Laboratory 10

Title of the Laboratory Exercise: Sorting and Searching Algorithms

1. Introduction and Purpose of Experiment

   Sorting provide us with means of organising information to facilitate the retrieval of specific data. Searching methods are designed to take advantage of the organisation of information. By solving these problems students will be able to use sorting algorithms to sort a randomly ordered set of numbers, and search for key element.

2. Aim and Objectives

   Aim

   - To develop programs for sorting algorithms

   Objectives

   At the end of this lab, the student will be able to

   - Create C programs using sorting algorithms such as **insertion sort**
   - Create C programs using sorting algorithms such as **merge sort**
   - Create C program for **linear search**
   - Create C program for **binary search**

3. Experimental Procedure

   i. Analyse the problem statement
   ii. Design an algorithm for the given problem statement and develop a flowchart/pseudo-code
   iii. Implement the algorithm in C language
   iv. Compile the C program
   v. Test the implemented program
   vi. Document the Results
   vii. Analyse and discuss the outcomes of your experiment

4. Calculations/Computations/Algorithms

1. Linear Search

```
Step 1 : Start
Step 2 : Start from the leftmost element of arr[] and one by one compare x
with each element of arr[]
Step 2.1 : If x matches with an element, return the index.
Step 2.2 : If x doesn’t match with any of elements, return -1.
Step 3: Stop
```

2. Binary Search

```
Step 1 : Start
Step 2 : Compare x with the middle element.
Step 2.1 : If x matches with middle element, we return the mid index.
Step 2.2 : Else If x is greater than the mid element, then x can only lie in
right half subarray after the mid element. So we recur for right half.
Step 2.3 : Else (x is smaller) recur for the left half.
Step 3: Stop
```

3. Merge Sort

```
Step 1 : Start
Step 2 : If r > l
Step 2.1 : Find the middle point to divide the array into two halves:
             middle m = (l+r)/2
Step 2.2 : Call mergeSort for first half:
             Call mergeSort(arr, l, m)
Step 2.3 : Call mergeSort for second half:
             Call mergeSort(arr, m+1, r)
Step 2.4 : Merge the two halves sorted in step 2 and 3:
             Call merge(arr, l, m, r)
Step 3: Stop
```

4. Insertion Sort

```
Step 1 : Start
// Sort an arr[] of size n
insertionSort(arr, n)
Step 2 : Loop from i = 1 to n-1.
```

Step 3: Pick element arr[i] and insert it into sorted sequence arr[0…i-1]

Step 4: Stop

```c
#include <stdio.h>

#define MAX 100

int list[MAX] = {1, 2, 3, 4, 5}, n = 5;

void menu();
void load_data();
void linear_search();
void binary_search();
int bsearch(int [], int, int, int);
void merge_sort(int [], int, int);
void merge(int [], int, int, int);
void insertion_sort(int [], int);

void print_array(int[], int);

int main() {
    menu();
    return 0;
}

void menu() {
    while (1) {
        int choice;
        printf("\nSearching and Sorting"
               "\n1.\tEnter List Data"
               "\n2.\tLinear Search"
               "\n3.\tBinary Search"
               "\n4.\tMerge Sort"
               "\n5.\tInsertion Sort"
               "\n6.\tExit"
               "\nEnter your choice : ");
        scanf("%d", &choice);

        switch (choice) {
            case 1 :
                load_data();
                break;
            case 2 :
                linear_search();
                break;
            case 3 :
                binary_search();
                break;
            case 4 :
                printf("\nBefore Sorting : \n");
                print_array(list, n);
                merge_sort(list, 0, n-1);
                printf("\nAfter Sorting : \n");
                print_array(list, n);
                break;
            case 5 :
                printf("\nBefore Sorting : \n");
                print_array(list, n);
                insertion_sort(list, n);
                printf("\nAfter Sorting : \n");
                print_array(list, n);
                break;
            case 6 :
                return;
            default :
                printf("Wrong Choice !\n");
        }
    }
}

void load_data() {
    printf("\nEnter the number of elements : ");
    scanf("%d", &n);

    printf("\nEnter %d Elements : ", n);
    for (int i = 0 ; i < n ; i++) {
        scanf("%d", &list[i]);
    }
}

void linear_search() {
    int to_search;
    printf("\nEnter the element to search for : ");
    scanf("%d", &to_search);

    int found = 0;
    for (int i = 0 ; i < n ; i++) {
        if (list[i] == to_search) {
            printf("\nFound %d at %d\n", to_search, i);
            found = 1;
            break;
        }
    }

    if (found == 0) {
        printf("\nElement not found ! \n");
    }
}

void binary_search() {
    int to_search;
    printf("\nEnter the element to search for : ");
    scanf("%d", &to_search);

    int found;
    if ((found = bsearch(list, to_search, 0, n-1)) > -1) {
        printf("\nFound %d at %d\n", to_search, found);
    } else {
        printf("\nElement not found !\n");
    }
}

int bsearch(int arr[], int ele, int start, int end) {
    if (end >= start) {
        int mid = (start + end) / 2;
        if (arr[mid] == ele)
            return mid;

        if (arr[mid] > ele)
            return bsearch(arr, ele, start, mid - 1);

        return bsearch(arr, ele, mid + 1, end);
    }

    return -1;
}

void merge_sort(int arr[], int l, int r) {
    if (l < r) {
        int m = (l + r) / 2;

        merge_sort(arr, l, m);
        merge_sort(arr, m+1, r);

        merge(arr, l, m, r);
    }
}

void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;

    int L[n1], R[n2];

    for (int i = 0 ; i < n1 ; i++)
        L[i] = arr[l + i];
    for (int j = 0 ; j < n2 ; j++)
        R[j] = arr[m + 1 + j];

    int i = 0;
    int j = 0;
    int k = l;

    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++;
```

```c
    }

}

void insertion_sort(int arr[], int n) {
    int key, j;
    for (int i = 0 ; i < n ; i++) {
        key = arr[i];
        j = i - 1;

        while (j >= 0 && arr[j] > key) {
            arr[j+1] = arr[j];
            j = j - 1;
        }

        arr[j+1] = key;
    }
}

void print_array(int arr[], int size) {
    for (int i = 0 ; i < size ; i++)
        printf(" %d", arr[i]);
    printf("\n");
}
```

5. Presentation of Results

```
Searching and Sorting

1.     Enter List Data

2.     Linear Search

3.     Binary Search

4.     Merge Sort

5.     Insertion Sort

6.     Exit

Enter your choice : 1

Enter the number of elements : 10

Enter 10 Elements : 10 9 8 7 6 5 4 3 2 1

10 9 8 7 6 5 4 3 2 1


Searching and Sorting

1.     Enter List Data

2.     Linear Search

3.     Binary Search

4.     Merge Sort
```

```
5.     Insertion Sort
6.     Exit
Enter your choice : 4
Before Sorting :
 10 9 8 7 6 5 4 3 2 1

After Sorting :
 1 2 3 4 5 6 7 8 9 10

Searching and Sorting
1.     Enter List Data
2.     Linear Search
3.     Binary Search
4.     Merge Sort
5.     Insertion Sort
6.     Exit
Enter your choice : 2
Enter the element to search for : 5
Found 5 at 4

Searching and Sorting
1.     Enter List Data
2.     Linear Search
3.     Binary Search
4.     Merge Sort
5.     Insertion Sort
6.     Exit
Enter your choice : 3
Enter the element to search for : 5
Found 5 at 4

Searching and Sorting
1.     Enter List Data
2.     Linear Search
3.     Binary Search
4.     Merge Sort
```

```
5.     Insertion Sort
6.     Exit
Enter your choice : 5
Before Sorting :
 1 2 3 4 5 6 7 8 9 10

After Sorting :
 1 2 3 4 5 6 7 8 9 10

Searching and Sorting
1.     Enter List Data
2.     Linear Search
3.     Binary Search
4.     Merge Sort
5.     Insertion Sort
6.     Exit
Enter your choice : 6
Process finished with exit code 0
```

6. Analysis and Discussions

<u>Linear Search:</u>

Time Complexity : O(n)

<u>Binary Search:</u>

Time Complexity:

The time complexity of Binary Search can be written as

T(n) = T(n/2) + c

The above recurrence can be solved either using Recurrence T ree method or Master method. It falls in case II of Master Method and solution of the recurrence is Theta(Logn).

Auxiliary Space: O(1) in case of iterative implementation. In case of recursive implementation, O(Logn) recursion call stack space.

<u>Merge Sort:</u>

Time Complexity: O(n logn)

Auxiliary Space: O(n)

Algorithmic Paradigm: Divide and Conquer

Sorting In Place: No in a typical implementation

Stable: Yes

<u>Insertion Sort:</u>

Time Complexity: O(n*2)

Auxiliary Space: O(1)

Boundary Cases: Insertion sort takes maximum time to sort if elements are sorted in reverse order. And it takes minimum time (Order of n) when elements are already sorted.

Algorithmic Paradigm: Incremental Approach

Sorting In Place: Yes

Stable: Yes

7. Conclusions

Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.

The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats. Following are some of the examples of sorting in real-life scenarios −

Telephone Directory − The telephone directory stores the telephone numbers of people sorted by their names, so that the names can be searched easily.

Dictionary − The dictionary stores words in an alphabetical order so that searching of any word becomes easy.