NAME: MARADANA SRIJA

ROLL NO: CH.SC.U4CSE24126

(Design and Analysis Algorithms)

WEEK2 & 3 :-

1Q) BUBBLE SORT

CODE:

```c
#include <stdio.h>

void swap(int* arr, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}

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

int main() {
    int arr[] = { 45, 36, 18, 93 };
    int n = sizeof(arr) / sizeof(arr[0]);

    bubbleSort(arr, n);

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

OUTPUT:

```
amma@amma23:~/126$ gcc bubbsort.c -o bubbsort
amma@amma23:~/126$ ./bubbsort
18
36
45
93
```

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)

2Q) INSERTION SORT

CODE:

```c
#include <stdio.h>

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

    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;
    }
    printf("Sorted array:\n");
    for(i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    return 0;
}
```

OUTPUT:

```
amma@amma23:~/DAA$ gcc insertionsort.c -o insertionsort
amma@amma23:~/DAA$ ./insertionsort
Enter number of elements: 5
Enter 5 elements:
1
2
6
4
5
Sorted array:
1 2 4 5 6 amma@amma23:~/DAA$
```

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)

3Q) SELECTION SORT

CODE:

```c
#include <stdio.h>

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

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

    for(i = 0; i < n - 1; i++) {
        minIndex = i;

        for(j = i + 1; j < n; j++) {
            if(arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        temp = arr[i];
        arr[i] = arr[minIndex];
        arr[minIndex] = temp;
    }
    printf("Sorted array:\n");
    for(i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
```

OUTPUT:

```
amma@amma23:~/DAA$ gcc selectionsort.c -o selectionsort
amma@amma23:~/DAA$ ./selectionsort
Enter number of elements: 5
Enter 5 elements:
5
1
2
3
7
Sorted array:
1 2 3 5 7 amma@amma23:~/DAA$
```

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)

4Q) BUCKET SORT

CODE:

```c
//CH.SC.U4CSE24126

#include <stdio.h>
#include <stdlib.h>

#define N 10

void bucketSort(float arr[], int n) {
    int i, j;
    float bucket[N][N];
    int count[N] = {0};

    for (i = 0; i < n; i++) {
        int index = n * arr[i];
        bucket[index][count[index]++] = arr[i];
    }

    for (i = 0; i < N; i++) {
        for (j = 0; j < count[i] - 1; j++) {
            for (int k = j + 1; k < count[i]; k++) {
                if (bucket[i][j] > bucket[i][k]) {
                    float temp = bucket[i][j];
                    bucket[i][j] = bucket[i][k];
                    bucket[i][k] = temp;
                }
            }
```

```
        }
    }
    int idx = 0;
    for (i = 0; i < N; i++) {
        for (j = 0; j < count[i]; j++) {
            arr[idx++] = bucket[i][j];
        }
    }
}
int main() {
    float arr[] = {0.45, 0.33, 0.23, 0.18, 0.93, 0.10, 0.64};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("CH.SC.U4CSE24126\n");
    printf("Before Sorting:\n");
    for (int i = 0; i < n; i++)
        printf("%.2f ", arr[i]);
    bucketSort(arr, n);
    printf("\nAfter Sorting:\n");
    for (int i = 0; i < n; i++)
        printf("%.2f ", arr[i]);
    return 0;
}
```

OUTPUT:

```
CH.SC.U4CSE24126
Before Sorting:
0.45 0.33 0.23 0.18 0.93 0.10 0.64
After Sorting:
0.10 0.18 0.23 0.33 0.45 0.64 0.93
```

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)

5Q) HEAP SORT

CODE:

```c
//CH.SC.U4CSE24126

#include <stdio.h>

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

    if (left < n && arr[left] > arr[largest])
        largest = left;

    if (right < n && arr[right] > arr[largest])
        largest = right;

    if (largest != i) {
        int temp = arr[i];
        arr[i] = arr[largest];
        arr[largest] = temp;

        heapify(arr, n, largest);
    }
}

void heapSort(int arr[], int n) {
    for (int i = n / 2 - 1; i >= 0; i--)
```

```
        heapify(arr, n, i);

    for (int i = n - 1; i > 0; i--) {
        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;

        heapify(arr, i, 0);
    }
}

int main() {
    int arr[] = {12, 11, 13, 5, 6, 7};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("CH.SC.U4CSE24126\n");
    printf("Before Sorting:\n");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);

    heapSort(arr, n);

    printf("\nAfter Sorting:\n");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
```

```
    return 0;
}
```

OUTPUT:

```
CH.SC.U4CSE24126
Before Sorting:
12 11 13 5 6 7
After Sorting:
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)

6Q) BFS

CODE:

```c
#include <stdio.h>

int queue[50], front = -1, rear = -1;

void enqueue(int x) {
    if (rear == 49) return;
    if (front == -1) front = 0;
    queue[++rear] = x;
}

int dequeue() {
    if (front == -1 || front > rear) return -1;
    return queue[front++];
}

int main() {
    int n, adj[50][50], visited[50] = {0}, start;

    printf("Enter number of vertices: ");
    scanf("%d", &n);

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

    printf("Enter starting vertex (0 to %d): ", n-1);
    scanf("%d", &start);
```

OUTPUT:

```
BFS Traversal: 0 amma@amma23:~/DAA$ gcc BFS.c -o BFS
amma@amma23:~/DAA$ ./BFS
Enter number of vertices: 3
Enter adjacency matrix:
1
2
3
4
5
6
7
8
9
Enter starting vertex (0 to 2): 1
BFS Traversal: 1 amma@amma23:~/DAA$
```

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.

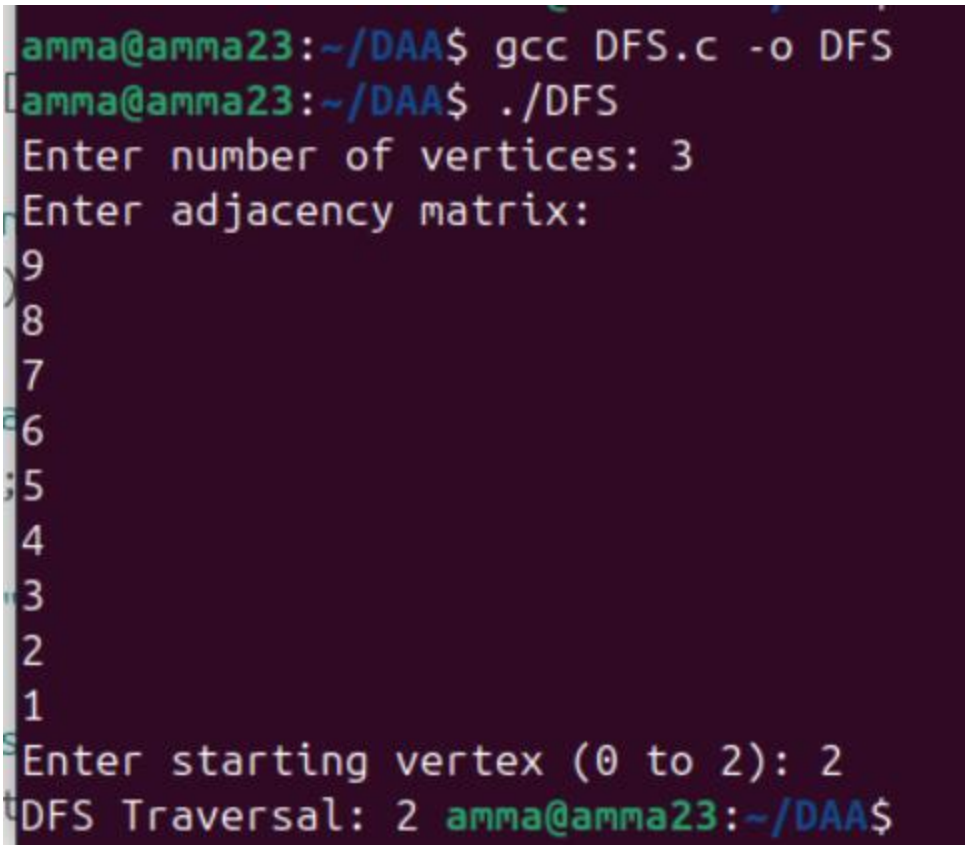
Space Complexity = O(1)

7Q) DFS

CODE:

```c
#include <stdio.h>

void dfs(int node, int n, int adj[50][50], int visited[50
    visited[node] = 1;
    printf("%d ", node);

    for (int i = 0; i < n; i++) {
        if (adj[node][i] == 1 && !visited[i]) {
            dfs(i, n, adj, visited);
        }
    }
}

int main() {
    int n, adj[50][50], visited[50] = {0}, start;

    printf("Enter number of vertices: ");
    scanf("%d", &n);

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

    printf("Enter starting vertex (0 to %d): ", n-1);
    scanf("%d", &start);

    printf("DFS Traversal: ");
    dfs(start, n, adj, visited);
```

OUTPUT:



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)