Ahsania Mission University of Science & Technology

Department of Computer Science and Engineering

1st Batch, 2nd Year 2nd Semester, Spring 2025

Lab Report

Course Code: CSE 2202

Course Title: Computer Algorithms Sessional

Experiment No. : 04 ----------

Submission Date : 28-05-2025 -------------------------

Submitted To:

Md. Fahim Faisal
Lecturer,
Department of Computer Science and Engineering (CSE)
Faculty of Engineering, Ahsania Mission University of Science & Technology

Submitted By-

Name : -- Md. Aktaruzzaman Aktar ---

ID No. : -- 1012320005101015----------------------------

1st Batch, 2nd Year 2nd Semester, Spring 2025
Department of Computer Science and Engineering, AMUST.

Task 01: A C++ program that will implement Merge-Sort Algorithm.

Theory:

Merge Sort is an efficient, comparison-based sorting algorithm that uses the divide and conquer strategy. It repeatedly splits the array into two halves until each subarray has one element, which is inherently sorted. The merging phase then carefully combines these sorted subarrays in linear time. This method guarantees a consistent time complexity of O(n log n) for all cases. The algorithm is stable, preserving the original order of equal elements, which is beneficial in many applications. Its recursive structure not only simplifies the conceptual design but also improves clarity in implementation.

Code:

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

void Merge(int arr[] , int p , int q , int r)
{
   int n1 = q - p + 1;
   int n2 = r - q;

   int L[n1] , R[n2];

   for(int i = 0 ; i < n1 ; i++)
   {
      L[i] = arr[p + i];
   }

   for(int j = 0 ; j < n2 ; j++)
   {
      R[j] = arr[q + 1 + j];
   }

   int i = 0 , j = 0 , k = p;

   while(i < n1 && j < n2)
   {
      if(L[i] <= R[j])
      {
```

```
            arr[k] = L[i];
            i++;
        }
        else
        {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while(i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }

    while(j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void Merge_sort(int a[] , int p , int r)
{
    if(p < r)
    {
        int q = floor((p + r) / 2);
        Merge_sort(a , p , q);
        Merge_sort(a, (q + 1) , r);

        Merge(a, p , q , r);

    }
}

int main()
{
    int n;
    cout << "Enter the number of elements of the array: ";
    cin >> n;

    int A[n];
```

```
    cout << "Enter the elements of the array: ";
    for(int i = 0 ; i < n ; i++)
    {
        cin >> A[i];
    }

    Merge_sort(A , 0 , n - 1);

    cout << "Sorted array: ";
    for(int i = 0 ; i < n ; i++)
    {
        cout << A[i] << " ";
    }

    cout << "\n";

    return 0;
}
```

Output:

```
Enter the number of elements of the array: 8
Enter the elements of the array: 5 4 1 8 7 2 6 3
Sorted array: 1 2 3 4 5 6 7 8

Process returned 0 (0x0)   execution time : 16.144 s
Press any key to continue.
```

Task 02: A C++ program that will implement Merge Sort and Count Inversions.

Theory:

Merge Sort can be augmented to count inversions, which are pairs of elements that are out of order. An inversion gives a measure of how unsorted an array is, thus providing extra insight into the data. During the merge phase, whenever an element from the right subarray is positioned before the remaining left subarray elements, an inversion is recorded. This technique leverages the efficient structure of merge sort, maintaining the O(n log n) time complexity. It thereby combines sorting with an analytical evaluation of the array's disorder. Such dual functionality is useful in applications that require both sorted data and a measure of its original disorder.

Code:

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

int inversionCount = 0;

void Merge(int arr[] , int p , int q , int r)
{
  int n1 = q - p + 1;
  int n2 = r - q;

  int L[n1] , R[n2];

  for(int i = 0 ; i < n1 ; i++)
  {
    L[i] = arr[p + i];
  }

  for(int j = 0 ; j < n2 ; j++)
  {
    R[j] = arr[q + 1 + j];
  }

  int i = 0 , j = 0 , k = p;

  while(i < n1 && j < n2)
```

```
    {
        if(L[i] <= R[j])
        {
            arr[k] = L[i];
            i++;
        }
        else
        {
            arr[k] = R[j];
            j++;
            inversionCount += (n1 - i);
        }
        k++;
    }
    while(i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }
    while(j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void Merge_sort(int a[] , int p , int r)
{
    if(p < r)
    {
        int q = floor((p + r) / 2);
        Merge_sort(a , p , q);
        Merge_sort(a, (q + 1) , r);

        Merge(a, p , q , r);
    }
}

int main()
{
    int n;
    cout << "Enter the number of elements of the array: ";
    cin >> n;
```

```
    int A[n];
    cout << "Enter the elements of the array: ";
    for(int i = 0 ; i < n ; i++)
    {
        cin >> A[i];
    }

    inversionCount = 0;
    Merge_sort(A , 0 , n - 1);

    cout << "Sorted array: ";
    for(int i = 0 ; i < n ; i++)
    {
        cout << A[i] << " ";
    }

    cout << "\nNumber of inversions: " << inversionCount << "\n";

    return 0;
}
```

Output:

```
Enter the number of elements of the array: 8
Enter the elements of the array: 5 4 1 8 7 2 6 3
Sorted array: 1 2 3 4 5 6 7 8
Number of inversions: 15

Process returned 0 (0x0)   execution time : 23.257 s
Press any key to continue.
```

Task 03: A C++ program that implement Linear Search Algorithm.

Theory:

Linear search is a straightforward algorithm that scans each element in an array sequentially to find a target value. It does not require the array to be sorted, making it versatile for various datasets. Each element is compared with the search item until a match is found or the end of the array is reached. While its worst-case time complexity is O(n), the algorithm's simplicity makes it ideal for small or unorganized datasets. It serves as a fundamental example of search techniques and helps build the foundation for more advanced methods. Overall, linear search emphasizes clarity and directness in its implementation.

Code:

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

int linear_search(int item, int arr[], int n)
{
   for (int i = 0; i < n; i++)
   {
      if (arr[i] == item)
      {
         cout << "Item found at index " << i << "\n";
         return i;
      }
   }
   cout << "Item not found" << "\n";
   return -1;
}

int main()
{
   int n;
   cin >> n;

   int A[n];
   for(int i = 0 ; i < n ; i++)
   {
      cin >> A[i];
   }
```

```
    int item;
    cin >> item;

    linear_search(item , A , n);

    return 0;
}
```

Output:

```
10
2 5 1 9 3 0 2 4 6 8
6
Item found at index 8

Process returned 0 (0x0)   execution time : 49.481 s
Press any key to continue.
```

Task 04: A C++ program that will Balance Heights (Seating Arrangement) using Merge Sort.

Theory:

This task involves ensuring Sam is seated in the middle with an equal number of friends on both sides by balancing their heights. To achieve this, merge sort is used to arrange the friends' heights in ascending order efficiently. Once sorted, the number of friends shorter and taller than Sam is determined by comparing their heights with Sam's. The cost is calculated as the absolute difference between these two groups, representing the minimal operations needed for balance. Merge sort plays a crucial role by providing an O(n log n) time complexity solution for the sorting step, making the process computationally feasible for large datasets. This method highlights how sorting can simplify seemingly complex problems like balanced arrangement.

Code:

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

void Merge(int arr[] , int p , int q , int r)
{
   int n1 = q - p + 1;
   int n2 = r - q;

   int L[n1] , R[n2];

   for(int i = 0 ; i < n1 ; i++)
   {
      L[i] = arr[p + i];
   }
   for(int j = 0 ; j < n2 ; j++)
   {
      R[j] = arr[q + 1 + j];
   }

   int i = 0 , j = 0 , k = p;

   while(i < n1 && j < n2)
   {
      if(L[i] <= R[j])
```

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

void Merge_sort(int a[] , int p , int r)
{
    if(p < r)
    {
        int q = floor((p + r) / 2);
        Merge_sort(a , p , q);
        Merge_sort(a, (q + 1) , r);

        Merge(a, p , q , r);
    }
}

int main()
{
    int test;
    cin >> test;

    while(test--)
    {
        int n , sam_h;
```

```
        cin >> n >> sam_h;

        int heights[n];
        for(int j = 0 ; j < n ; j++)
        {
           cin >> heights[j];
        }
        Merge_sort(heights , 0 , n - 1);

        int L = 0;
        for(int k = 0 ; k < n ; k++)
        {
           if(heights[k] < sam_h)
           {
              L++;
           }
        }

        int R = n - L;
        int cost = abs(L - R);
        cout << "Cost: " << cost << "\n\n";

    }

    return 0;
}
```

Output:

```
2
3 2
4 3 1
Cost: 1

1 5
6
Cost: 1


Process returned 0 (0x0)   execution time : 31.956 s
Press any key to continue.
```

Task 05: A C++ program that will efficiently sort a Linked List with Merge Sort.

Theory:

Merge sort is an ideal choice for linked list sorting due to its ability to efficiently break down the list into smaller parts. In this approach, the list is divided into two halves using the fast and slow pointer technique, ensuring balanced splits. Each sublist is recursively sorted and then merged back together by comparing node values rather than array indices. The merge process uses a dummy node for simplified pointer management without requiring extra array space. This method maintains the O(n log n) time complexity, which is optimal for sorting. Though the recursion introduces O(log n) auxiliary space, it avoids unnecessary overhead associated with other sorting techniques.

Code:

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

// Definition for singly-linked list.
class ListNode
{
public:
    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 || !head->next)
        {
            return head;
        }
```

```
        // Find the middle of the list using the slow and fast pointers approach
        ListNode* slow = head;
        ListNode* fast = head->next;

        while(fast && fast->next)
        {
            slow = slow->next;
            fast = fast->next->next;
        }
        ListNode* mid = slow->next;
        slow->next = nullptr;

        // Recursively sort each half
        ListNode* left = sortList(head);
        ListNode* right = sortList(mid);

        return merge(left, right);
    }

    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;
        }
        tail->next = l1 ? l1 : l2;

        return dummy.next;
    }
};

void printList(ListNode* head)
{
```

```
    while(head)
    {
        cout << head->val << " ";
        head = head->next;
    }
    cout << "\n";
}

int main()
{
    ListNode* head = new ListNode(-1);
    head->next = new ListNode(5);
    head->next->next = new ListNode(3);
    head->next->next->next = new ListNode(4);
    head->next->next->next->next = new ListNode(0);

    cout << "Original list: ";
    printList(head);

    Solution solution;
    ListNode* sortedHead = solution.sortList(head);

    cout << "Sorted list: ";
    printList(sortedHead);


    return 0;
}
```

Output:

```
Original list: -1 5 3 4 0
Sorted list: -1 0 3 4 5

Process returned 0 (0x0)   execution time : 0.088 s
Press any key to continue.
```