Design and Analysis of Algorithms -Sorting Techniques

Name:S.Halitha Begam

Roll-No:CH.SC.U4CSE24157

1. Write a program to implement **Bubble sort**

Code:

```c
#include <stdio.h>

void bubbleSort(int a[], int n) {
    for(int i = 0; i < n - 1; i++) {
        int swapped = 0;
        for(int j = 0; j < n - i - 1; j++) {
            if(a[j] > a[j + 1]) {
                int t = a[j];
                a[j] = a[j + 1];
                a[j + 1] = t;
                swapped = 1;
            }
        }
        if(swapped == 0) {
            break;
        }
    }
}

int main() {
    int n;
    printf("Enter number of elements: ");
    scanf("%d",&n);

    int a[n];
    printf("Enter %d elements: ",n);
    for(int i = 0; i < n; i++) {
        scanf("%d",&a[i]);
    }

    bubbleSort(a,n);

    printf("Sorted array: ");
    for(int i = 0; i < n; i++) {
        printf("%d ",a[i]);
    }

    return 0;
}
```

Output:

```
root@amma53:/home/amma/Documents/24157/daa# gcc -o bubble bubble.c
root@amma53:/home/amma/Documents/24157/daa# ./bubble
Enter number of elements: 5
Enter 5 elements: 6
9
3
8
5
Sorted array: 3 5 6 8 9
root@amma53:/home/amma/Documents/24157/daa#
```

Space Complexity of Bubble sort: O(1) –uses only a few constant variables.

Time complexity of Bubble sort: $O(n^2)$ –double nested loops compare elements repeatedly in worst case.

2. Write a program to implement **Insertion sort**

Code:

```c
#include <stdio.h>

void insertionSort(int a[], int n) {
    for(int i = 1; i < n; i++) {
        int key = a[i];
        int j = i - 1;
        while(j >= 0 && a[j] > key) {
            a[j + 1] = a[j];
            j--;
        }
        a[j + 1] = key;
    }
}

int main() {
    int n;
    printf("Enter number of elements: ");
    scanf("%d",&n);

    int a[n];
    printf("Enter %d elements: ",n);
    for(int i = 0; i < n; i++) {
        scanf("%d",&a[i]);
    }

    insertionSort(a,n);

    printf("Sorted array: ");
    for(int i = 0; i < n; i++) {
        printf("%d ",a[i]);
    }

    return 0;
}
```

Output:

```
root@amma53:/home/amma/Documents/24157/daa# gcc -o insertion insertion.c
root@amma53:/home/amma/Documents/24157/daa# ./insertion
Enter number of elements: 5
Enter 5 elements: 6
9
3
8
5
Sorted array: 3 5 6 8 9
root@amma53:/home/amma/Documents/24157/daa#
```

Space Complexity of Insertion sort : O(1) – No extra array is used, only one temporary variable.

Time complexity of insertion sort: $O(n^2)$ – Each element is compared and shifted with all previous elements using a double nested loop.

3. Write a program to implement **Selection sort**

Code:

```c
#include <stdio.h>

void selectionSort(int a[], int n) {
    for(int i = 0; i < n - 1; i++) {
        int min = i;
        for(int j = i + 1; j < n; j++) {
            if(a[j] < a[min]) {
                min = j;
            }
        }
        int t = a[i];
        a[i] = a[min];
        a[min] = t;
    }
}

int main() {
    int n;
    printf("Enter number of elements: ");
    scanf("%d",&n);

    int a[n];
    printf("Enter %d elements: ",n);
    for(int i = 0; i < n; i++) {
        scanf("%d",&a[i]);
    }

    selectionSort(a,n);

    printf("Sorted array: ");
    for(int i = 0; i < n; i++) {
        printf("%d ",a[i]);
    }

    return 0;
}
```

Output:

```
root@amma53:/home/amma/Documents/24157/daa# gcc -o selection selection.c
root@amma53:/home/amma/Documents/24157/daa# ./selection
Enter number of elements: 5
Enter 5 elements: 6
9
3
8
5
Sorted array: 3 5 6 8 9
root@amma53:/home/amma/Documents/24157/daa#
```

Space Complexity of Selection sort: O(1) – Uses only one temporary variable for swapping

Time complexity of Selection sort: $O(n^2)$ – For every position, the minimum element is searched in the remaining array using a double nested loop.

4a. Write a program to implement **Heap sort(max)**

```c
#include <stdio.h>

void heapifyMax(int a[], int n, int i) {
    int largest = i;
    int l = 2 * i + 1;
    int r = 2 * i + 2;

    if(l < n && a[l] > a[largest]) {
        largest = l;
    }
    if(r < n && a[r] > a[largest]) {
        largest = r;
    }

    if(largest != i) {
        int t = a[i];
        a[i] = a[largest];
        a[largest] = t;
        heapifyMax(a,n,largest);
    }
}

void heapSortMax(int a[], int n) {
    for(int i = n / 2 - 1; i >= 0; i--) {
        heapifyMax(a,n,i);
    }
    for(int i = n - 1; i > 0; i--) {
        int t = a[0];
        a[0] = a[i];
        a[i] = t;
        heapifyMax(a,i,0);
    }
}

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

    heapSortMax(a,n);

    printf("Ascending Order: ");
    for(int i = 0; i < n; i++) {
        printf("%d ",a[i]);
    }
    return 0;
}
```

Output:

```
C:\Users\halit\OneDrive\Documents\sem4\daa\daaa worksheet\sorting_techniques>gcc daa_2_4heapmaxinp.c

C:\Users\halit\OneDrive\Documents\sem4\daa\daaa worksheet\sorting_techniques>a
Enter number of elements: 5
Enter 5 elements: 6
9
3
8
5
Ascending Order: 3 5 6 8 9
```

Space Complexity of Max Heap sort: O(logn)) – Extra space is used due to recursive calls.

Time complexity of Max Heap sort: O(nlogn) – Heap is built once and heapify() takes log n time for each element(n elements).

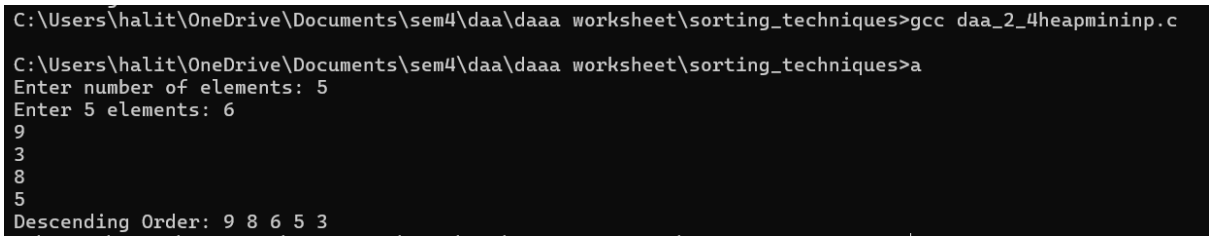
- ➔ Heapify takes O(log n) time because it adjusts an element along the height of the heap where heap is a complete binary tree and its height is logn.

4b. Write a program to implement **Heap sort(min)**

Code:

```c
#include <stdio.h>
void heapifyMin(int a[], int n, int i) {
    int smallest = i;
    int l = 2 * i + 1;
    int r = 2 * i + 2;

    if(l < n && a[l] < a[smallest]) {
        smallest = l;
    }
    if(r < n && a[r] < a[smallest]) {
        smallest = r;
    }

    if(smallest != i) {
        int t = a[i];
        a[i] = a[smallest];
        a[smallest] = t;
        heapifyMin(a,n,smallest);
    }
}

void heapSortMin(int a[], int n) {
    for(int i = n / 2 - 1; i >= 0; i--) {
        heapifyMin(a,n,i);
    }
    for(int i = n - 1; i > 0; i--) {
        int t = a[0];
        a[0] = a[i];
        a[i] = t;
        heapifyMin(a,i,0);
    }
}

int main() {
    int n;
    printf("Enter number of elements: ");
    scanf("%d",&n);

    int a[n];
    printf("Enter %d elements: ",n);
    for(int i = 0; i < n; i++) {
        scanf("%d",&a[i]);
    }
    heapSortMin(a,n);

    printf("Descending Order: ");
    for(int i = 0; i < n; i++) {
        printf("%d ",a[i]);
    }
    return 0;
}
```

Output:



Space Complexity of Min Heap sort: O(logn) – Extra space is used due to recursive calls.

Time complexity of Min Heap sort: O(nlogn) – Heap is built once and heapify() takes log n time for each element.

5. Write a program to implement **Bucket sort**

Code:

```c
#include <stdio.h>
#define N 7

void bucketSort(float a[]) {
    float bucket[N][N];
    int count[N] = {0};

    for(int i = 0; i < N; i++) {
        int b = a[i] * N;
        bucket[b][count[b]++] = a[i];
    }

    for(int i = 0; i < N; i++) {
        for(int j = 1; j < count[i]; j++) {
            float key = bucket[i][j];
            int k = j - 1;
            while(k >= 0 && bucket[i][k] > key) {
                bucket[i][k + 1] = bucket[i][k];
                k--;
            }
            bucket[i][k + 1] = key;
        }
    }

    int index = 0;
    for(int i = 0; i < N; i++) {
        for(int j = 0; j < count[i]; j++) {
            a[index++] = bucket[i][j];
        }
    }
}

int main() {
    float a[N];
    printf("Enter %d fractional values between 0 and 1: ",N);

    for(int i = 0; i < N; i++) {
        scanf("%f",&a[i]);
    }

    bucketSort(a);

    printf("Sorted array: ");
    for(int i = 0; i < N; i++) {
        printf("%.2f ",a[i]);
    }

    return 0;
}
```

Output:

```
root@amma53:/home/amma/Documents/24157/daa# gcc -o bucket bucket.c
root@amma53:/home/amma/Documents/24157/daa# ./bucket
Enter 7 fractional values between 0 and 1: 0.01
0.99
0.67
0.23
0.58
0.87
0.04
Sorted array: 0.01 0.04 0.23 0.58 0.67 0.87 0.99
```

Space Complexity of Bucket sort: $O(N^2)$ **–** Extra buckets and count arrays are used.

Time complexity of Bucket sort: $O(N^2)$ – Worst case happens when all the elements fall into one bucket.

6. Write a program to implement **bfs**

Code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define MAX 5

struct Vertex {
    char label;
    bool visited;
};

// Queue variables
int queue[MAX];
int rear = -1;
int front = 0;
int queueItemCount = 0;

// Graph variables
struct Vertex* lstVertices[MAX];
int adjMatrix[MAX][MAX];
int vertexCount = 0;

// Queue functions
void insert(int data) {
    queue[++rear] = data;
    queueItemCount++;
}

int removeData() {
    queueItemCount--;
    return queue[front++];
}

bool isQueueEmpty() {
    return queueItemCount == 0;
}

// Graph functions
void addVertex(char label) {
    struct Vertex* vertex = (struct Vertex*) malloc(sizeof(struct Vertex));
    vertex->label = label;
    vertex->visited = false;
    lstVertices[vertexCount++] = vertex;
```

```
}

void addEdge(int start, int end) {
    adjMatrix[start][end] = 1;
    adjMatrix[end][start] = 1;
}

void displayVertex(int vertexIndex) {
    printf("%c ", lstVertices[vertexIndex]->label);
}

int getAdjUnvisitedVertex(int vertexIndex) {
    for (int i = 0; i < vertexCount; i++) {
        if (adjMatrix[vertexIndex][i] == 1 && lstVertices[i]->visited ==
false) {
            return i;
        }
    }
    return -1;
}

void breadthFirstSearch() {
    lstVertices[0]->visited = true;
    displayVertex(0);
    insert(0);

    while (!isQueueEmpty()) {
        int tempVertex = removeData();
        int unvisitedVertex;
        while ((unvisitedVertex = getAdjUnvisitedVertex(tempVertex)) != -1) {
            lstVertices[unvisitedVertex]->visited = true;
            displayVertex(unvisitedVertex);
            insert(unvisitedVertex);
        }
    }

    // Reset visited flags for next traversal
    for (int i = 0; i < vertexCount; i++) {
        lstVertices[i]->visited = false;
    }
}

int main() {
    // Initialize the adjacency matrix
    for (int i = 0; i < MAX; i++) {
        for (int j = 0; j < MAX; j++) {
            adjMatrix[i][j] = 0;
        }
```

```
    }

    addVertex('S'); // 0
    addVertex('A'); // 1
    addVertex('B'); // 2
    addVertex('C'); // 3
    addVertex('D'); // 4

    addEdge(0, 1); // S - A
    addEdge(0, 2); // S - B
    addEdge(0, 3); // S - C
    addEdge(1, 4); // A - D
    addEdge(2, 4); // B - D
    addEdge(3, 4); // C - D

    printf("Breadth First Search: ");
    breadthFirstSearch();

    return 0;
}
```

Output:

```
C:\Documents\sem4\daa\daa>gcc 44.bfs.c

C:\Documents\sem4\daa\daa>a
Breadth First Search: S A B C D
```

Space Complexity: $O(v^2)$ – Adjacency matrix and queue needs extra memory.

Time complexity: $O(v^2)$ – Each vertex checks all other vertices in the matrix.

7. Write a program to implement **dfs**

Code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define MAX 5

struct Vertex {
    char label;
    bool visited;
};

// Stack variables
int stack[MAX];
int top = -1;

// Graph variables
struct Vertex* lstVertices[MAX];
int adjMatrix[MAX][MAX];
int vertexCount = 0;

// Stack functions
void push(int item) {
    stack[++top] = item;
}

int pop() {
    return stack[top--];
}

int peek() {
    return stack[top];
}

bool isStackEmpty() {
    return top == -1;
}

// Graph functions
void addVertex(char label) {
    struct Vertex* vertex = (struct Vertex*) malloc(sizeof(struct Vertex));
    vertex->label = label;
    vertex->visited = false;
    lstVertices[vertexCount++] = vertex;
}
```

```
void addEdge(int start, int end) {
    adjMatrix[start][end] = 1;
    adjMatrix[end][start] = 1;
}

void displayVertex(int vertexIndex) {
    printf("%c ", lstVertices[vertexIndex]->label);
}

int getAdjUnvisitedVertex(int vertexIndex) {
    for (int i = 0; i < vertexCount; i++) {
        if (adjMatrix[vertexIndex][i] == 1 && lstVertices[i]->visited ==
false) {
            return i;
        }
    }
    return -1;
}

void depthFirstSearch() {
    lstVertices[0]->visited = true;
    displayVertex(0);
    push(0);

    while (!isStackEmpty()) {
        int unvisitedVertex = getAdjUnvisitedVertex(peek());
        if (unvisitedVertex == -1) {
            pop();
        } else {
            lstVertices[unvisitedVertex]->visited = true;
            displayVertex(unvisitedVertex);
            push(unvisitedVertex);
        }
    }

    for (int i = 0; i < vertexCount; i++) {
        lstVertices[i]->visited = false;
    }
}

int main() {
    for (int i = 0; i < MAX; i++) {
        for (int j = 0; j < MAX; j++) {
            adjMatrix[i][j] = 0;
        }
    }

    addVertex('S'); // 0
```

```
    addVertex('A'); // 1
    addVertex('B'); // 2
    addVertex('C'); // 3
    addVertex('D'); // 4

    addEdge(0, 1); // S - A
    addEdge(0, 2); // S - B
    addEdge(0, 3); // S - C
    addEdge(1, 4); // A - D
    addEdge(2, 4); // B - D
    addEdge(3, 4); // C - D

    printf("Depth First Search: ");
    depthFirstSearch();

    return 0;
}
```

Output:

```
C:\Documents\sem4\daa\daa>gcc 43.dfs.c

C:\Documents\sem4\daa\daa>a
Depth First Search: S A D B C
```

Space Complexity: $O(v^2)$ – Adjacency matrix and stack uses extra space.

Time complexity: $O(v^2)$ – Each vertex checks all other vertices in the matrix.