Experiment: 4(a)

Student Name: Zatch **UID:**
Branch: BE CSE **Section:**
Semester: 5th **Date of Performance:**
Subject Name: AP Lab-1 **Subject Code: 22CSP-314**

1. **Title:** Missing Number

2. **Aim:** You're given the pointer to the head nodes of two linked lists. Compare the data in the nodes of the linked lists to check if they are equal. If all data attributes are equal and the lists are the same length, return 1 . Otherwise, return 0.
   Example:
   linked list 1 = 1 →2→3→NULL
   linked list 2 = 1→2→3→4→NULL
   The two lists have equal data attributes for the first 3 nodes. Llist2 is longer, though, so the lists are not equal. Return 0 write a code in C++11

3. **Objective:** The objective of the code is to compare two linked lists by checking if all corresponding nodes contain equal data and if the lists have the same length. If both conditions are satisfied, the code returns 1; otherwise, it returns 0.

4. **Algorithm:**

1. **Initialize Pointers**: Start with pointers to the head of both linked lists (head1 and head2).
2. **Traverse Both Lists**: While both pointers are not nullptr:
   - Compare the data in the nodes pointed to by head1 and head2.
   - If the data is not equal, return 0 (lists are not equal).
   - Move both pointers to the next node.
3. **Check for Length Mismatch**:After the loop, if one pointer is nullptr but the other is not, return 0 (lists are of different lengths).
4. **Return Success**:If both pointers are nullptr, return 1 (lists are equal in data and length).

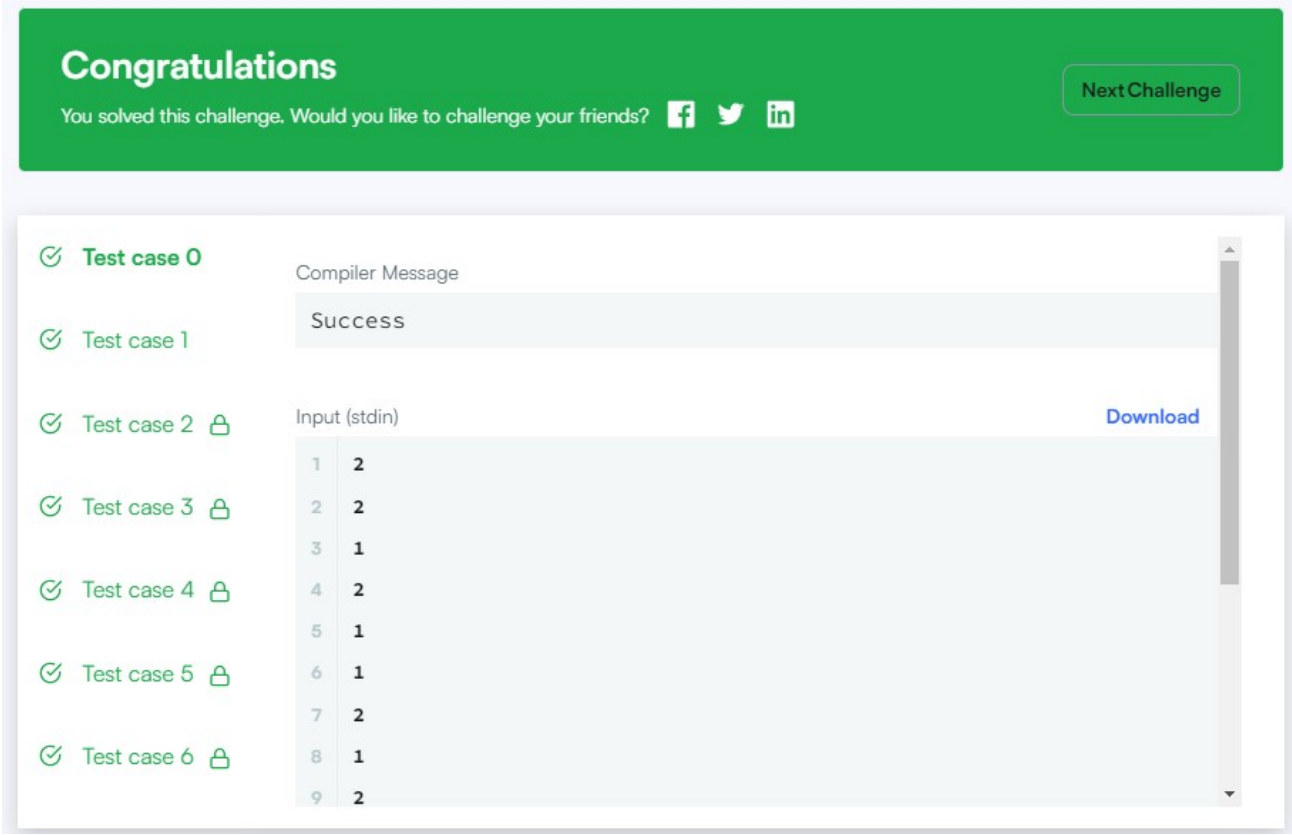
5. Code:

```
#include <bits/stdc++.h>
using namespace std;

class SinglyLinkedListNode {
   public:
      int data;
      SinglyLinkedListNode *next;

      SinglyLinkedListNode(int node_data) {
     this->data = node_data;
         this->next = nullptr;
      }
};

class SinglyLinkedList {
   public:
      SinglyLinkedListNode *head;
      SinglyLinkedListNode *tail;

      SinglyLinkedList() {
         this->head = nullptr;
         this->tail = nullptr;
      }

      void insert_node(int node_data) {
         SinglyLinkedListNode* node =
newSinglyLinkedListNode(node_data);

         if (!this->head) {
            this->head = node;
         } else {
            this->tail->next = node;
         }

         this->tail = node;
      }
};

void print_singly_linked_list(SinglyLinkedListNode* node, string
sep,ofstream& fout) {
```

```
    while (node) {
      fout << node->data;

        node = node->next;

        if (node) {
            fout << sep;
        }
    }
}

void free_singly_linked_list(SinglyLinkedListNode* node) {
    while (node) {
        SinglyLinkedListNode* temp = node;
        node = node->next;

        free(temp);
    }
}

bool compare_lists(SinglyLinkedListNode* head1,
SinglyLinkedListNode*head2) {
while (head1 != nullptr && head2 != nullptr) {
        if (head1->data != head2->data) {
            return false;
        }
        head1 = head1->next;
        head2 = head2->next;
    }

    // If both lists are exhausted and of the same length, they are equal
    return head1 == nullptr && head2 == nullptr;
}

int main()
{
    ofstream fout(getenv("OUTPUT_PATH"));

    int tests;
    cin >> tests;
    cin.ignore(numeric_limits<streamsize>::max(), '\n');
```

```
    for (int tests_itr = 0; tests_itr < tests; tests_itr++) {
        SinglyLinkedList* llist1 = new SinglyLinkedList();

        int llist1_count;
        cin >> llist1_count;
        cin.ignore(numeric_limits<streamsize>::max(), '\n');

        for (int i = 0; i < llist1_count; i++) {
            int llist1_item;
            cin >> llist1_item;
            cin.ignore(numeric_limits<streamsize>::max(), '\n');

            llist1->insert_node(llist1_item);
        }

        SinglyLinkedList* llist2 = new SinglyLinkedList();

        int llist2_count;
        cin >> llist2_count;
        cin.ignore(numeric_limits<streamsize>::max(), '\n');

        for (int i = 0; i < llist2_count; i++) {
            int llist2_item;
            cin >> llist2_item;
            cin.ignore(numeric_limits<streamsize>::max(), '\n');

            llist2->insert_node(llist2_item);
        }

        bool result = compare_lists(llist1->head, llist2->head);

        fout << result << "\n";
    }

    fout.close();

    return 0;
    }
```

6. **Output:**



7. **Learning outcomes:**
   1. **Understanding Linked Lists**: Learn how to traverse and manipulate linked lists using pointers.
   2. **Data Comparison**: Gain experience in comparing data within nodes of linked lists.
   3. **Algorithm Efficiency**: Recognize how to efficiently determine the equality of two linked structures in terms of both data and length.
   4. **Edge Case Handling**: Develop skills in identifying and handling edge cases, such as lists of different lengths or empty lists

8. **Time Complexity:** O(n)
9. **Space Complexity:** O(1)

Experiment: 4(b)

Student Name: Zatch **UID:**
Branch: BE-CSE **Section:**
Semester: 5th **Date of Performance:**
Subject Name: Advanced Lab-1 **Subject Code: 22CSP-314**

1. **Title:** Quick Sort.

2. **Aim**: Choose some pivot element, p, and partition your unsorted array, arr, into three smaller arrays: left, right, and equal, where each element in left <p. each element in right > p. and each element in equal = p.

   Example

   arr = [5,7,4,3,8] In this challenge, the pivot will always be at arr[0], so the pivot is 5.

   arr is divided into left = {4,3). equal = {5}, and right = {7,8}.

   Putting them all together, you get {4, 3, 5, 7, 8). There is a flexible checker that allows the elements of left and right to be in any order. For example. (3, 4, 5, 8, 7} is valid as well.

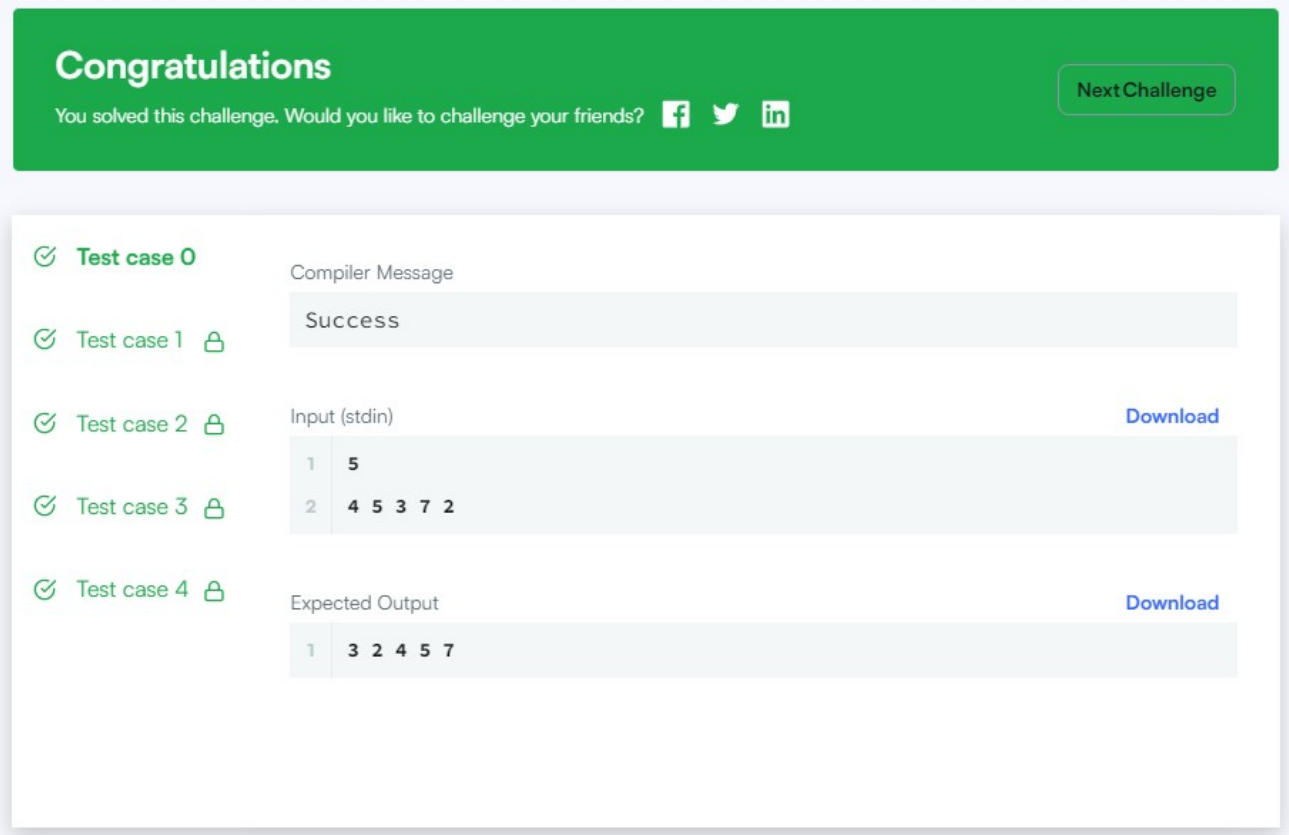
3. **Objective**: Partition an array around a pivot element by dividing it into three sub-arrays: elements less than the pivot, elements equal to the pivot, and elements greater than the pivot, and then combine these sub-arrays into a single sorted array.

4. **Algorithm:**
   1.Initialize Vectors: Create three vectors: left, equal, and right to store elements less than, equal to, and greater than the pivot, respectively.
   2. Iterate Through the Array: Loop through each element in arr:
   If the element is less than the pivot, add it to left.
   If the element is equal to the pivot, add it to equal.
   If the element is greater than the pivot, add it to right.
   3. Combine Results: Concatenate the left, equal, and right vectors into a single result
   4. Return the Result: Return the combined vector as the final partitioned array.

5. **Code:**

```
#include <iostream>
#include <vector>

using namespace std;

vector<int> partition(vector<int>& arr) {
    int pivot = arr[0];
    vector<int> left, equal, right;

    for (int num : arr) {
        if (num < pivot) {
            left.push_back(num);
        } else if (num == pivot) {
            equal.push_back(num);
        } else {
            right.push_back(num);
        }
    }

    // Combine the left, equal, and right vectors
    left.insert(left.end(), equal.begin(), equal.end());
    left.insert(left.end(), right.begin(), right.end());

    return left;
}

int main() {
    vector<int> arr = {5, 7, 4, 3, 8};
    vector<int> result = partition(arr);

    for (int num : result) {
        cout << num << " ";
    }

    return 0;
     }
```

6. **Output:**



7. **Learning outcomes:**

1.Array Partitioning: Understand how to partition an array into sub-arrays based on a pivot element.

2.Vector Operations: Learn to efficiently use vectors in C++11 for dynamic array manipulation.

3.Element Comparison: Develop skills in comparing and categorizing elements relative to a specific value.

4.Algorithm Efficiency: Recognize the importance of linear time complexity in array partitioning tasks.

8. **Time Complexity:** O(n)
9. **Space Complexity:** O(n)