Nisarg Wath
642403014

Assignment 7

1. Searching Algorithms

HEADER.H

```
typedef struct Array{
    int * data;
    int size;
    int len;

}Array;

void init(Array * a, int size);
void append(Array * a, int data);
void display(Array *arr);
void sort(Array * arr);
void Linear_Search(Array * a, int key);
void Binary_Search(Array * a, int key);
```

LOGIC.C

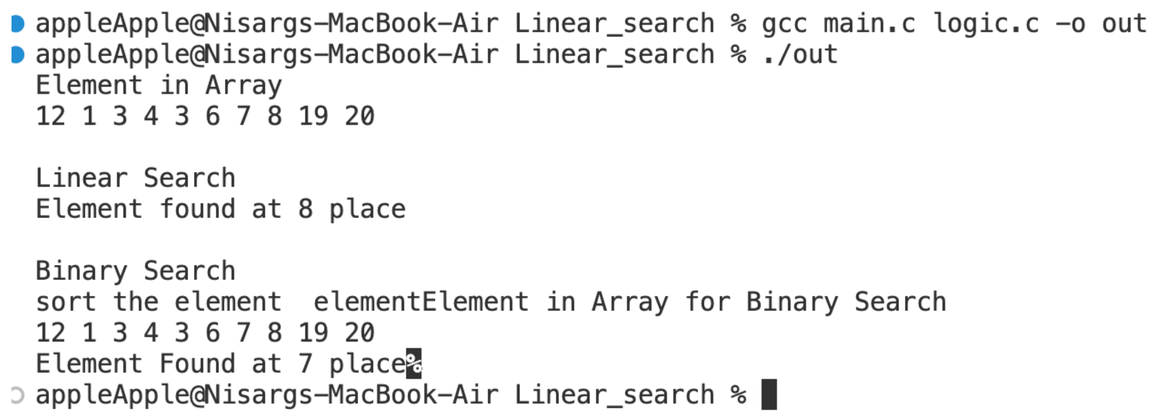
```
#include<stdio.h>
#include<stdlib.h>
#include "header.h"

void init(Array * a, int size){
    a->data = (int *)malloc(sizeof(int) * size);
    a->size = size;
    a->len = 0;
}
void append(Array * a, int data){
    if(a->len > a->size)
    return;

    a->data[a->len++] = data;
}

void Linear_Search(Array * a, int key){

    for(int i = 0;i<a->len;i++){
        if(a->data[i] == key){
            printf("Element found at %d place\n", i);
```

```c
        }
    }
}

void Binary_Search(Array *a, int key) {
    int low = 0;
    int high = a->len - 1;

    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (a->data[mid] == key) {
            printf("Element Found at %d place", mid);
            return;
        }
        if (a->data[mid] < key) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    printf("Element not found\n");
}

void sort(Array * arr){
    for(int i  = 0;i<arr->len-1;i++){
        for(int j = 0;j<arr->len-i-1;j++){
            if(arr->data[j]>arr->data[j+1]){
                int temp = arr->data[j+1];
                arr->data[j+1] =arr->data[j];
                arr->data[j] = temp;
            }
        }
    }
    return;
}


void display(Array *a){
    for(int i = 0;i<a->len;i++){
        printf("%d ",a->data[i]);
    }
    printf("\n");
}

}
```
Main.c

```c
#include<stdio.h>
#include<stdlib.h>
#include "header.h"
```

```c
int main(){
    Array a;
    init(&a,50);
    int n[10] = {12,1,3,4,3,6,7,8,19,20};
    for(int i = 0;i<10;i++){
        append(&a, n[i]);
    }
    printf("Element in Array\n");
    display(&a);
    printf("\nLinear Search\n");
    Linear_Search(&a, 19);
    printf("\nBinary Search\n");
     printf("sort the element  elementElement in Array for Binary
Search\n");
    display(&a);
    sort(&a);

    Binary_Search(&a,12);
}
```

OUTPUT



```
appleApple@Nisargs-MacBook-Air Linear_search % gcc main.c logic.c -o out
appleApple@Nisargs-MacBook-Air Linear_search % ./out
Element in Array
12 1 3 4 3 6 7 8 19 20

Linear Search
Element found at 8 place

Binary Search
sort the element  elementElement in Array for Binary Search
12 1 3 4 3 6 7 8 19 20
Element Found at 7 place%
appleApple@Nisargs-MacBook-Air Linear_search %
```

Time Complexity Analysis of Linear and Binary Searches

Linear Search
- **Description:** Checks each element sequentially in an unsorted or sorted list.
- **Time Complexity:**
- **Best Case:** O(1) (key is first element)
- **Average & Worst Case:** O(n) (key in middle/end or not present)
- **Strengths:** Works on unsorted lists; simple to implement.
- **Weaknesses:** Slow for large datasets.

2. Binary Search
- **Description:** Efficiently finds a key in a sorted list by halving the search space.
- **Time Complexity:**
- **Best Case:** O(1) (key at middle)
- **Average & Worst Case:** O(log n) (each step halves the search space)
- **Strengths:** Fast for large, sorted datasets; logarithmic time complexity.
- **Weaknesses:** Requires a sorted list, limiting use with dynamic or unsorted data.

3. Comparison Summary
- **Linear Search:** Good for small/unsorted datasets but slow (O(n)) as n grows.
- **Binary Search:** Ideal for large, sorted datasets, as it performs in O(log n) time.
- **Recommendation:** Use linear search for unsorted or small lists; binary search is best for large, sorted lists due to its efficiency.

Linear Search	**Binary Search**
In linear search input data need not to be in sorted.	In binary search input data need to be in sorted order.
It is also called sequential search.	It is also called half-interval search.
The time complexity of linear search **O(n)**.	The time complexity of binary search **O(log n)**.
Multidimensional array can be used.	Only single dimensional array is used.
Linear search performs equality comparisons	Binary search performs ordering comparisons
It is less complex.	It is more complex.
It is very slow process.	It is very fast process.

2.Sorting Algorithms

HEADER.H

```c
typedef struct Array{
    int * data;
    int size;
    int len;

}Array;

void init(Array * a, int size);
void append(Array * a, int data);
void display(Array *arr);
void Bubble_Sort(Array * arr);
void Selection_Sort(Array * arr);
void Insertion_Sort(Array * arr);
void swap(int* a, int* b);
int partition(Array * arr, int low, int high);
void quick_sort(Array * a, int low, int high);
void heap_sort(Array * arr);
void heapify(Array * arr, int n, int i);
```

LOGIC.C

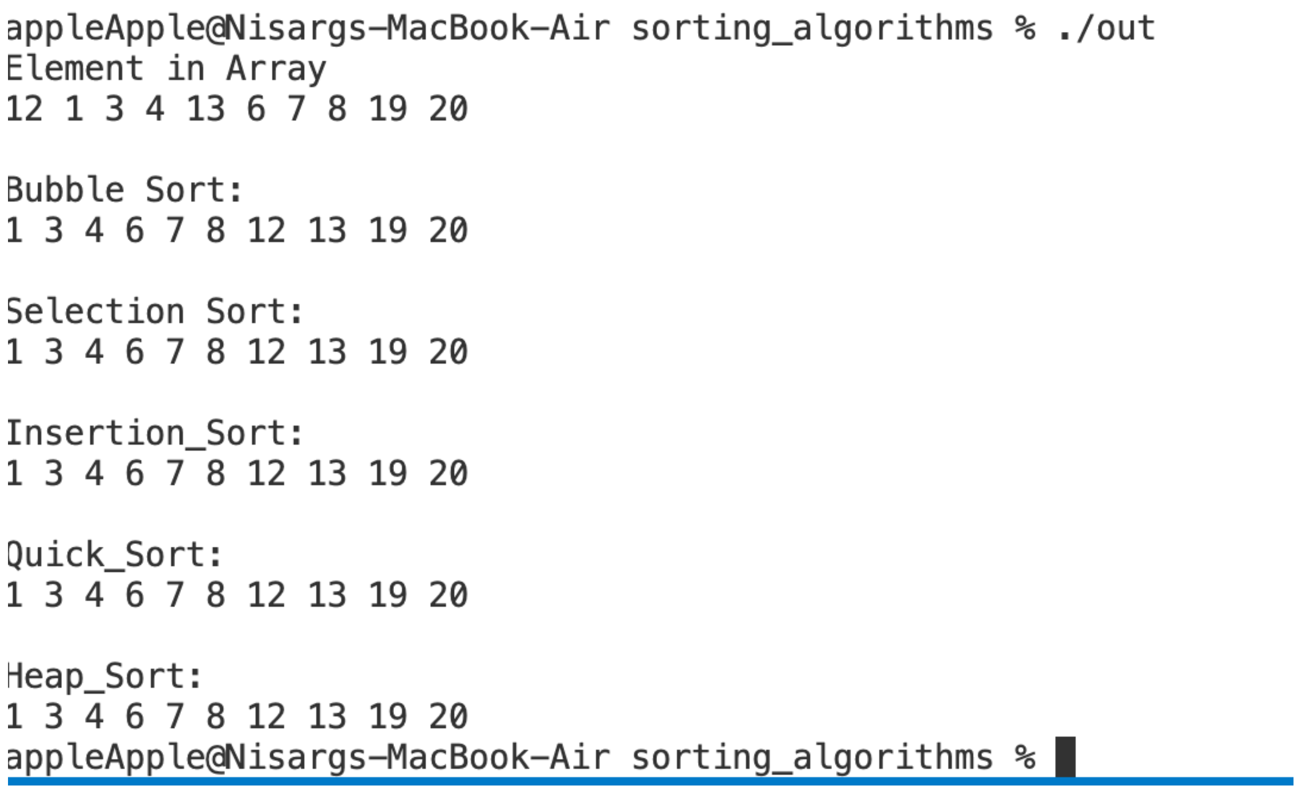
```c
#include<stdio.h>
#include<stdlib.h>
#include "header.h"

void init(Array * a, int size){
    a->data = (int *)malloc(sizeof(int) * size);
    a->size = size;
    a->len = 0;
}
void append(Array * a, int data){
    if(a->len > a->size)
    return;

    a->data[a->len++] = data;
}

void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

```c
void Bubble_Sort(Array * arr){
    for(int i  = 0;i<arr->len-1;i++){
        for(int j = 0;j<arr->len-i-1;j++){
            if(arr->data[j]>arr->data[j+1]){
                int temp = arr->data[j+1];
                arr->data[j+1] =arr->data[j];
                arr->data[j] = temp;
            }
        }
    }
    return;
}

void Selection_Sort(Array * arr){
    for(int i = 0; i<arr->len;i++){
        int key =  i;
        for(int j = i+1;j<arr->len;j++){
            if(arr->data[j]< arr->data[key]){
                key = j;
            }
        }
        int temp = arr->data[i];
        arr->data[i] = arr->data[key];
        arr->data[key] =  temp;
        return;
    }
}


void Insertion_Sort(Array * arr){
    for(int i = 1;i<arr->len;i++){
        int key = arr->data[i];
        int j = i-1;

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


int partition(Array * arr, int low, int high) {
    int pivot = arr->data[high]; // Choosing the last element as 
pivot
    int i = low - 1; // Index of the smaller element

    for (int j = low; j < high; j++) {
        if (arr->data[j] < pivot) {
            i++;
            // Swap arr->data[i] and arr->data[j]
            int temp = arr->data[i];
            arr->data[i] = arr->data[j];
            arr->data[j] = temp;
        }
    }
    // Swap arr->data[i + 1] and arr->data[high] (pivot)
    int temp = arr->data[i + 1];
    arr->data[i + 1] = arr->data[high];
    arr->data[high] = temp;

    return i + 1;
}


void quick_sort(Array * a, int low, int high) {
    if (low < high) {
        int pi = partition(a, low, high);


        quick_sort(a, low, pi - 1);
        quick_sort(a, pi + 1, high);
    }
}






void heapify(Array * arr, int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;


    if (left < n && arr->data[left] > arr->data[largest]) {
        largest = left;
    }


    if (right < n && arr->data[right] > arr->data[largest]) {
        largest = right;
    }


    if (largest != i) {
        int temp = arr->data[i];
        arr->data[i] = arr->data[largest];
        arr->data[largest] = temp;
        heapify(arr, n, largest);
    }
}


void heap_sort(Array * arr) {
    int n = arr->size;

    for (int i = n / 2 - 1; i >= 0; i--) {
        heapify(arr, n, i);
    }

    for (int i = n - 1; i >= 1; i--) {

        int temp = arr->data[0];
        arr->data[0] = arr->data[i];
        arr->data[i] = temp;
        heapify(arr, i, 0);
    }
}






void display(Array *a){
    for(int i = 0;i<a->len;i++){
        printf("%d ",a->data[i]);
    }
    printf("\n");
}
```

Main.c

```c
#include<stdio.h>
#include<stdlib.h>
#include "header.h"


int main(){
    Array a;
    init(&a,50);
    int n[10] = {12,1,3,4,13,6,7,8,19,20};
    for(int i = 0;i<10;i++){
        append(&a, n[i]);
    }
    printf("Element in Array\n");
    display(&a);
    printf("\nBubble Sort:\n");
     Bubble_Sort(&a);
    display(&a);
    printf("\nSelection Sort:\n");
    Selection_Sort(&a);
    display(&a);
    printf("\nInsertion_Sort:\n");
    Insertion_Sort(&a);
    display(&a);
    printf("\nQuick_Sort:\n");
    quick_sort(&a, 0, a.len-1);
    display(&a);
    printf("\nMerge_Sort:\n");
    merge_sort(&a, 0,a.len-1);
    display(&a);


}
```

OUTPUT



```
appleApple@Nisargs-MacBook-Air sorting_algorithms % ./out
Element in Array
12 1 3 4 13 6 7 8 19 20

Bubble Sort:
1 3 4 6 7 8 12 13 19 20

Selection Sort:
1 3 4 6 7 8 12 13 19 20

Insertion_Sort:
1 3 4 6 7 8 12 13 19 20

Quick_Sort:
1 3 4 6 7 8 12 13 19 20

Heap_Sort:
1 3 4 6 7 8 12 13 19 20
appleApple@Nisargs-MacBook-Air sorting_algorithms % 
```

2 Sorting Algorithms Report

Objective:
The goal of this report is to analyse and compare five common sorting algorithms—Bubble Sort, Selection Sort, Insertion Sort, Quick Sort, and Heap Sort—by examining their time complexities, strengths, weaknesses, and performance on various data sets.

1. Bubble Sort

- Overview: Repeatedly compares adjacent elements and swaps them if needed.
- Time Complexity:
- Best: $O(n)$ (already sorted)
- Average: $O(n^2)$
- Worst: $O(n^2)$ (reverse sorted)
- Performance: Efficient for small or nearly sorted datasets, but slow for large datasets due to $O(n^2)$ complexity.

---

2. Selection Sort

- Overview: Selects the smallest element from the unsorted part and swaps it into position.
- Time Complexity:
- Best: $O(n^2)$
- Average: $O(n^2)$
- Worst: $O(n^2)$
- Performance: Suitable for small datasets, but inefficient for larger ones due to consistent $O(n^2)$ complexity.

3. Insertion Sort

- Overview: Builds the sorted array by inserting each element into its correct position.
- Time Complexity:
- Best: $O(n)$ (already sorted)
- Average: $O(n^2)$
- Worst: $O(n^2)$
- Performance: Efficient for small or nearly sorted data, better than Bubble and Selection Sort in these scenarios.

4. Quick Sort

- Overview: A divide-and-conquer algorithm that partitions the array and recursively sorts subarrays.
- Time Complexity:
- Best: O(n log n)
- Average: O(n log n)
- Worst: $O(n^2)$ (poor pivot choice)
- Performance: Fast for large datasets, but can degrade to $O(n^2)$ without careful pivot selection.

5. Heap Sort

- Overview: Uses a binary heap to sort elements by repeatedly extracting the maximum element.
- Time Complexity:
- Best: O(n log n)
- Average: O(n log n)
- Worst: O(n log n)
- Performance: Consistent O(n log n) performance, ideal for memory-constrained environments.

Sorting algorithm	Efficiency	Passes	Sort stability
Bubble sort	0(n2)	n-1	stable
Selection sort	0(n2)	n-1	unstable (can be stable using Linked List)

Insertion sort
Best case
Worst case
	0(n)
	0(n2)
		n-1
		n-1
			stable

Quick sort
Best case
Worst case
	0(n log n)
	0(n2)
		log n
		n-1
			unstable

Conclusion

- Quick Sort is the fastest for large datasets on average, but can degrade to $O(n^2)$ if the pivot is chosen poorly.
- Heap Sort offers consistent O(n log n) performance and is ideal for memory-constrained environments.
- Insertion Sort is efficient for small or nearly sorted data.
- Bubble Sort and Selection Sort are less efficient for large datasets but can be useful for small arrays