AMRITA VISHWA VIDYAPEETHAM
DEEMED TO BE UNIVERSITY UNDER SECTION 3 OF UGC ACT, 1956

SCHOOL OF COMPUTING

Devadharshan.S

CH.SC.U4CSE24113

Week – 2

Date - 04/12/2025

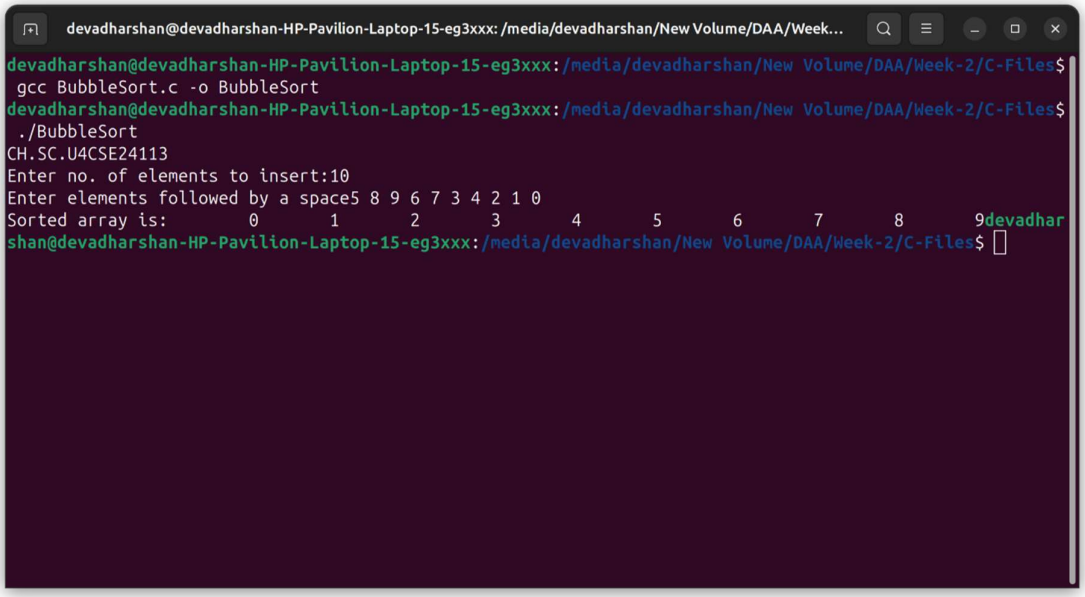
Design and Analysis of Algorithm(23CSE211)

1. Bubble Sort

Code:

```
#include<stdio.h>
int arr[100];
int n;
void swap(int *a,int *b){
  int temp=*a;
  *a=*b;
  *b=temp;
}
void Sort(int arr[]){
  for(int i=0;i<n;i++){
    for(int j=0;j<n-i-1;j++){
      if(arr[j]>arr[j+1]){
        swap(&arr[j],&arr[j+1]);
      }
    }
  }
}
int main(){
  printf("CH.SC.U4CSE24113\n");
  printf("Enter no. of elements to insert:");
  scanf("%d",&n);
  printf("Enter elements followed by a space");
  for(int i=0;i<n;i++){
    scanf("%d",&arr[i]);
  }
  Sort(arr);
  printf("Sorted array is:");
  for(int i=0;i<n;i++){
    printf("\t%d",arr[i]);
  }
}
```

Output:



Space Complexity: O(1)
Justification: The sorting happens in-place within the original array. You only use a single temporary variable (temp) for swapping, so no extra memory is required relative to the input size.

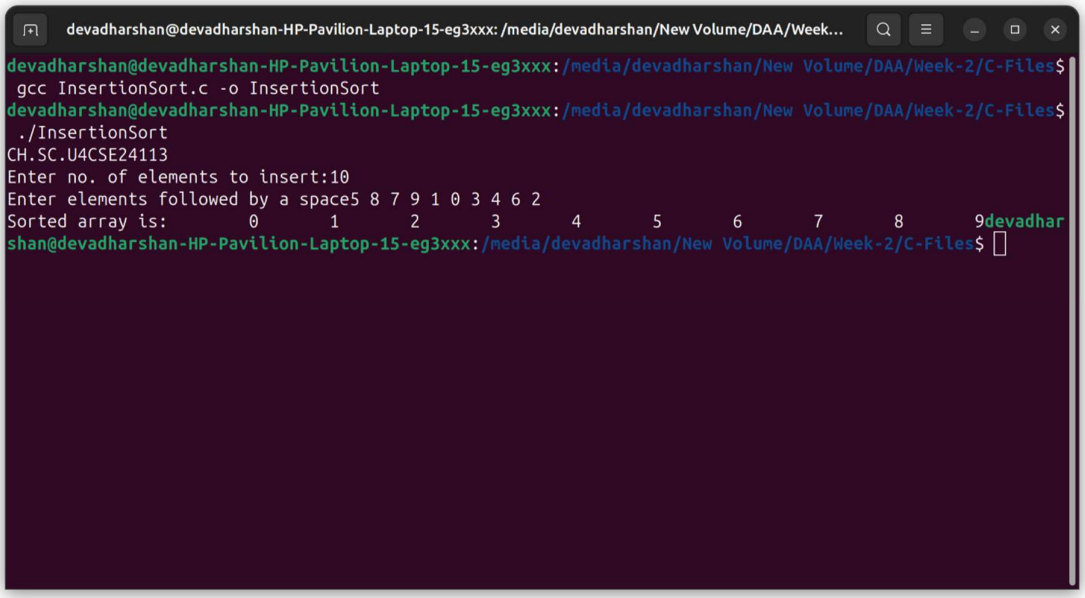
Time Complexity: $O(n^2)$
Justification: We have two nested loops. The outer loop runs n times, and for every iteration of the outer loop, the inner loop also runs roughly n times. This results in n x n operations.

2. Insertion Sort

Code:

```
#include<stdio.h>
int arr[100];
int n;
void Sort(int arr[]){
  for(int i=1;i<n;i++){
    int j=i-1;
    int temp=arr[i];
    while(j>=0 && temp<arr[j]){
      arr[j+1]=arr[j];
      j--;
    }
    arr[j+1]=temp;
  }
}
int main(){
  printf("CH.SC.U4CSE24113\n");
  printf("Enter no. of elements to insert:");
  scanf("%d",&n);
  printf("Enter elements followed by a space");
  for(int i=0;i<n;i++){
    scanf("%d",&arr[i]);
  }
  Sort(arr);
  printf("Sorted array is:");
  for(int i=0;i<n;i++){
    printf("\t%d",arr[i]);
  }
}
```

Output:



Space Complexity: O(1)
Justification: The sorting is performed in-place using the same array. Only one temporary variable (temp) is used, so no extra memory is needed.

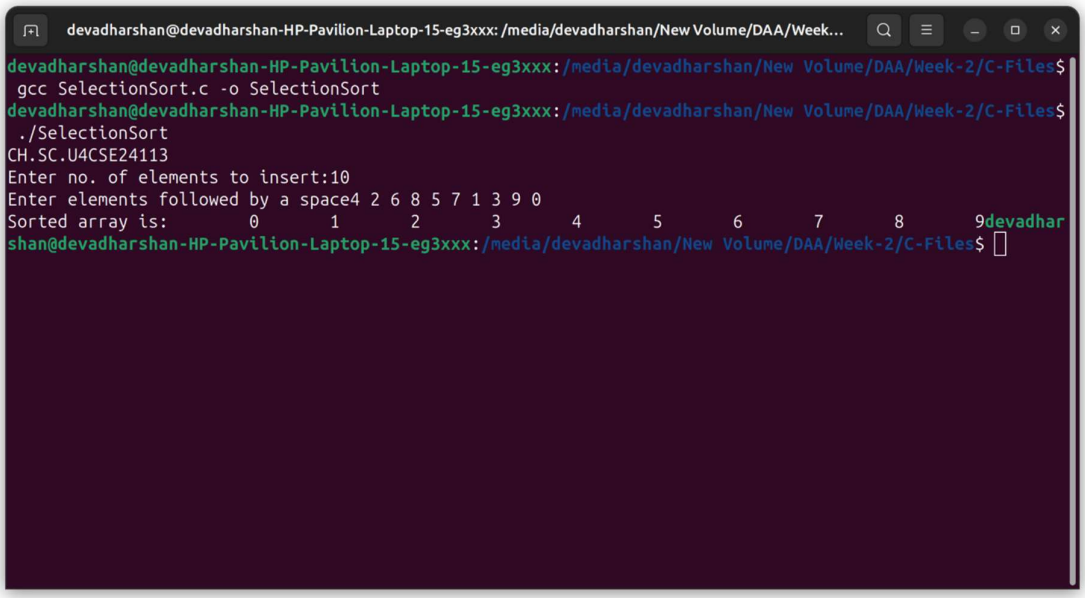
Time Complexity: $O(n^2)$
Justification: For every element i, the inner while loop runs i times to shift elements. This results in n x n operations.

3.Selection Sort

Code:

```
#include<stdio.h>
int arr[100];
int n;
void swap(int *a,int *b){
  int temp=*a;
  *a=*b;
  *b=temp;
}
void Sort(int arr[]){
  for(int i=0;i<n;i++){
    int min=i;
    for(int j=i+1;j<n;j++){
      if(arr[min]>arr[j]){
        min=j;
      }
    }
    if(min!=i){
      swap(&arr[min],&arr[i]);
    }
  }
}
int main(){
  printf("CH.SC.U4CSE24113\n");
  printf("Enter no. of elements to insert:");
  scanf("%d",&n);
  printf("Enter elements followed by a space");
  for(int i=0;i<n;i++){
    scanf("%d",&arr[i]);
  }
  Sort(arr);
  printf("Sorted array is:");
  for(int i=0;i<n;i++){
    printf("\t%d",arr[i]);
  }
}
```

Output:



Time Complexity: $O(n^2)$
Justification: There are two nested loops. Regardless of the data order, the inner loop iterates through the remaining unsorted elements to find the minimum. This always results in roughly $\frac{n(n-1)}{2}$ comparisons, or n x n operations.

Space Complexity: O(1)
Justification: The algorithm sorts the array in-place. It uses only a few variables (min, i, j, temp) for bookkeeping, requiring no extra memory proportional to the input size.

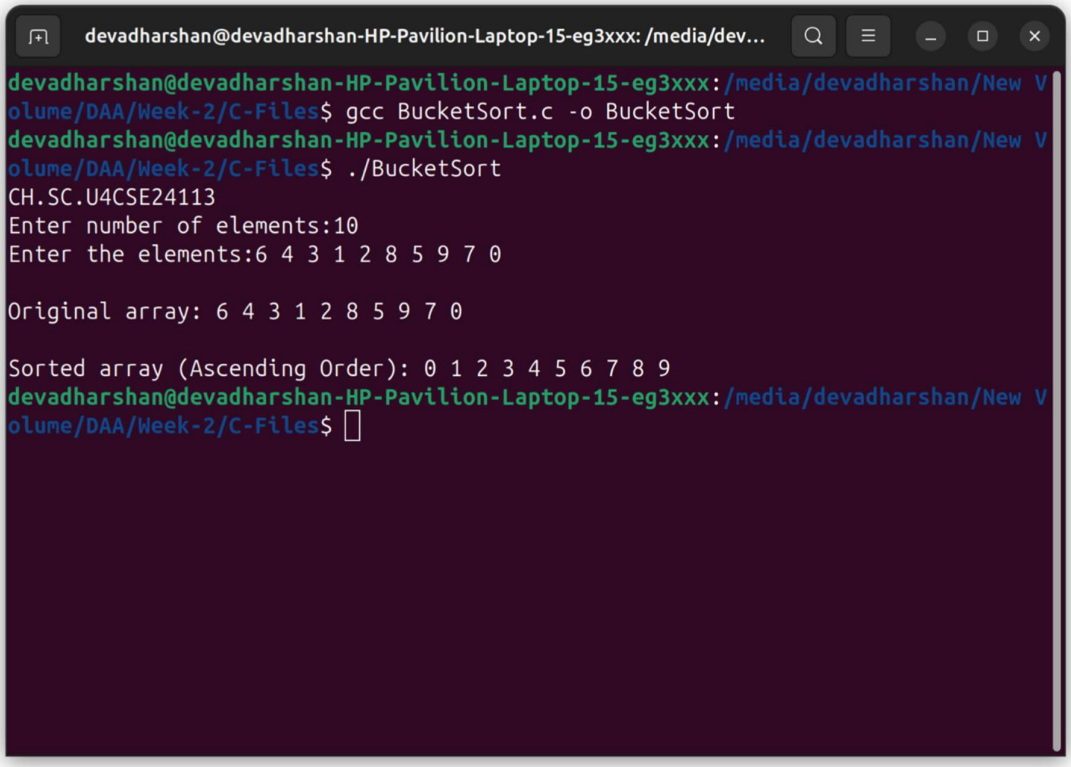
4. Bucket Sort

Code:

```
#include <stdio.h>
#define MAX 100
int arr[MAX];
int size = 0;
void readArray(int n) {
    size = n;
    printf("Enter the elements:");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
}
void display(int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
void bucketSort() {
    int buckets[10][MAX];
    int count[10];
    int i, j, k, max = 0;
    for (i = 0; i < 10; i++) count[i] = 0;
    for (i = 0; i < size; i++) {
        if (arr[i] > max) max = arr[i];
    }
    for (i = 0; i < size; i++) {
        int idx = (arr[i] * 10) / (max + 1);
        buckets[idx][count[idx]++] = arr[i];
    }
    for (i = 0; i < 10; i++) {
        for (j = 1; j < count[i]; j++) {
            int key = buckets[i][j];
            k = j - 1;
            while (k >= 0 && buckets[i][k] > key) {
                buckets[i][k + 1] = buckets[i][k];
                k--;
            }
            buckets[i][k + 1] = key;
        }
    }
    k = 0;
    for (i = 0; i < 10; i++) {
        for (j = 0; j < count[i]; j++) {
            arr[k++] = buckets[i][j];
        }
    }
}
int main() {
    int n;
    printf("CH.SC.U4CSE24113\n");
    printf("Enter number of elements:");
```

```
    scanf("%d", &n);
    readArray(n);
    printf("\nOriginal array: ");
    display(size);
    bucketSort();
    printf("\nSorted array (Ascending Order): ");
    display(size);
    return 0;
}
```

Output:



Time Complexity: $O(n^2)$
Justification: In the worst case, if the data is not uniformly distributed (e.g., all elements fall into the same bucket), the algorithm relies on the inner loop which performs Insertion Sort. Since Insertion Sort has a worst-case time of $O(n^2)$, the total time becomes quadratic.

Space Complexity: $O(n)$
Justification: The algorithm uses an auxiliary 2D array buckets[10][MAX] to temporarily store and sort the elements. This requires extra memory proportional to the size of the input data (n).

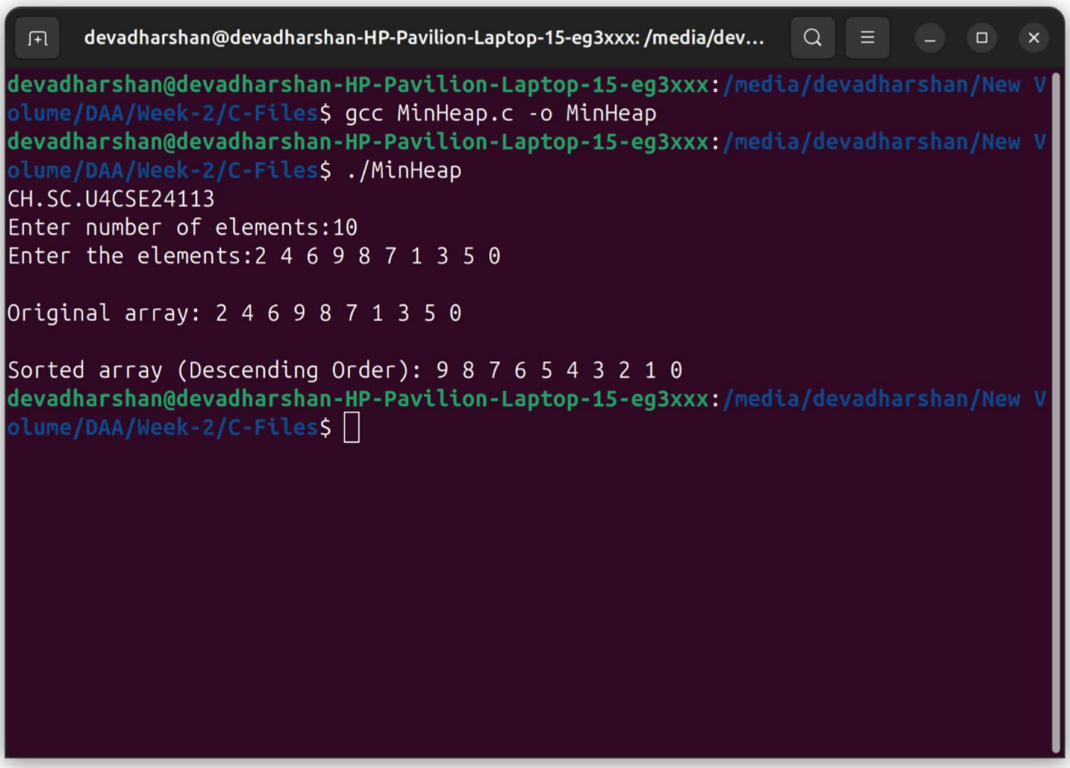
5.Heap Sort (Min Heap)

Code:

```
#include <stdio.h>
#define MAX 100
int arr[MAX];
int size = 0;
void swap(int *a, int *b) {
    int t = *a; *a = *b; *b = t;
}
void min_heapify(int n, int i) {
    int smallest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;
    if (left < n && arr[left] < arr[smallest]) {
        smallest = left;
    }
    if (right < n && arr[right] < arr[smallest]) {
        smallest = right;
    }
    if (smallest != i) {
        swap(&arr[i], &arr[smallest]);
        min_heapify(n, smallest);
    }
}
void buildMinHeap(int n) {
    for (int i = n / 2 - 1; i >= 0; i--) {
        min_heapify(n, i);
    }
}
void heapSort() {
    int n = size;
    buildMinHeap(n);
    for (int i = n - 1; i > 0; i--) {
        swap(&arr[0], &arr[i]);
        min_heapify(i, 0);
    }
}
void readArray(int n) {
    size = n;
    printf("Enter the elements:");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
}
void display(int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
int main() {
    int n;
    printf("CH.SC.U4CSE24113\n");
```

```
    printf("Enter number of elements:");
    scanf("%d", &n);
    readArray(n);
    printf("\nOriginal array: ");
    display(size);
    heapSort();
    printf("\nSorted array (Descending Order): ");
    display(size);
    return 0;
}
```

Output:



Time Complexity: O(n log n)
Justification: The algorithm first builds a heap in O(n). Then, it performs a loop n times to extract elements. In each iteration, it calls min_heapify, which traverses the height of the tree (log n) to restore the heap property. Therefore, the total time is n log n.

Space Complexity: O(1)
Justification: The sorting is performed in-place. While the recursive min_heapify uses stack space proportional to the tree height (O(log n)), the algorithm does not allocate any new arrays or data structures to hold the elements.

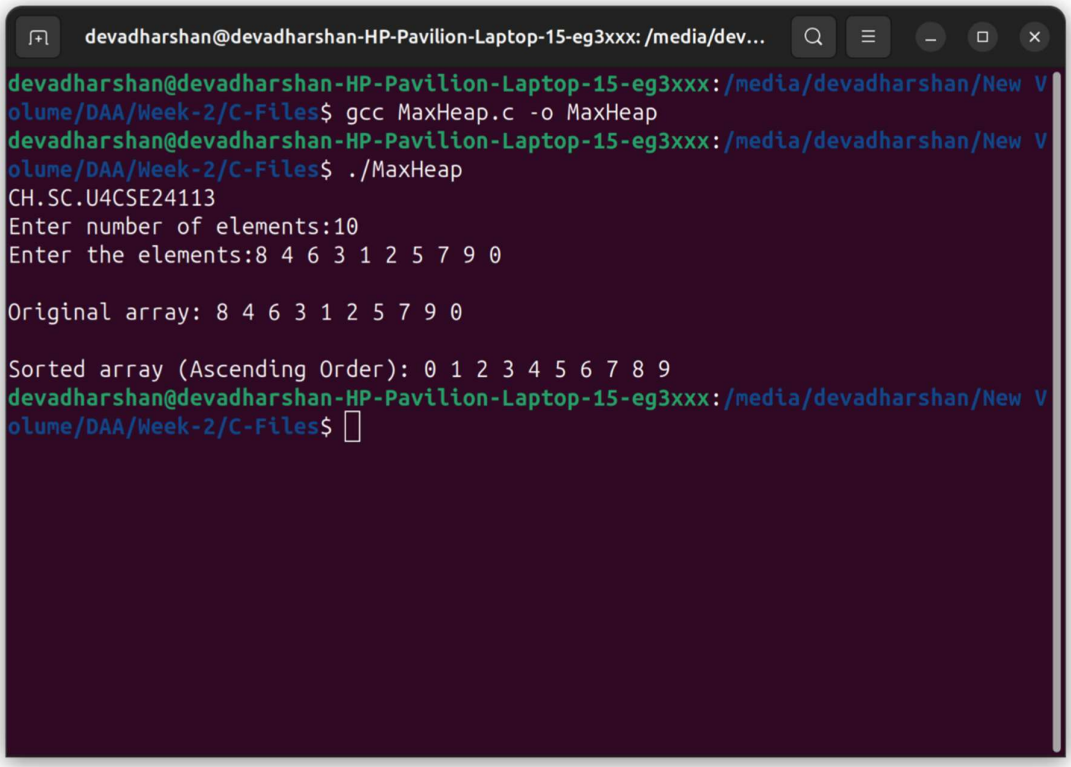
6. Heap Sort (Max Heap)

Code:

```
#include <stdio.h>
#define MAX 100
int arr[MAX];
int size = 0;
void swap(int *a, int *b) {
    int t = *a; *a = *b; *b = t;
}
void max_heapify(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) {
        swap(&arr[i], &arr[largest]);
        max_heapify(n, largest);
    }
}
void buildMaxHeap(int n) {
    for (int i = n / 2 - 1; i >= 0; i--) {
        max_heapify(n, i);
    }
}
void heapSort() {
    int n = size;
    buildMaxHeap(n);
    for (int i = n - 1; i > 0; i--) {
        swap(&arr[0], &arr[i]);
        max_heapify(i, 0);
    }
}
void readArray(int n) {
    size = n;
    printf("Enter the elements:");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
}
void display(int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
```

```
int main() {
    int n;
    printf("CH.SC.U4CSE24113\n");
    printf("Enter number of elements:");
    scanf("%d", &n);
    readArray(n);
    printf("\nOriginal array: ");
    display(size);
    heapSort();
    printf("\nSorted array (Ascending Order): ");
    display(size);
    return 0;
}
```

Output:



Time Complexity: O(n log n)
Justification: The algorithm first builds the max heap, which takes O(n). Then, it loops n times to swap the root with the last element. In each iteration, it calls max_heapify, which traverses the height of the tree (log n) to restore the heap property. The total time is dominated by n log n.

Space Complexity: O(1)
Justification: The sorting is performed in-place within the given array. While the recursive max_heapify function uses stack space proportional to the tree height (O(log n)), the algorithm does not allocate any separate data structures to store the input.

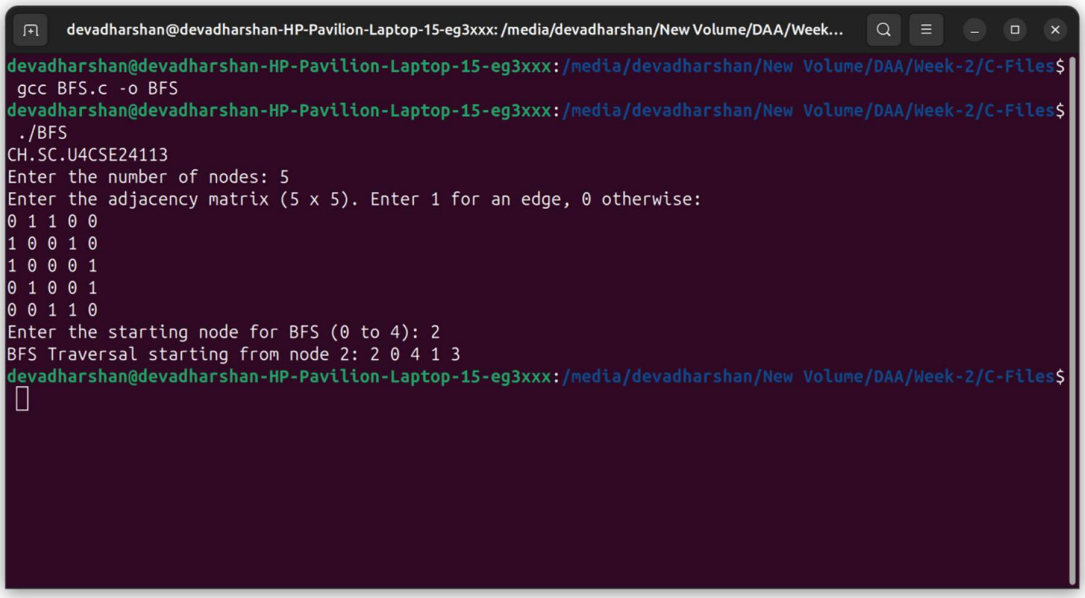
7. Breadth first search (BFS)

Code:

```
#include <stdio.h>
#include <stdbool.h>
#define MAX_NODES 100
int queue[MAX_NODES];
int front = -1;
int rear = -1;
void enqueue(int data) {
    if (front == -1) {
        front = 0;
    }
    rear++;
    queue[rear] = data;
}
int dequeue() {
    int data;
    if (front == -1) {
        return -1;
    }
    data = queue[front];
    front++;
    if (front > rear) {
        front = rear = -1;
    }
    return data;
}
bool is_empty() {
    return front == -1;
}
void bfs(int graph[MAX_NODES][MAX_NODES], int num_nodes, int start_node) {
    bool visited[MAX_NODES] = {false};
    enqueue(start_node);
    visited[start_node] = true;
    printf("BFS Traversal starting from node %d: ", start_node);
    while (!is_empty()) {
        int current_node = dequeue();
        printf("%d ", current_node);
        for (int i = 0; i < num_nodes; i++) {
            if (graph[current_node][i] == 1 && !visited[i]) {
                enqueue(i);
                visited[i] = true;
            }
        }
    }
    printf("\n");
}
int main() {
    printf("CH.SC.U4CSE24113\n");
    int num_nodes;
    int graph[MAX_NODES][MAX_NODES];
    int start_node;
    printf("Enter the number of nodes: ");
```

```
    scanf("%d", &num_nodes);
    printf("Enter the adjacency matrix (%d x %d). Enter 1 for an edge, 0
otherwise:\n", num_nodes, num_nodes);
    for (int i = 0; i < num_nodes; i++) {
        for (int j = 0; j < num_nodes; j++) {
            scanf("%d", &graph[i][j]);
        }
    }
    printf("Enter the starting node for BFS (0 to %d): ", num_nodes - 1);
    scanf("%d", &start_node);
    bfs(graph, num_nodes, start_node);
    return 0;
}
```

Output:



Time Complexity: $O(n^2)$
Justification: The while loop runs for every node (n times). Inside this loop, the for loop iterates through the entire row of the adjacency matrix (size n) to find connected neighbors. This results in n x n operations.

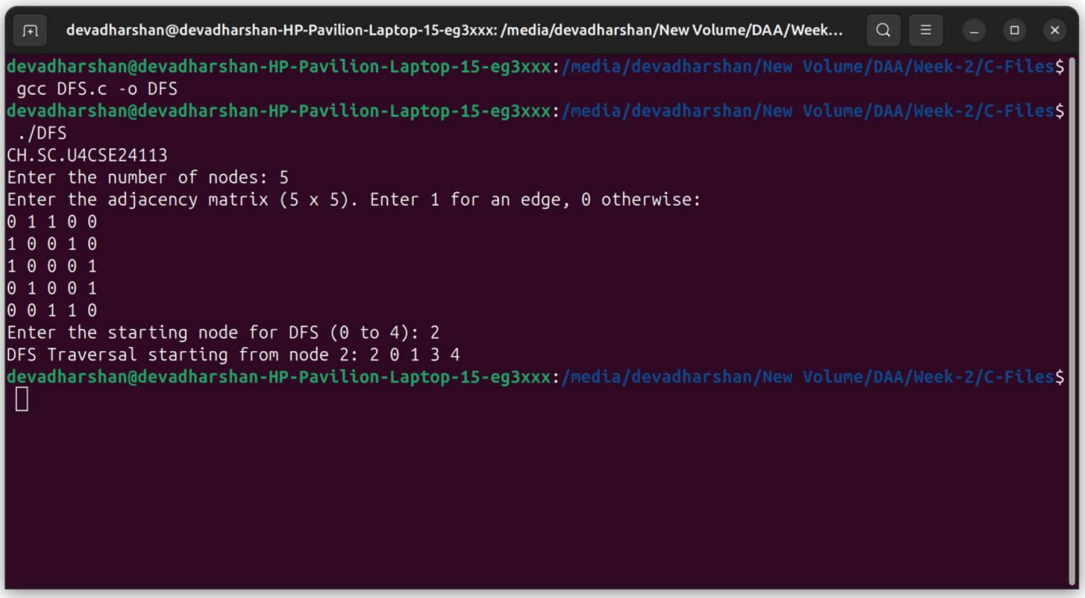
Space Complexity: O(n)
Justification: The algorithm uses two additional data structures: the queue array and the visited array. Both of these scales linearly with the number of nodes (n).
(Note: The input graph itself takes $O(n^2)$ space to store the matrix, but the traversal logic only requires O(n) extra space.)

8. Depth first search (DFS)

Code:

```
#include <stdio.h>
#include <stdbool.h>
#define MAX_NODES 100
bool visited[MAX_NODES];
void dfs_recursive(int graph[MAX_NODES][MAX_NODES], int num_nodes, int
current_node) {
    visited[current_node] = true;
    printf("%d ", current_node);
    for (int i = 0; i < num_nodes; i++) {
        if (graph[current_node][i] == 1 && !visited[i]) {
            dfs_recursive(graph, num_nodes, i);
        }
    }
}
void dfs(int graph[MAX_NODES][MAX_NODES], int num_nodes, int start_node) {
    for (int i = 0; i < num_nodes; i++) {
        visited[i] = false;
    }

    printf("DFS Traversal starting from node %d: ", start_node);
    dfs_recursive(graph, num_nodes, start_node);
    printf("\n");
}
int main() {
    printf("CH.SC.U4CSE24113\n");
    int num_nodes;
    int graph[MAX_NODES][MAX_NODES];
    int start_node;
    printf("Enter the number of nodes: ");
    scanf("%d", &num_nodes);
    printf("Enter the adjacency matrix (%d x %d). Enter 1 for an edge, 0
otherwise:\n", num_nodes, num_nodes);
    for (int i = 0; i < num_nodes; i++) {
        for (int j = 0; j < num_nodes; j++) {
            scanf("%d", &graph[i][j]);
        }
    }
    printf("Enter the starting node for DFS (0 to %d): ", num_nodes - 1);
    scanf("%d", &start_node);
    dfs(graph, num_nodes, start_node);
    return 0;
}
```

Output:



Time Complexity: $O(n^2)$
Justification: The function dfs_recursive visits every node once (n times). Inside each recursive call, the loop runs n times to check the adjacency row for connected neighbors. This leads to a total of n x n operations.

Space Complexity: O(n)
Justification: The auxiliary space comes from the recursion stack and the visited array. In the worst case (a skewed tree/path graph), the recursion stack can go as deep as the number of nodes (n).
(Note: Similar to BFS, while the matrix input takes $O(n^2)$, the algorithm's extra memory usage is only O(n).)