**Experiment 3**

**Student Name: Sanju Arora UID: 22BET10059**

**Branch: BE-IT Section/Group: 22BET\_IOT\_702/A**

**Semester: 6th Date of Performance: 31-1-25 Subject Name: Advanced Programming Lab-2 Subject Code: 22ITP-351**

**Problem 1.** Given a linked list. Print all the elements of the linked list separated by space followed.

# Algorithm:

1. Initialize a pointer temp and set it to head.
2. Loop while temp is not nullptr:
   * Print the data of the current node.
   * Move temp to the next node.
3. End loop when temp reaches nullptr.
4. Exit function (No return value since it prints to the console).

# Code:

class Solution { public:

void printList(Node \*head) { Node\* temp = head;

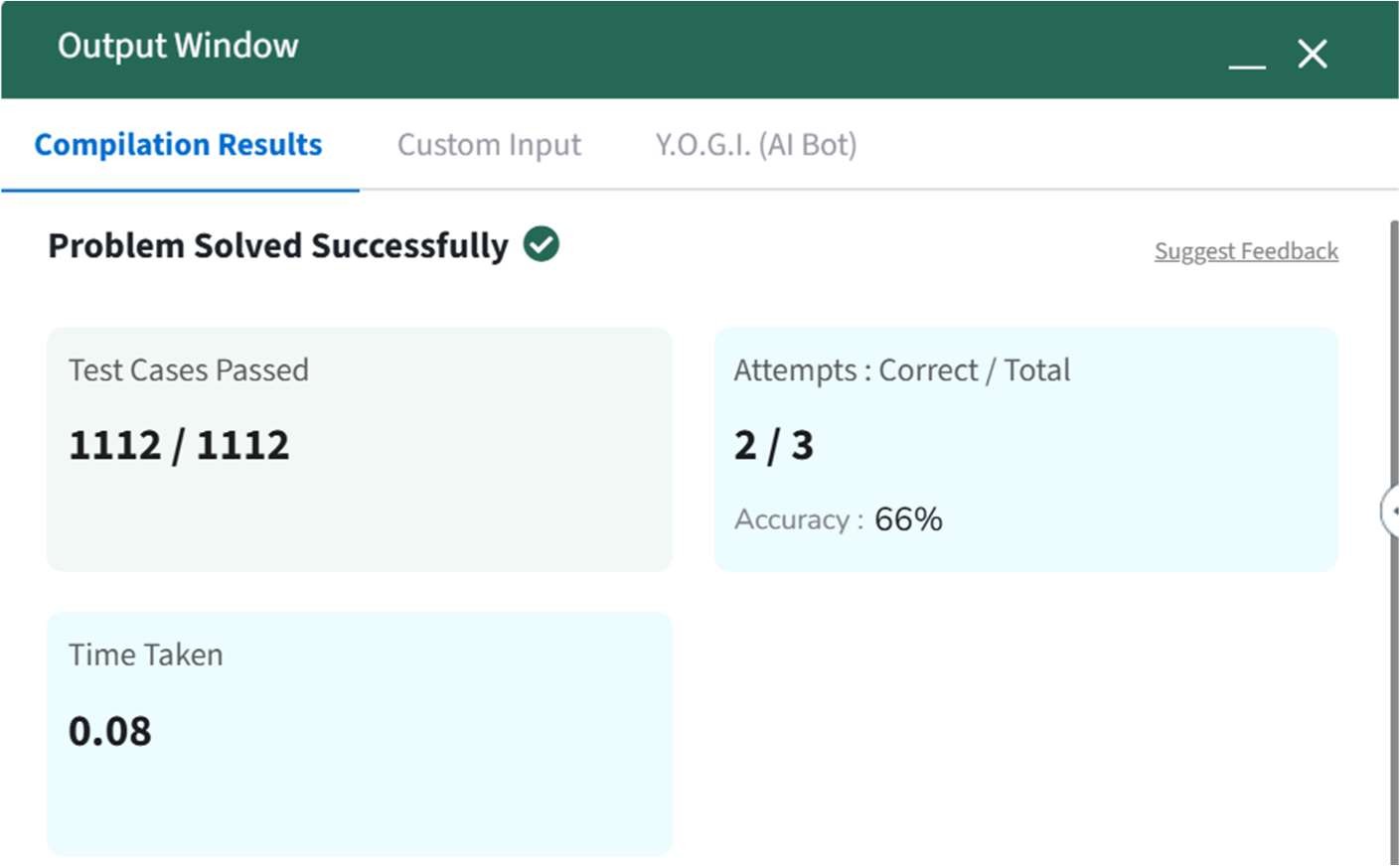
while (temp != nullptr) { cout << temp->data << " "; temp = temp->next;

}

}

};

# Output:



**Problem 2.** Given the head of a sorted linked list, *delete all duplicates such that each element appears only once*. Return *the linked list sorted as well*.

# Algorithm:

1. **Initialize** a pointer current and set it to head.
2. **Traverse** the linked list while current and current->next exist:
   * If current->val == current->next->val:
     + Store the duplicate node in a temporary pointer temp.
     + Update current->next to current->next->next (skipping the duplicate node).
     + Delete temp to free memory.
   * Else, move current to current->next (proceed to the next distinct node).
3. **Return** the modified head pointer after processing.

# Code:

class Solution { public:

ListNode\* deleteDuplicates(ListNode\* head) { ListNode\* current = head;

while (current && current->next) {

if (current->val == current->next->val) {

ListNode\* temp = current->next; current->next = current->next->next; delete temp;

} else {

current = current->next;

}

}

return head;

}

# Output:

# 

**Problem 3.** Given the head of a sorted linked list, *delete all nodes that have duplicate numbers, leaving only distinct numbers from the original list*. Return *the linked list sorted as well*.

# Algorithm:

1. Create a dummy node pointing to head. This helps handle edge cases where the first node gets deleted.
2. Initialize prev pointer to dummy, which will track the last unique node.
3. Traverse the list while head is not nullptr:
   * If head->val == head->next->val, move head forward until all duplicates are skipped.
   * Update prev->next to head->next to remove duplicates.
   * Otherwise, move prev forward.
4. Return dummy->next, as the modified linked list.

# Code:

class Solution { public:

ListNode\* deleteDuplicates(ListNode\* head) { ListNode\* dummy = new ListNode(0, head); ListNode\* prev = dummy

while (head != nullptr)

if (head->next != nullptr && head->val == head->next->val) {

while (head->next != nullptr && head->val == head->next->val) { head = head->next;

}

prev->next = head->next;

} else {

prev = prev->next;

}

head = head->next;

}

return dummy->next;

} };

# Output:

# 

**Problem 4.** Given head, the head of a linked list, determine if the linked list has a cycle in it.

# Algorithm:

## Initialize two pointers:

* + slow moves **one step** at a time.
  + fast moves **two steps** at a time.

1. **Traverse the linked list** while fast and fast->next are not nullptr:
   * Move slow forward by one step.
   * Move fast forward by two steps.
   * If slow == fast, a cycle is detected → **Return true**.
2. If the loop exits (fast reaches nullptr), return **false** (no cycle).

# Code:

class Solution { public:

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

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

fast = fast->next->next;

if (slow == fast) {

return true;

}

}

return false;

}

};

# Output:

# 

**Problem 5.** Given the head of a singly linked list and two integers left and right where left <= right, reverse the nodes of the list from position left to position right, and return *the reversed list*.

# Algorithm:

1. Edge case: If the list is empty or left == right, no need to reverse anything, so return the list as is.
2. Create a dummy node and set it to point to head. This simplifies handling the edge case when the sublist starts at the head.
3. Move the prev pointer to the node just before the left-th position.
4. Reverse the sublist:
   * Start from the left-th position, iteratively reverse the nodes between left and right.
   * For each iteration, adjust pointers so that the nodes between left and right are reversed.
5. Return the modified list starting from dummy->next.

# Code:

class Solution { public:

ListNode\* reverseBetween(ListNode\* head, int left, int right) { if (head == nullptr || left == right) {

return head;

}

ListNode\* dummy = new ListNode(0); dummy->next = head;

ListNode\* prev = dummy;

for (int i = 1; i < left; ++i) { prev = prev->next;

}

ListNode\* curr = prev->next; ListNode\* next = nullptr;

for (int i = 0; i < right - left; ++i) { next = curr->next;

curr->next = next->next; next->next = prev->next; prev->next = next;

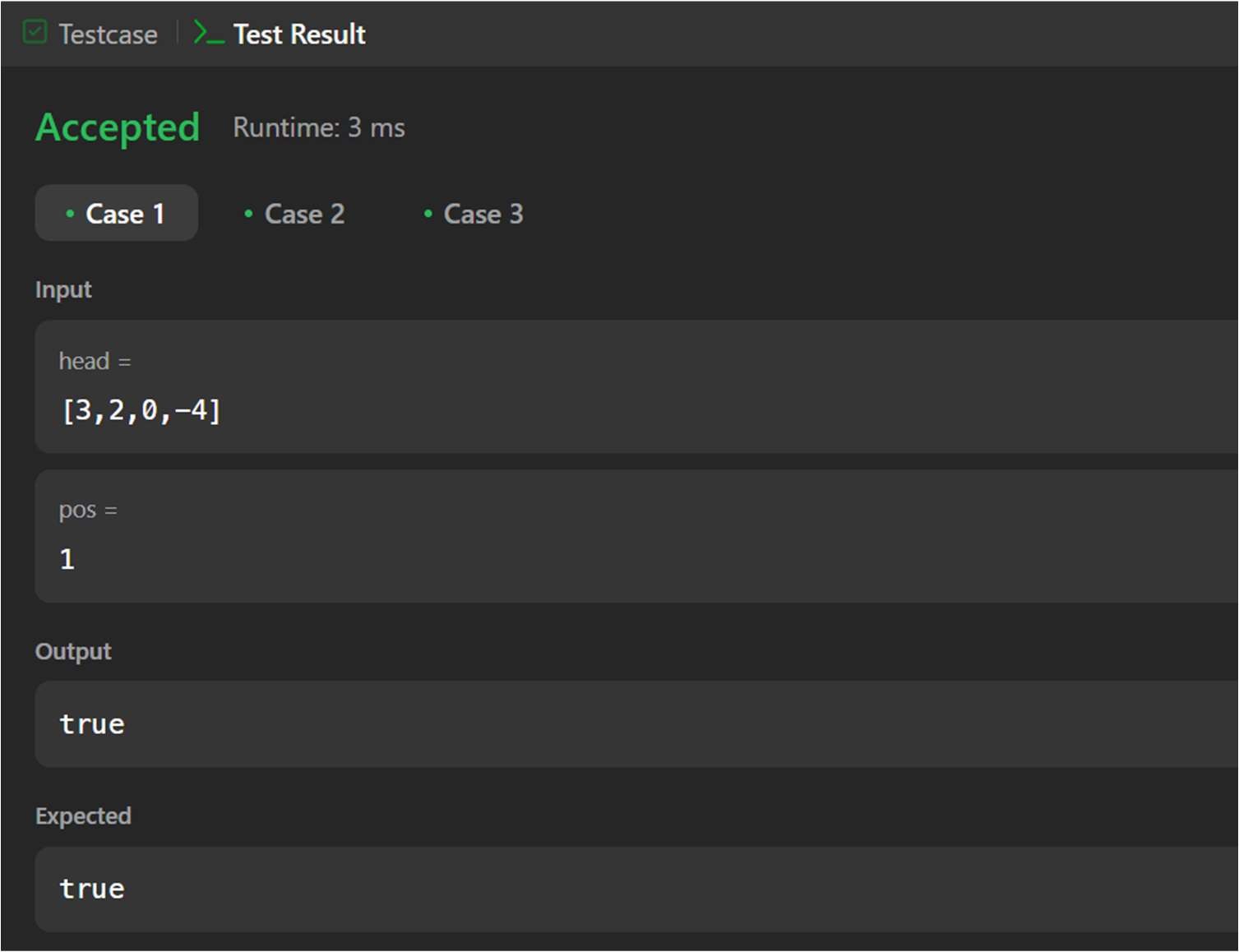
}

return dummy->next;

}

};

# Output:



**Problem 6.** Given the head of a linked list, rotate the list to the right by k places.

# Algorithm:

1. Edge case check:
   * If head is nullptr, the list is empty, so return head.
   * If the list has only one node or k == 0, return head as no rotation is needed.
2. Find the length of the list:
   * Traverse the list to calculate its length and find the last node.
3. Form a circular list:
   * Connect the last node's next pointer to the head to form a circular linked list.
4. Calculate the effective k:
   * Since rotating the list by k positions is the same as rotating it by k % length, compute the effective number of rotations.
5. Find the new head:
   * Traverse the list to find the new tail, which is at position length - k - 1, and set the new head to newTail->next.
6. Break the circular list:
   * Set newTail->next = nullptr to disconnect the circular connection and finalize the list rotation.

6. Return the new head.

# Code:

class Solution { public:

ListNode\* rotateRight(ListNode\* head, int k) {

if (head == nullptr || head->next == nullptr || k == 0) { return head;

}

ListNode\* current = head; int length = 1;

while (current->next != nullptr) { current = current->next; length++;

}

current->next = head; k = k % length;

if (k == 0) {

current->next = nullptr; return head;

}

ListNode\* newTail = head;

for (int i = 1; i < length - k; i++) {

newTail = newTail->next;

}

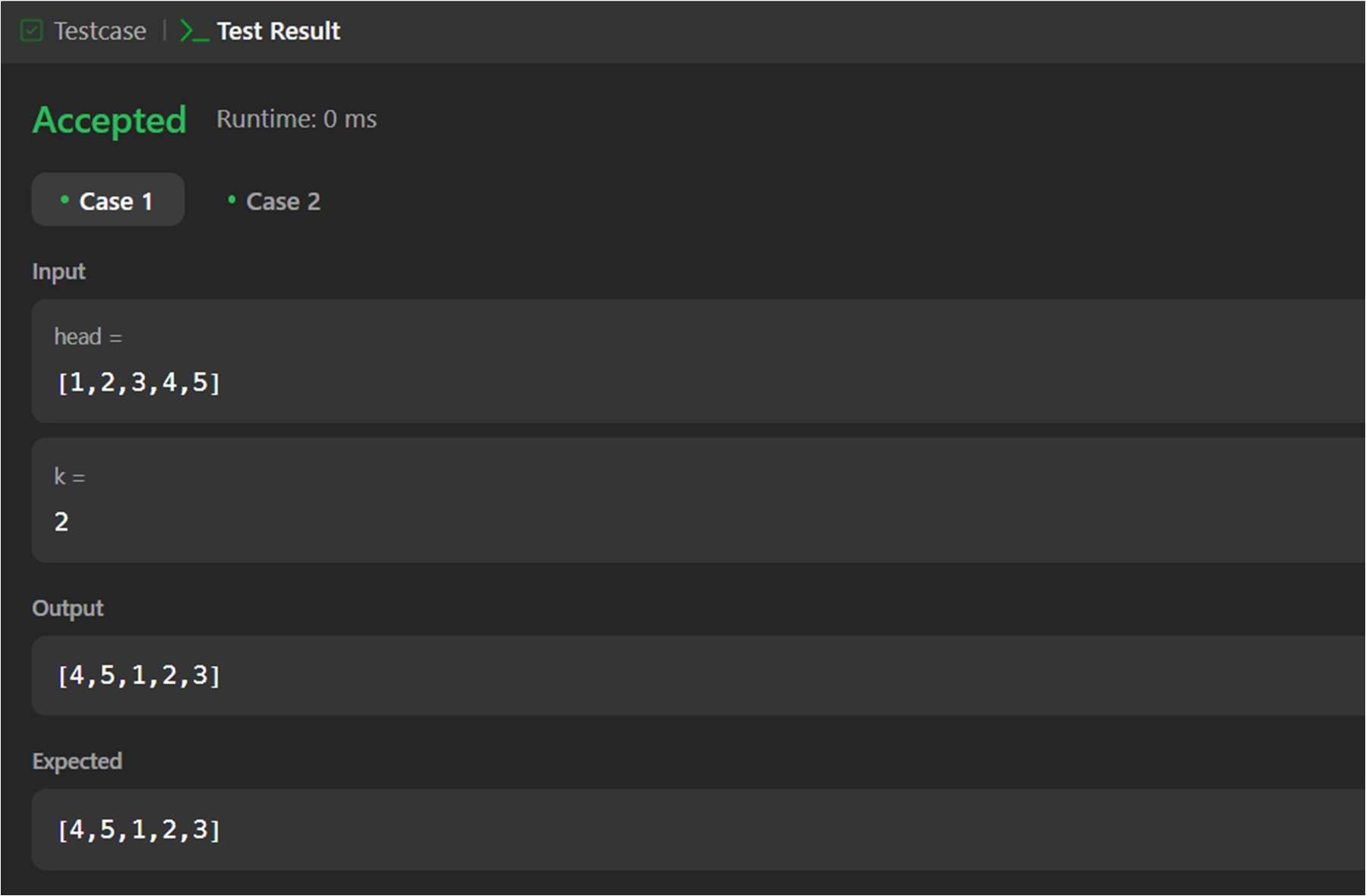
ListNode\* newHead = newTail->next; newTail->next = nullptr;

return newHead;

}

};

# Output:

****