListNode* tail = &dummy;

while (!minHeap.empty()) {
    ListNode* minNode = minHeap.top();
    minHeap.pop();

    // Append the smallest node to the merged list
    tail->next = minNode;
    tail = tail->next;

    // Push the next node from the same list into the heap
    if (minNode->next) {
        minHeap.push(minNode->next);
    }
}
```

```
        return dummy.next;  // Return merged linked list
    }
};
```

Output: