Experiment-3

Student Name:Siddhant Kapoor UID:22BCS10090
Branch:BE-CSE Section/Group:KRG-IOT-1-A
Semester: 6th Date of Performance:17/01/2025
Subject Name: AP LAB-2 Subject Code:22CSP-351

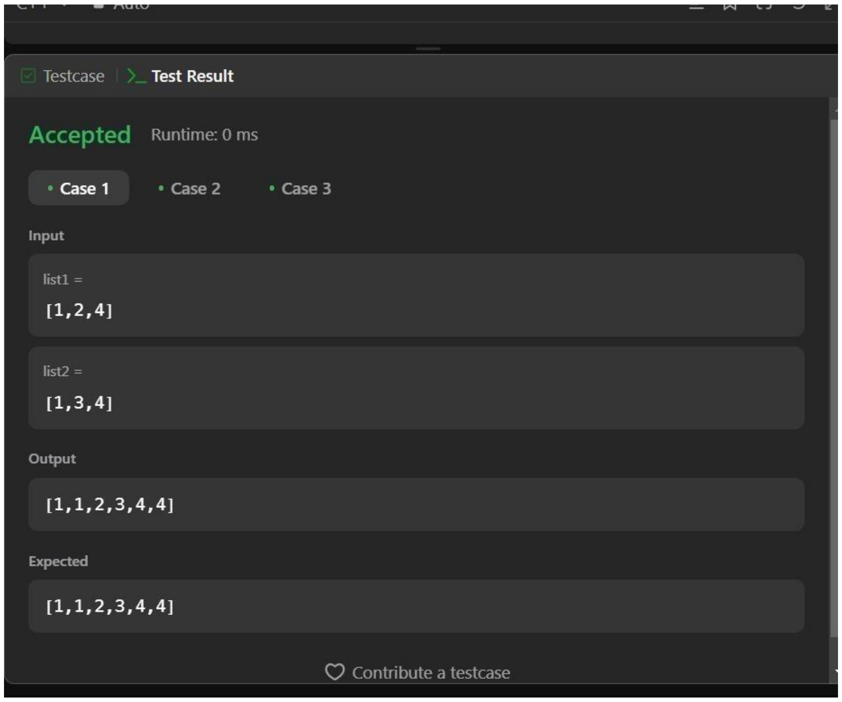
1. **Aim:** You are given the heads of two sorted linked lists list1 and list2. Merge the two lists into one sorted list. The list should be made by splicing together the nodes of the first two lists. Return the head of the merged linked list.

2. **Algorithm:**
- Initialize a dummy node to serve as the starting point for the merged list.
- Create a pointer current to traverse and build the new list.
- Compare the elements from list1 and list2:
- If the value of list1 is smaller, append it to the new list and move list1 pointer to the next node.
- If the value of list2 is smaller, append it to the new list and move list2 pointer to the next node.
- If one list is exhausted, append the remaining nodes of the other list to the merged list.
- Return the next node of the dummy (since it's the start of the merged list)..

3. **Implementation/Code:**

```
class Solution {

public:

   ListNode*
mergeTwoLists(ListNode*
list1, ListNode* list2) {

      ListNode* dummy =
new ListNode(0);

      ListNode* cur =
dummy;



      while (list1 && list2) {

         if (list1->val > list2-
>val) {
            cur->next = list2;
            list2 = list2->next;
         } else {
            cur->next = list1;
            list1 = list1->next;
         }
         cur = cur->next;
      }

      cur->next = list1 ? list1
: list2;

      ListNode* head =
dummy->next;
      delete dummy;
      return head;
   }
};
```

4. **Output:**



Accepted Runtime: 0 ms

Input

list1 =
```
[1,2,4]
```
list2 =
```
[1,3,4]
```
Output
```
[1,1,2,3,4,4]
```
Expected
```
[1,1,2,3,4,4]
```

5. **Learning Outcome:**

- Learn Merge Two Sorted Structures: Understand merging two sorted linked lists into one.
- Pointer Manipulation: Learn to adjust pointers to combine lists while preserving order.
- Iterative vs Recursive Approach: Explore both iterative and recursive solutions for merging.
- Linked List Traversal: Gain proficiency in traversing multiple linked lists simultaneously.
- In-place Modification: Practice merging without using extra space for new lists.

6. **Time Complexity:** O(n + m), where n and m are the lengths of list1 and list2. We are traversing each list once.

7. **Space Complexity:** O(1), since we only use a few extra variables (excluding the space for the merged list)

Problem-2

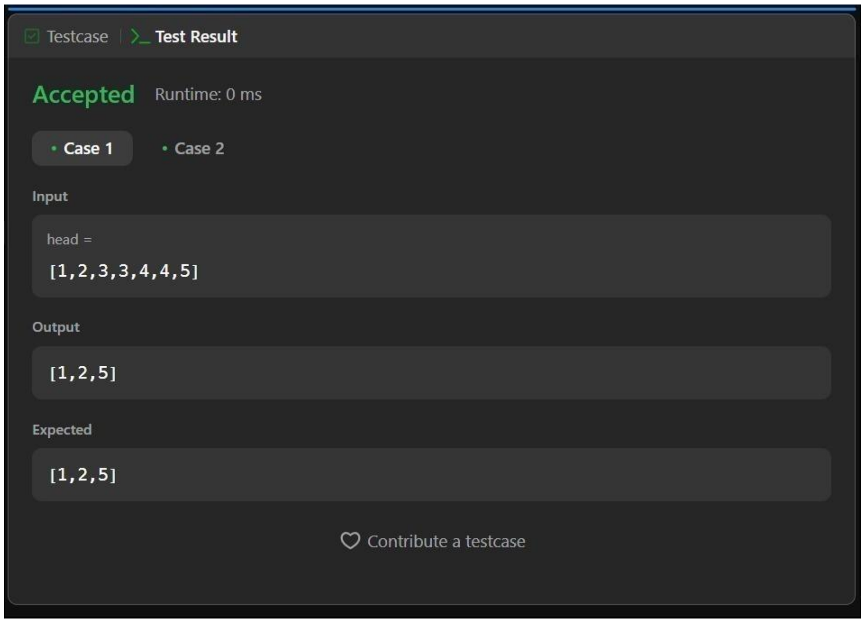
1. **Aim:** Given the head of a sorted linked list, delete all nodes that have duplicate numbers, leaving only distinct numbers from the original list. Return the linked list sorted as well.

2. **Algorithm:**
   - Initialize a dummy node to handle edge cases.
   - Use a pointer prev to point to the dummy node and current to traverse the list.
   - Check for duplicate values by comparing the current node's value with the next node.
   - If duplicates are found, skip all nodes with that value.
   - If no duplicates are found, update the prev pointer to point to the current node.
   - Return the next node of the dummy (which represents the start of the updated list).

3. **Implementation/Code:**

```
        class Solution {

        public:

           ListNode*
        deleteDuplicates(List
        Node* head) {

              ListNode
        dummy(0, head);

              ListNode* prev
        = &dummy;



              while (head) {

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

 while (head->next && head-
>val == head->next->val) {

              head = head->next;

                       }

  prev>next= head->next;

                    }
        else {

           prev = prev->next;

                    }

          head = head->next;

              }

          return dummy.next;

           }

        };
```

4. Output


Testcase | Test Result

Accepted Runtime: 0 ms

Case 1 Case 2

Input

head =

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

Output

```
[1,2,5]
```

Expected

```
[1,2,5]
```

5. Learning Outcome:

- Learn Understand Linked List Traversal: Learn to navigate through a linked list node by node.
- Identify and Handle Duplicates: Develop logic to detect and handle consecutive duplicate nodes in a sorted linked list.
- Manipulate Pointers: Practice adjusting pointers to remove unwanted nodes without breaking the list structure.
- Edge Case Handling: Learn to handle edge cases like an empty list, lists with all duplicates, or no duplicates.

6. **Time Complexity:** O(n), where n is the number of nodes in the list. We traverse each node once.

7. **Space Complexity:** O(1), since we are not using extra space other than the input list.

Experiment-4

Student Name: Siddhant Kapoor **UID:** 22BCS10090
Branch: BE-CSE **Section/Group:** KRG-IOT-1A
Semester: 6th **Date of Performance:**17/01/25
Subject Name: Advanced Programming Lab - 2 **Subject Code:** 22CSP-351

1. Aim:

1. Problem: 1.4.1: Given an unsorted array of elements, the objective is to sort the array in ascending order using the Merge Sort algorithm, which follows the Divide and Conquer paradigm.
2. Problem: 1.4.2: Given a sorted array and a target element, return true if the target element exists in the array and false otherwise.

2. Implementation/Code:

1.)

```
#include <iostream>
using namespace std;

void merge(int arr[], int left, int mid, int right) {
   int n1 = mid - left + 1, n2 = right - mid;
   int L[n1], R[n2];

   for (int i = 0; i < n1; i++) L[i] = arr[left + i];
   for (int i = 0; i < n2; i++) R[i] = arr[mid + 1 + i];

   int i = 0, j = 0, k = left;
   while (i < n1 && j < n2) {
      if (L[i] <= R[j]) arr[k++] = L[i++];
      else arr[k++] = R[j++];
```

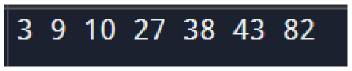
```
        }
        while (i < n1) arr[k++] = L[i++];
        while (j < n2) arr[k++] = R[j++];
    }

    void mergeSort(int arr[], int left, int right) {
        if (left < right) {
            int mid = left + (right - left) / 2;
            mergeSort(arr, left, mid);
            mergeSort(arr, mid + 1, right);
            merge(arr, left, mid, right);
        }
    }

    int main() {
        int arr[] = {38, 27, 43, 3, 9, 82, 10};
        int n = sizeof(arr) / sizeof(arr[0]);

        mergeSort(arr, 0, n - 1);

        for (int i = 0; i < n; i++) cout << arr[i] << " ";
        return 0;
    }
```

2.)

```
#include <iostream>

using namespace std;


int binarySearch(int arr[], int left, int right, int target) {
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == target) return mid;
        if (arr[mid] < target) left = mid + 1;
        else right = mid - 1;
    }
    return -1;
}

int main() {
    int arr[] = {2, 3, 4, 10, 40};
    int n = sizeof(arr) / sizeof(arr[0]);
    int target = 10;

    int result = binarySearch(arr, 0, n - 1, target);
    if (result != -1) cout << "Element found at index " << result;
    else cout << "Element not found";
    return 0;
}
```

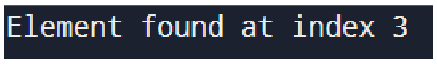
3. Output:

1.



```
3 9 10 27 38 43 82
```

2.



```
Element found at index 3
```

4. Time Complexity:

1. O(n*logn)
2. O(logn)

5. Space Complexity:

1. O(n)
2. O(1)

6. Learning Outcome:

1. Learn how Merge Sort and Binary Search utilize the divide-and-conquer approach for efficient problem-solving.
2. Gain hands-on experience in implementing Merge Sort, understanding how recursive sorting and merging work.
3. Develop skills in implementing Binary Search, analyzing its efficiency in searching sorted arrays.
4. Analyze and compare the time and space complexities of Merge Sort and Binary Search.

Experiment-5

Student Name: Siddhant Kapoor **UID:** 22BCS10090
Branch: BE-CSE **Section/Group:** KRG-IOT-1A
Semester: 6th **Date of Performance:**24/01/25
Subject Name: Advanced Programming Lab - 2 **Subject Code:** 22CSP-351

1. **Aim:**
   Sort Colors: The goal is to sort an array containing 0s, 1s, and 2s without using built-in sorting. This helps in learning efficient ways to organize data manually.
2. **Implementation/Code:**

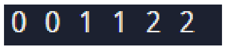
1.)

```
#include <iostream>
#include <vector>
using namespace std;

void sortColors(vector<int>& nums) {
   int low = 0, mid = 0, high = nums.size() - 1;

   while (mid <= high) {
      if (nums[mid] == 0) swap(nums[low++], nums[mid++]);
      else if (nums[mid] == 1) mid++;
      else swap(nums[mid], nums[high--]);
   }
}

int main() {
   vector<int> nums = {2, 0, 2, 1, 1, 0};
   sortColors(nums);

   for (int num : nums) cout << num << " ";
```

```
    return 0;
}
```

3. Output:

1.



```
0 0 1 1 2 2
```

4. Time Complexity:

O(n)

5. Space Complexity:

O(1)

6. Learning Outcome:

1. Learn how to efficiently sort an array containing only three distinct values using a single pass.
2. Gain experience in implementing an in-place sorting algorithm with O(1) space complexity.
3. Understand the two-pointer technique and how it optimizes sorting by reducing unnecessary swaps and iterations.