DESIGN AND ANALYSIS OF ALGORITHMS

LAB WORKBOOK WEEK – 2 & 3

NAME: Konda Bhavani

ROLL NUMBER: CH.SC.U4CSE24120

CLASS: CSE-B

Sorting Techniques

Bubble Sort:

Code:

```c
#include <stdio.h>
void bubbleSort(int a[], int n) {
    for(int i = 0; i < n-1; i++)
        for(int j = 0; j < n-i-1; j++)
            if(a[j] > a[j+1]) {
                int temp = a[j];
                a[j] = a[j+1];
                a[j+1] = temp;
            }
}
int main() {
    int n;
    printf("Enter size: ");
    scanf("%d", &n);
    int a[n];
    printf("Enter elements: ");
    for(int i = 0; i < n; i++)
        scanf("%d", &a[i]);
    bubbleSort(a, n);
    printf("Sorted: ");
    for(int i = 0; i < n; i++) printf("%d ", a[i]);
}
```

Output:

```
Enter size: 5
Enter elements: 5 2 9 1 5
Sorted: 1 2 5 5 9
```

Time Complexity:

Outer loop runs (n−1) times.

Inner loop runs (n−i−1) times.

Total comparisons ≈ n * n.

Time Complexity = O(n^2)

Space Complexity:

int n → 4 bytes

int i → 4 bytes

int j → 4 bytes

int temp → 4 bytes

Total space used is constant.

Space Complexity = O(1)

Insertion Sort:

Code:

```c
#include <stdio.h>
void insertionSort(int a[], int n) {
    for(int i = 1; i < n; i++) {
        int key = a[i], 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 size: ");
    scanf("%d", &n);
    int a[n];
    printf("Enter elements: ");
    for(int i = 0; i < n; i++)
    scanf("%d", &a[i]);
    insertionSort(a, n);
    printf("Sorted: ");
    for(int i = 0; i < n; i++)
    printf("%d ", a[i]);
    printf("\n");
}
```

Output:

```
Enter size: 6
Enter elements: 8 4 3 7 6 1
Sorted: 1 3 4 6 7 8
```

Time Complexity:

Outer loop runs n times.

Inner while loop runs up to n times.

Total operations ≈ n * n.

Time Complexity = O(n^2)

Space Complexity:

int n → 4 bytes

int i → 4 bytes

int j → 4 bytes

int key → 4 bytes

Only constant extra space is used.

Space Complexity = O(1)

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

    selectionSort(a, n);

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

Output:

```
Enter size: 5
Enter elements: 10 3 5 2 9
Sorted: 2 3 5 9 10
```

Time Complexity:

Outer loop runs (n−1) times.

Inner loop runs n times.

Total comparisons ≈ n*n.

Time Complexity = O(n^2)

Space Complexity:

int n → 4 bytes

int i → 4 bytes

int j → 4 bytes

int min → 4 bytes

int temp → 4 bytes

Total space is constant.

Space Complexity = O(1)

Heap Sort:

Code:

```c
#include <stdio.h>
void heapify(int a[], int n, int i) {
    int largest = i;
    int left = 2*i + 1;
    int right = 2*i + 2;
    if(left < n && a[left] > a[largest]) largest = left;
    if(right < n && a[right] > a[largest]) largest = right;
    if(largest != i) {
        int temp = a[i];
        a[i] = a[largest];
        a[largest] = temp;
        heapify(a, n, largest);
    }
}
void heapSort(int a[], int n) {
    for(int i = n/2 - 1; i >= 0; i--)
        heapify(a, n, i);
    for(int i = n-1; i >= 0; i--) {
        int temp = a[0]; a[0] = a[i]; a[i] = temp;
        heapify(a, i, 0);
    }
}
```

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

Output:

```
amma@amma17:~/120$ ./heap
Enter size: 6
Enter elements: 12 11 13 5 6 7
Sorted: 5 6 7 11 12 13
```

Time Complexity:

Heap is built in O(n).

Heapify is called n times.

Each heapify takes log n time.

Time Complexity = O(n log n)

Space Complexity:

int n → 4 bytes

int i → 4 bytes

int temp → 4 bytes

Sorting is done in the same array.

Space Complexity = O(1)

Bucket Sort:

Code:

```c
#include <stdio.h>
#include <stdlib.h>
struct Node {
    float data;
    struct Node *next;
};
void bucketSort(float arr[], int n) {
    struct Node *b[n];
    for(int i = 0; i < n; i++) b[i] = NULL;
    for(int i = 0; i < n; i++) {
        int idx = arr[i] * n;
        struct Node *newNode = malloc(sizeof(struct Node));
        newNode->data = arr[i];
        newNode->next = b[idx];
        b[idx] = newNode;
    }
    for(int i = 0; i < n; i++)
        for(struct Node *p = b[i]; p && p->next; p = p->next)
            for(struct Node *q = p->next; q; q = q->next)
                if(p->data > q->data) {
                    float t = p->data; p->data = q->data; q->data = t;
                }
    int k = 0;
    for(int i = 0; i < n; i++)
        for(struct Node *p = b[i]; p; p = p->next)
            arr[k++] = p->data;
}
```

```c
int main() {
    int n;
    printf("Enter size: ");
    scanf("%d", &n);
    float arr[n];
    printf("Enter float elements (0 to 1): ");
    for(int i = 0; i < n; i++) scanf("%f", &arr[i]);
    bucketSort(arr, n);
    printf("Sorted: ");
    for(int i = 0; i < n; i++) printf("%.2f ", arr[i]);
    printf("\n");
}
```

Output:

```
Enter size: 7
Enter float elements (0 to 1): 0.25 0.65 0.10 0.45 0.80 0.32 0.50
Sorted: 0.10 0.25 0.32 0.45 0.50 0.65 0.80
```

Time Complexity:

Elements are distributed into n buckets.

Each bucket is sorted.

Average case runs in linear time.

Time Complexity = O(n)

Space Complexity:

float arr[n] → n elements

n buckets are created.

Extra memory increases with n.

Space Complexity = O(n)

Graph Traversals

BFS:

Code:

```c
#include <stdio.h>

#define MAX 100

int queue[MAX], front = 0, rear = 0, visited[MAX];

void enqueue(int x) { queue[rear++] = x; }
int dequeue() { return queue[front++]; }

void BFS(int graph[][MAX], int n, int start) {
    for(int i = 0; i < n; i++) visited[i] = 0;

    enqueue(start);
    visited[start] = 1;

    while(front != rear) {
        int v = dequeue();
        printf("%d ", v);

        for(int i = 0; i < n; i++)
            if(graph[v][i] == 1 && !visited[i]) {
                enqueue(i);
                visited[i] = 1;
            }
    }
}
```

```c
int main() {
    int n;
    printf("Number of nodes: ");
    scanf("%d", &n);

    int graph[MAX][MAX];
    printf("Enter adjacency matrix:\n");
    for(int i = 0; i < n; i++)
        for(int j = 0; j < n; j++)
            scanf("%d", &graph[i][j]);

    int start;
    printf("Enter starting node: ");
    scanf("%d", &start);

    printf("BFS: ");
    BFS(graph, n, start);
    printf("\n");
}
```

Output:

```
Number of nodes: 4
Enter adjacency matrix:
0 1 1 0
1 0 1 1
1 1 0 0
0 1 0 0
Enter starting node: 0
BFS: 0 1 2 3
```

Time Complexity:

Each vertex is visited once.

For each vertex, all n vertices are checked.

Total operations ≈ n * n.

Time Complexity = O(n^2)

Space complexity:

int graph[MAX][MAX] → fixed size

int queue[MAX] → fixed size

int visited[MAX] → fixed size

Total space is constant.

Space Complexity = O(1)

DFS:

Code:

```c
#include <stdio.h>

#define MAX 100

int visited[MAX];

void DFS(int graph[][MAX], int n, int v) {
    visited[v] = 1;
    printf("%d ", v);

    for(int i = 0; i < n; i++)
        if(graph[v][i] == 1 && !visited[i])
            DFS(graph, n, i);
}
```

```c
int main() {
    int n;
    printf("Number of nodes: ");
    scanf("%d", &n);

    int graph[MAX][MAX];
    printf("Enter adjacency matrix:\n");
    for(int i = 0; i < n; i++)
        for(int j = 0; j < n; j++)
            scanf("%d", &graph[i][j]);

    int start;
    printf("Enter starting node: ");
    scanf("%d", &start);

    for(int i = 0; i < n; i++) visited[i] = 0;

    printf("DFS: ");
    DFS(graph, n, start);
    printf("\n");
}
```

Output:

```
Number of nodes: 4
Enter adjacency matrix:
0 1 1 0
1 0 1 1
1 1 0 0
0 1 0 0
Enter starting node: 1
DFS: 1 0 2 3
```

Time Complexity:

Each vertex is visited once.

For each vertex, all n vertices are checked.

Time Complexity = O(n^2)

Space Complexity:

int visited[MAX] → fixed size

Recursive calls up to n.

Space depends on n.

Space Complexity = O(n)