LAB 4

Experiment 1

OBJECTIVE: To perform and implement various iterative sorting algorithms:

- Bubble Sort
- Insertion Sort
- Selection Sort

THEORY:

Sorting is a fundamental operation in computer science used to arrange data in a particular order (ascending or descending). This lab demonstrates three common iterative sorting techniques which are simple and easy to understand.

ALGORITHM:

1. Bubble Sort

Bubble Sort compares adjacent elements and swaps them if they are in the wrong order. This process continues until the array is sorted.

Key Points:

- Time Complexity: $O(n^2)$
- Best Case (already sorted): $O(n)$
- Worst Case: $O(n^2)$

2. Insertion Sort

Insertion Sort builds the final sorted array one element at a time. It removes one element from the input data, finds the location it belongs to, and inserts it there.

Key Points:

- Time Complexity: $O(n^2)$
- Best Case (already sorted): $O(n)$
- Worst Case: $O(n^2)$

3. Selection Sort

Selection Sort divides the input into a sorted and unsorted region. It repeatedly selects the smallest (or largest) element from the unsorted region and moves it to the end of the sorted region.

Key Points:

- Time Complexity: $O(n^2)$
- Best Case: $O(n^2)$
- Worst Case: $O(n^2)$

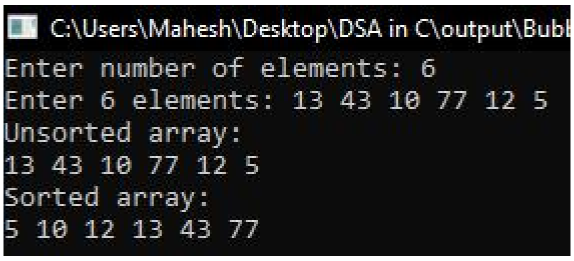
PROGRAMS

1. **Bubble Sort :**

```
#include<stdio.h>
#include<conio.h>

void BubbleSort(int arr[], int n) {
    int i, j, temp;
    for (i = 0; i < n - 1; i++) {
        for (j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
```

```
        printf("%d ", arr[i]);
    printf("\n");
}

int main() {
    int arr[100], n;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    printf("Enter %d elements: ", n);
    for (int i = 0; i < n; i++)
        scanf("%d", &arr[i]);

    printf("\nUnsorted array:\n");
    printArray(arr, n);
    BubbleSort(arr, n);
    printf("\nSorted array:\n");
    printArray(arr, n);
    getch();
    return 0;
}
```

Output:

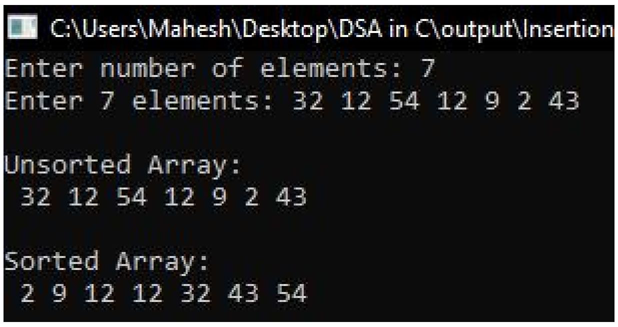


```
C:\Users\Mahesh\Desktop\DSA in C\output\Bubb
Enter number of elements: 6
Enter 6 elements: 13 43 10 77 12 5
Unsorted array:
13 43 10 77 12 5
Sorted array:
5 10 12 13 43 77
```

2. Insertion Sort:

```
#include<stdio.h>
#include<conio.h>

void InsertionSort(int arr[], int n) {
    int temp, i, j;
    for (i = 1; i < n; i++) {
        temp = arr[i];
        j = i - 1;
        while (j >= 0 && arr[j] > temp) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = temp;
    }
}

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

int main() {
    int arr[100], n;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    printf("Enter %d elements: ", n);
    for (int i = 0; i < n; i++)
        scanf("%d", &arr[i]);

    printf("\nUnsorted Array:\n");
    printArray(arr, n);
    InsertionSort(arr, n);
    printf("\nSorted Array:\n");
    printArray(arr, n);
    getch();
    return 0;
}
```

Output:



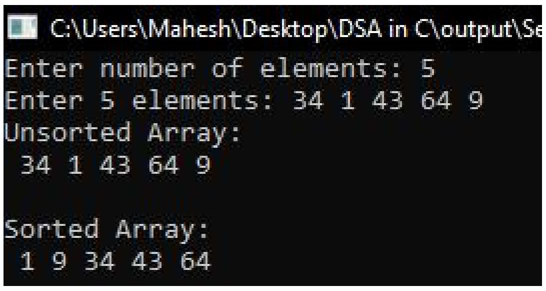
```
C:\Users\Mahesh\Desktop\DSA in C\output\Insertion
Enter number of elements: 7
Enter 7 elements: 32 12 54 12 9 2 43

Unsorted Array:
 32 12 54 12 9 2 43

Sorted Array:
 2 9 12 12 32 43 54
```

3. Selection Sort :

```
#include<stdio.h>
#include<conio.h>

void SelectionSort(int arr[], int n) {
        int i, j, pos, least;
        for (i = 0; i < n - 1; i++) {
                pos = i;
                least = arr[pos];
                for (j = i + 1; j < n; j++) {
                        if (least > arr[j]) {
                                pos = j;
                                least = arr[pos];
                        }
                }
                if (pos != i) {
                        int temp = arr[i];
                        arr[i] = least;
                        arr[pos] = temp;
                }
        }
}
```

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

int main() {
        int arr[100], n;
        printf("Enter number of elements: ");
        scanf("%d", &n);
        printf("Enter %d elements: ", n);
        for (int i = 0; i < n; i++)
                scanf("%d", &arr[i]);

        printf("\nUnsorted Array:\n");
        printArray(arr, n);
        SelectionSort(arr, n);
        printf("\nSorted Array:\n");
        printArray(arr, n);
        getch();
        return 0;
}
```

Output:



```
C:\Users\Mahesh\Desktop\DSA in C\output\Se
Enter number of elements: 5
Enter 5 elements: 34 1 43 64 9
Unsorted Array:
 34 1 43 64 9

Sorted Array:
 1 9 34 43 64
```

CONCLUSION:

The lab helped understand the working of iterative sorting algorithms. All three methods successfully sorted the input data. Among these, Insertion Sort performs better on smaller or partially sorted arrays, while Bubble Sort and Selection Sort are less efficient for larger datasets.

Experiment 2

OBJECTIVE: To perform and implement various recursive sorting algorithms:

- Merge Sort
- Quick Sort

THEORY:

Sorting is a key operation in computer science used to organize data in a particular sequence. Recursive sorting algorithms use the principle of divide and conquer to sort data by breaking the problem into smaller sub-problems and solving them recursively.

ALGORITHM:

1. Merge Sort

Merge Sort divides the array into two halves, recursively sorts them, and then merges the sorted halves.

Steps:

i. Divide the array into two halves.
ii. Recursively sort the sub-arrays.
iii. Merge the sorted halves.

Time Complexity:

- Best Case: O(n log n)
- Average Case: O(n log n)
- Worst Case: O(n log n

2. Quick Sort

Quick Sort picks a pivot element, partitions the array into elements less than and greater than the pivot, and recursively sorts the sub-arrays.

Steps:

i. Select a pivot.
ii. Partition the array around the pivot.
iii. Recursively apply the same steps to the sub-arrays.

Time Complexity:

- Best Case: O(n log n)
- Average Case: O(n log n)
- Worst Case: O(n^2)

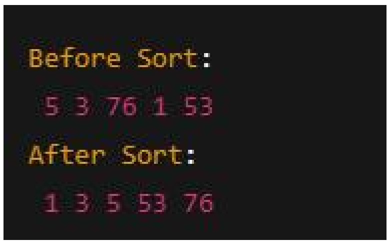
PROGRAMS

1. Merge Sort

```
#include<stdio.h>
#include<conio.h>

void Merge(int arr[], int start, int mid, int end) {
	int i = start, j = mid + 1, k = start;
	int temp[end - start + 1];

	while (i <= mid && j <= end) {
		if (arr[i] < arr[j]) temp[k++] =
arr[i++];
		else temp[k++] = arr[j++];
	}

	while (i <= mid) temp[k++] = arr[i++];
	while (j <= end) temp[k++] = arr[j++];

	for (int i = start; i <= end; i++)
		arr[i] = temp[i];
}

void MergeSort(int arr[], int start, int end) {
	if (start < end) {
		int mid = (start + end) / 2;
		MergeSort(arr, start, mid);
		MergeSort(arr, mid + 1, end);
		Merge(arr, start, mid, end);
	}
}

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

int main() {
	int arr[] = {5, 3, 76, 1, 53};
	int n = sizeof(arr) / sizeof(arr[0]);

	printf("Before Sort:\n");
```

```
    printArray(arr, n);

    MergeSort(arr, 0, n - 1);

    printf("After Sort:\n");
    printArray(arr, n);

    return 0;
}
```

Output

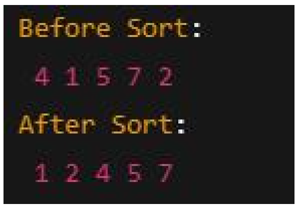


```
Before Sort:
 5 3 76 1 53
After Sort:
 1 3 5 53 76
```

2. Quick Sort

```
#include<stdio.h>
#include<conio.h>

int partition(int arr[], int start, int end){
    int l = start, r = end;
    int p = arr[start];

    while (l < r){
        while(arr[l] <= p) l++;
        while(arr[r] >= p) r--;
        if(l < r){
            int temp = arr[l];
            arr[l] = arr[r];
            arr[r] = temp;
        }
    }
    arr[start] = arr[r];
    arr[r] = p;
    return r;
}

void QuickSort(int arr[], int start, int end) {
    if (start < end) {
        int pi = partition(arr, start, end);
        QuickSort(arr, start, pi - 1);
        QuickSort(arr, pi + 1, end);
    }
}

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

int main() {
    int arr[] = {4, 1, 5, 7, 2};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Before Sort:\n");
    printArray(arr, n);

    QuickSort(arr, 0, n - 1);

    printf("After Sort:\n");
    printArray(arr, n);

    return 0;
}
```

Output:



```
Before Sort:
 4 1 5 7 2
After Sort:
 1 2 4 5 7
```

CONCLUSION:

Recursive sorting algorithms such as Merge Sort and Quick Sort are efficient and widely used. Merge Sort is stable and guarantees O(n log n) time, while Quick Sort is faster in practice but can degrade to O(n^2) in the worst case. This lab enhanced understanding of recursion and divide-and-conquer strategies.

Experiment 3

OBJECTIVE: To perform binary search on a sorted array using C programming.

THEORY:

Binary Search is a searching algorithm used in a sorted array by repeatedly dividing the search interval in half. It works on the principle of divide and conquer.

Features of Binary Search:

1. Fast Search Time: O(log n) time complexity.
2. Works Only on Sorted Arrays.
3. Efficient than Linear Search for large datasets.

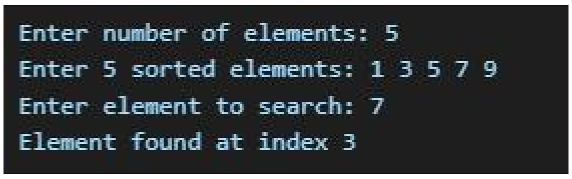
ALGORITHM:

1. Initialize low = 0 and high = size - 1.
2. Repeat while low <= high:
   - Find mid = (low + high) / 2
   - If arr[mid] == target, return mid
   - Else if arr[mid] > target, set high = mid - 1
   - Else set low = mid + 1
3. If not found, return -1

PROGRAMS

```c
#include <stdio.h>
int binarySearch(int arr[], int size, int key) {
    int low = 0, high = size - 1, mid;
    while (low <= high) {
        mid = (low + high) / 2;
        if (arr[mid] == key)
            return mid;
        else if (arr[mid] < key)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return -1;
}
int main() {
    int arr[100], n, key, result;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    printf("Enter %d sorted elements: ", n);
    for (int i = 0; i < n; i++)
        scanf("%d", &arr[i]);
    printf("Enter element to search: ");
    scanf("%d", &key);
    result = binarySearch(arr, n, key);
    if (result != -1)
        printf("Element found at index %d\n", result);
    else
        printf("Element not found\n");

    return 0;
}
```

Output:



```
Enter number of elements: 5
Enter 5 sorted elements: 1 3 5 7 9
Enter element to search: 7
Element found at index 3
```

CONCLUSION:

In this lab, we implemented binary search in C and verified its efficiency in searching sorted arrays. The time complexity was observed to be logarithmic, making it ideal for large datasets.