DATA ANALYSIS AND ALGORITHMS

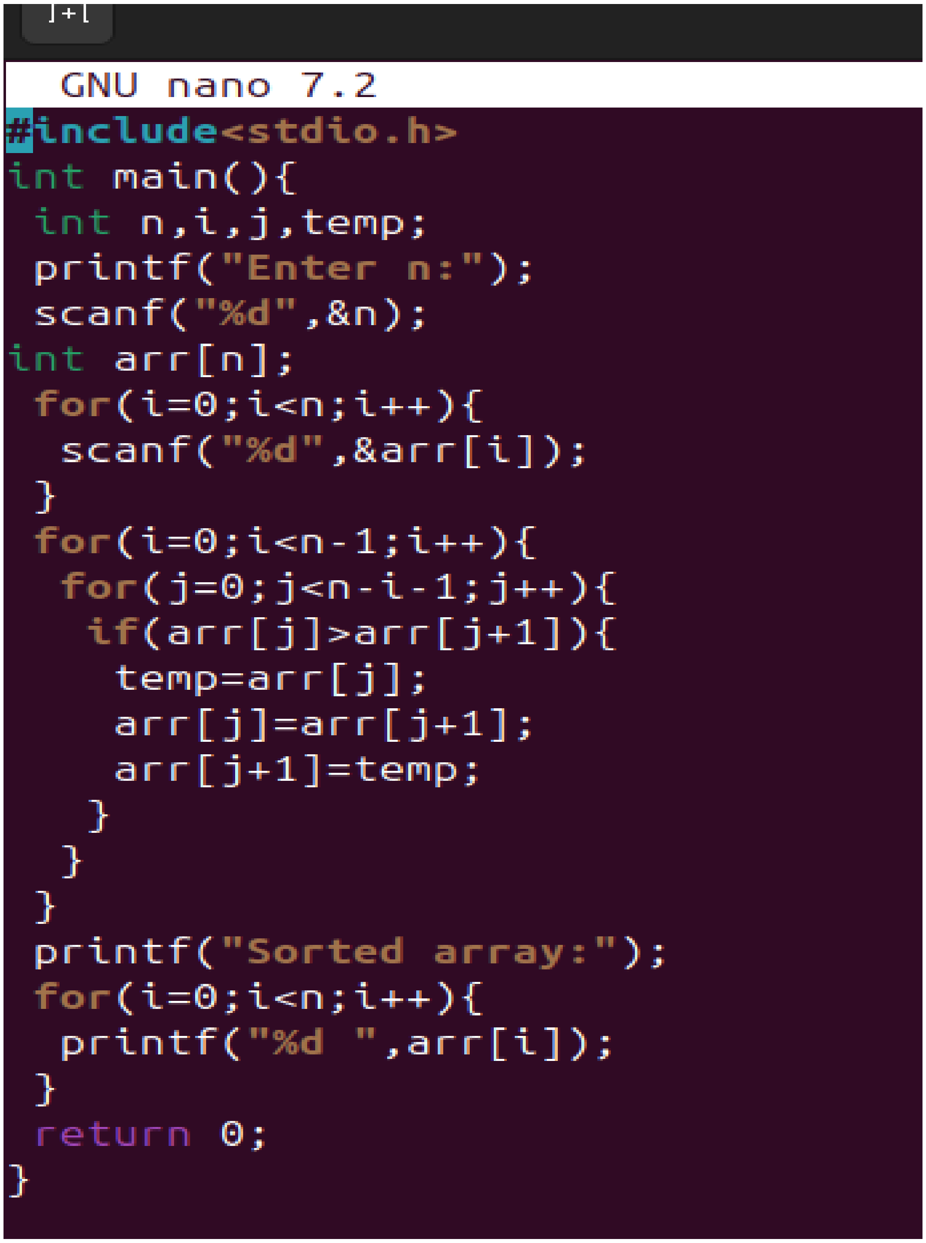
(23CSE211)

SORTING TECHNIQUES

P.JEEVAN SANDEEP

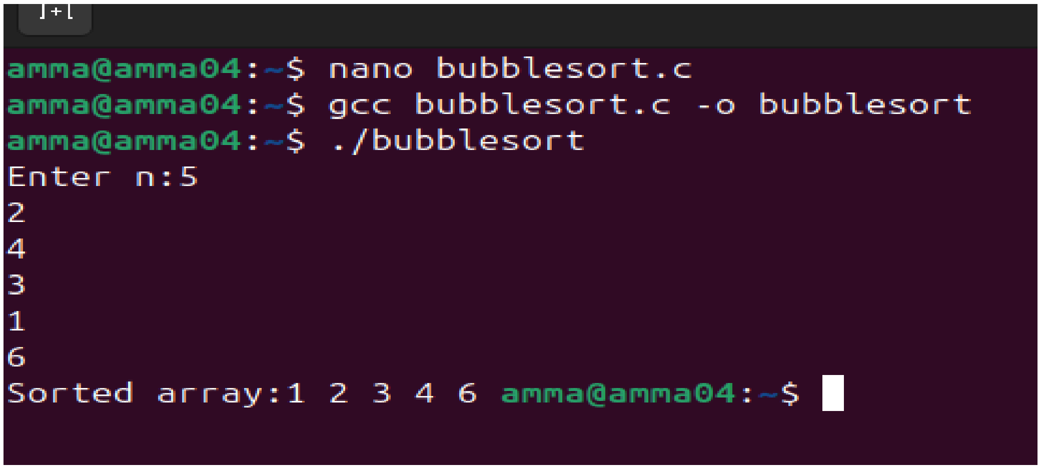
CH.SC.U4CSE24134

BUBBLE SORT:



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

Output:



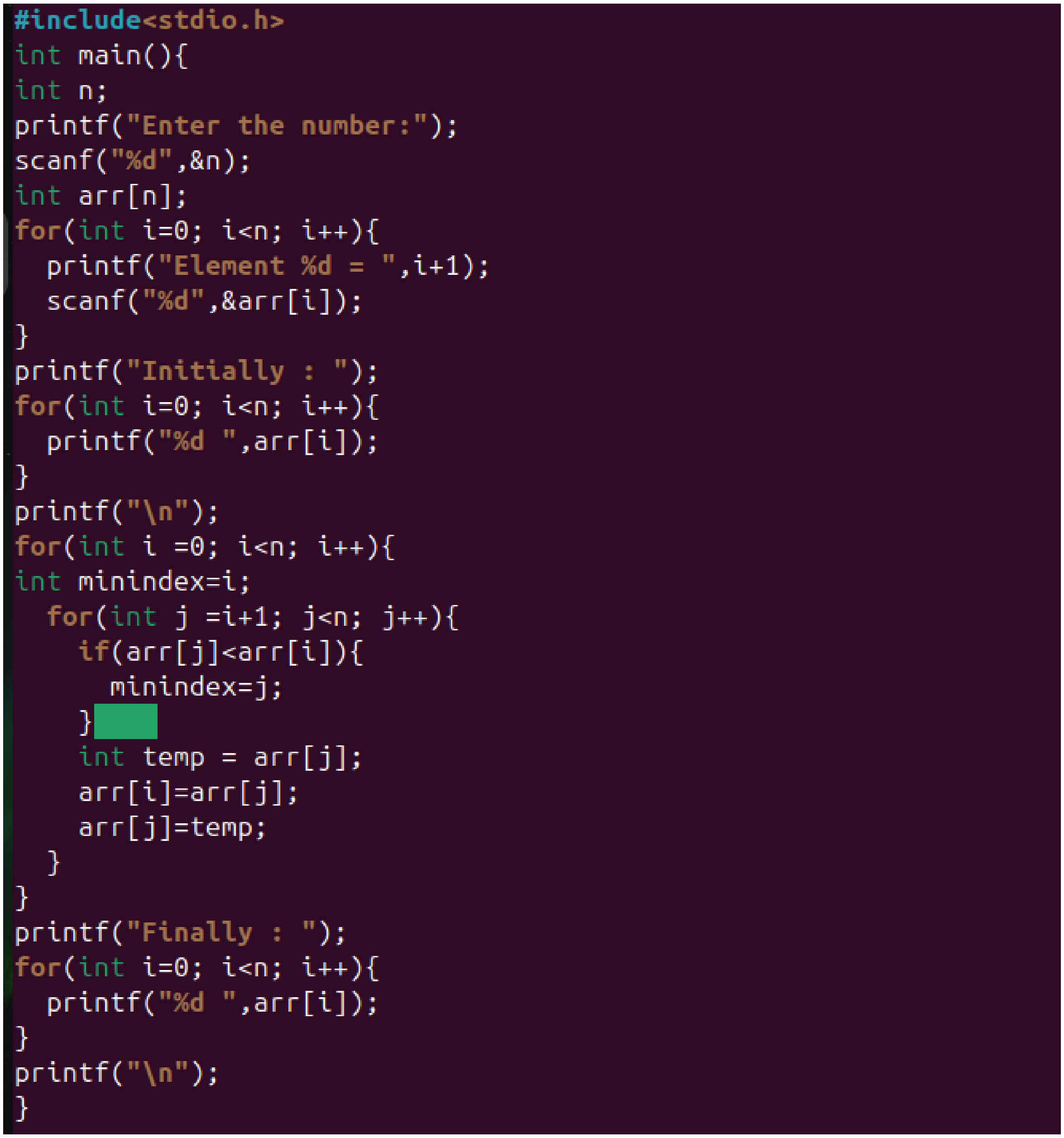
```
amma@amma04:~$ nano bubblesort.c
amma@amma04:~$ gcc bubblesort.c -o bubblesort
amma@amma04:~$ ./bubblesort
Enter n:5
2
4
3
1
6
Sorted array:1 2 3 4 6 amma@amma04:~$
```

Interpretation: Repeatedly compares adjacent elements and swaps them if they are in the wrong order until the array is sorted.

Complexity: Time – O(n^2) Space – O(1)

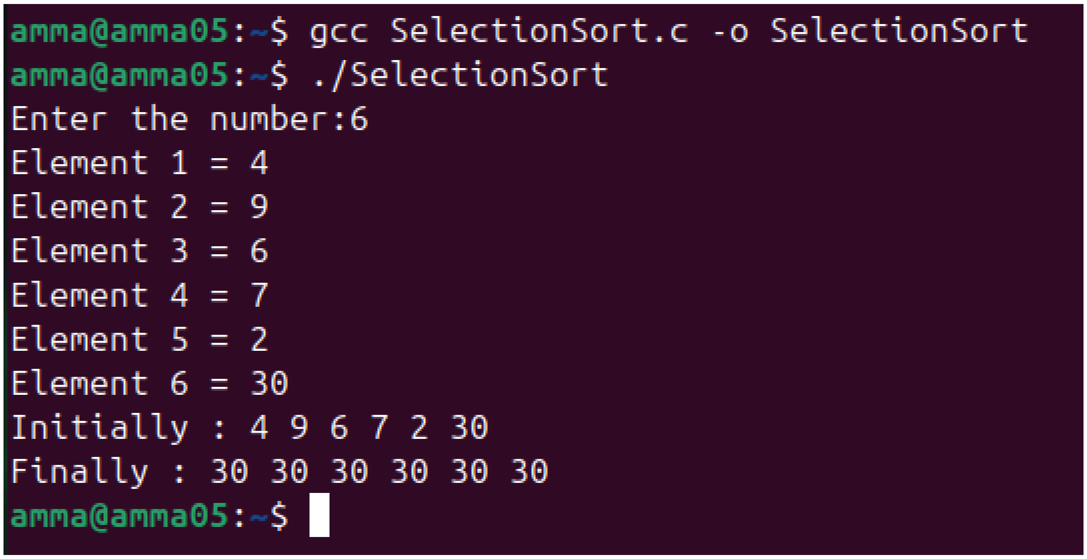
SELECTION SORT

Code:



```
#include<stdio.h>
int main(){
int n;
printf("Enter the number:");
scanf("%d",&n);
int arr[n];
for(int i=0; i<n; i++){
  printf("Element %d = ",i+1);
  scanf("%d",&arr[i]);
}
printf("Initially : ");
for(int i=0; i<n; i++){
  printf("%d ",arr[i]);
}
printf("\n");
for(int i =0; i<n; i++){
int minindex=i;
  for(int j =i+1; j<n; j++){
    if(arr[j]<arr[i]){
      minindex=j;
    }
    int temp = arr[j];
    arr[i]=arr[j];
    arr[j]=temp;
  }
}
printf("Finally : ");
for(int i=0; i<n; i++){
  printf("%d ",arr[i]);
}
printf("\n");
}
```

OUTPUT:



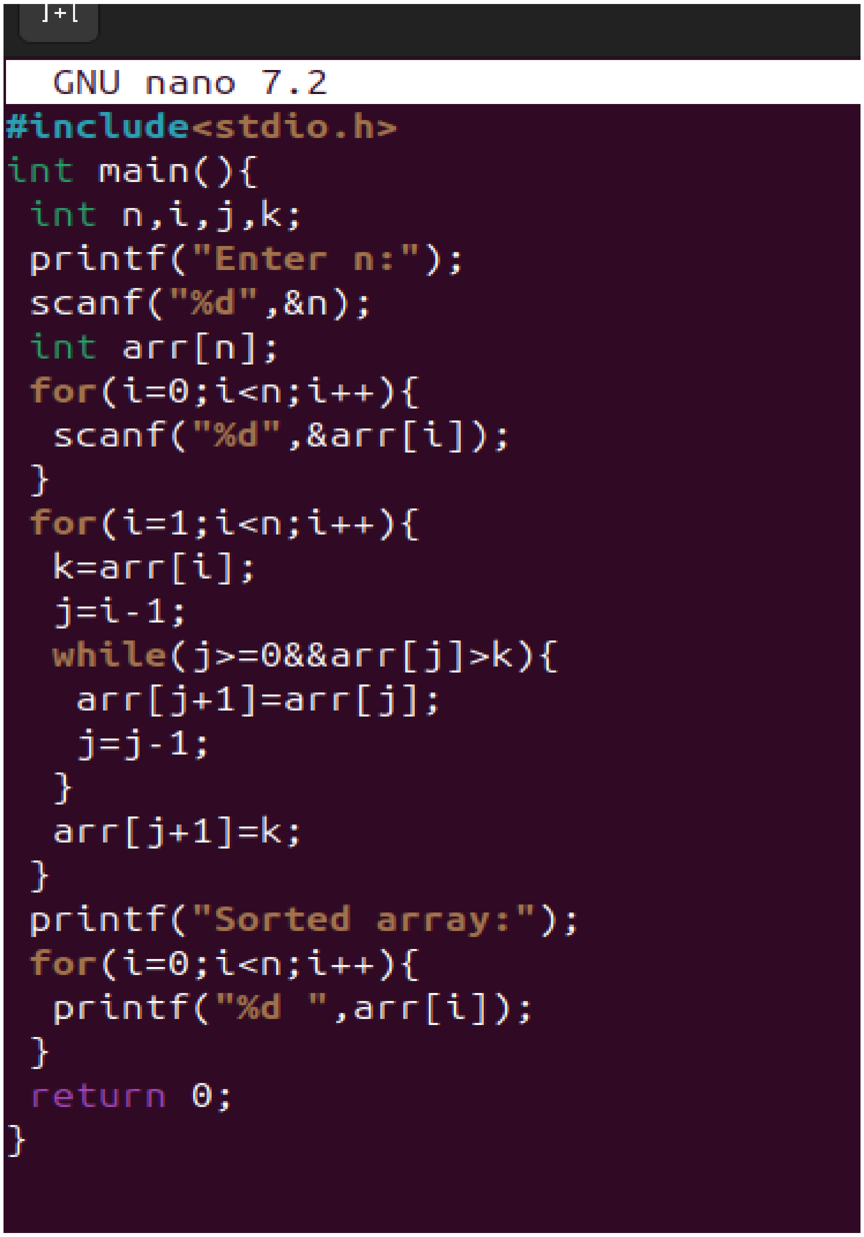
```
amma@amma05:~$ gcc SelectionSort.c -o SelectionSort
amma@amma05:~$ ./SelectionSort
Enter the number:6
Element 1 = 4
Element 2 = 9
Element 3 = 6
Element 4 = 7
Element 5 = 2
Element 6 = 30
Initially : 4 9 6 7 2 30
Finally : 30 30 30 30 30 30
amma@amma05:~$
```

Interpretation: Selects the minimum element from the unsorted part and places it at the correct position in each pass.

Complexity: Time – O(n^2) Space – O(1)

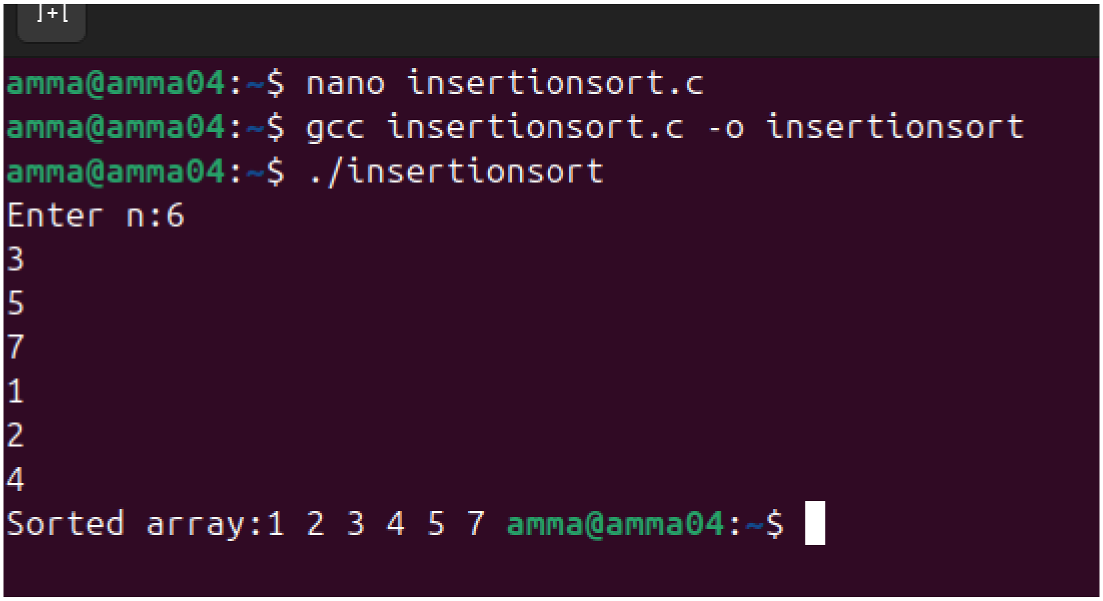
INSERTION SORT:

Code:



```
  GNU nano 7.2
#include<stdio.h>
int main(){
 int n,i,j,k;
 printf("Enter n:");
 scanf("%d",&n);
 int arr[n];
 for(i=0;i<n;i++){
  scanf("%d",&arr[i]);
 }
 for(i=1;i<n;i++){
  k=arr[i];
  j=i-1;
  while(j>=0&&arr[j]>k){
   arr[j+1]=arr[j];
   j=j-1;
  }
  arr[j+1]=k;
 }
 printf("Sorted array:");
 for(i=0;i<n;i++){
  printf("%d ",arr[i]);
 }
 return 0;
}
```

Output:



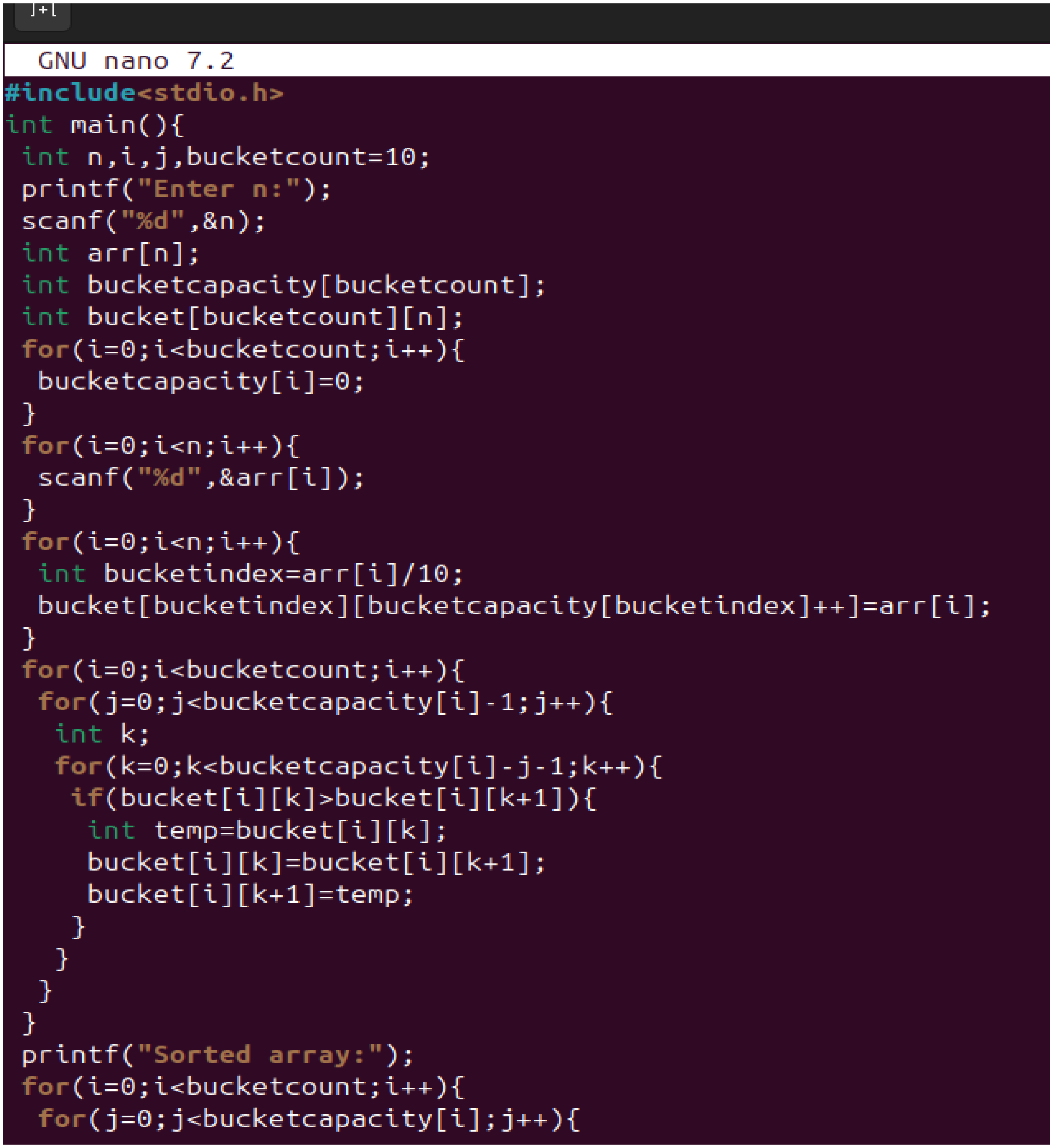
```
amma@amma04:~$ nano insertionsort.c
amma@amma04:~$ gcc insertionsort.c -o insertionsort
amma@amma04:~$ ./insertionsort
Enter n:6
3
5
7
1
2
4
Sorted array:1 2 3 4 5 7 amma@amma04:~$
```

Interpretation: Builds the sorted array one element at a time by inserting each element into its correct position.

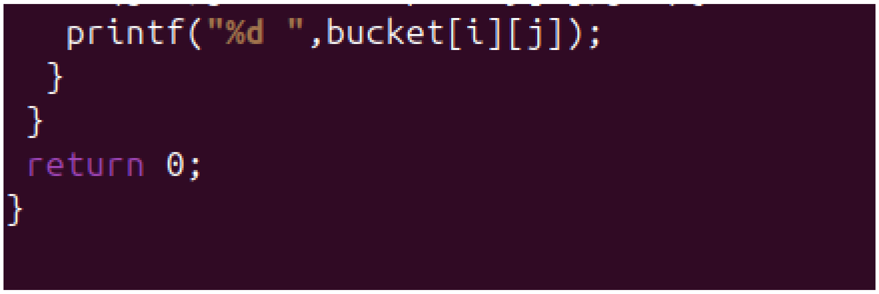
Complexity: Time – O(n^2) Space – O(1)

BUCKET SORT

CODE:

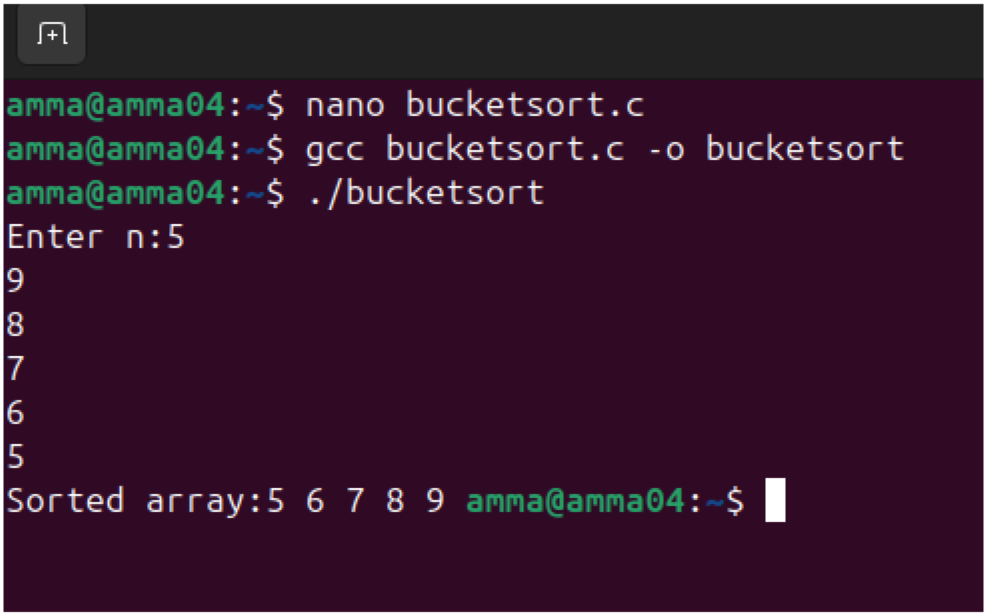


```
  GNU nano 7.2
#include<stdio.h>
int main(){
 int n,i,j,bucketcount=10;
 printf("Enter n:");
 scanf("%d",&n);
 int arr[n];
 int bucketcapacity[bucketcount];
 int bucket[bucketcount][n];
 for(i=0;i<bucketcount;i++){
  bucketcapacity[i]=0;
 }
 for(i=0;i<n;i++){
  scanf("%d",&arr[i]);
 }
 for(i=0;i<n;i++){
  int bucketindex=arr[i]/10;
  bucket[bucketindex][bucketcapacity[bucketindex]++]=arr[i];
 }
 for(i=0;i<bucketcount;i++){
  for(j=0;j<bucketcapacity[i]-1;j++){
   int k;
   for(k=0;k<bucketcapacity[i]-j-1;k++){
    if(bucket[i][k]>bucket[i][k+1]){
     int temp=bucket[i][k];
     bucket[i][k]=bucket[i][k+1];
     bucket[i][k+1]=temp;
    }
   }
  }
 }
 printf("Sorted array:");
 for(i=0;i<bucketcount;i++){
  for(j=0;j<bucketcapacity[i];j++){
```



```
   printf("%d ",bucket[i][j]);
  }
 }
 return 0;
}
```

OUTPUT:



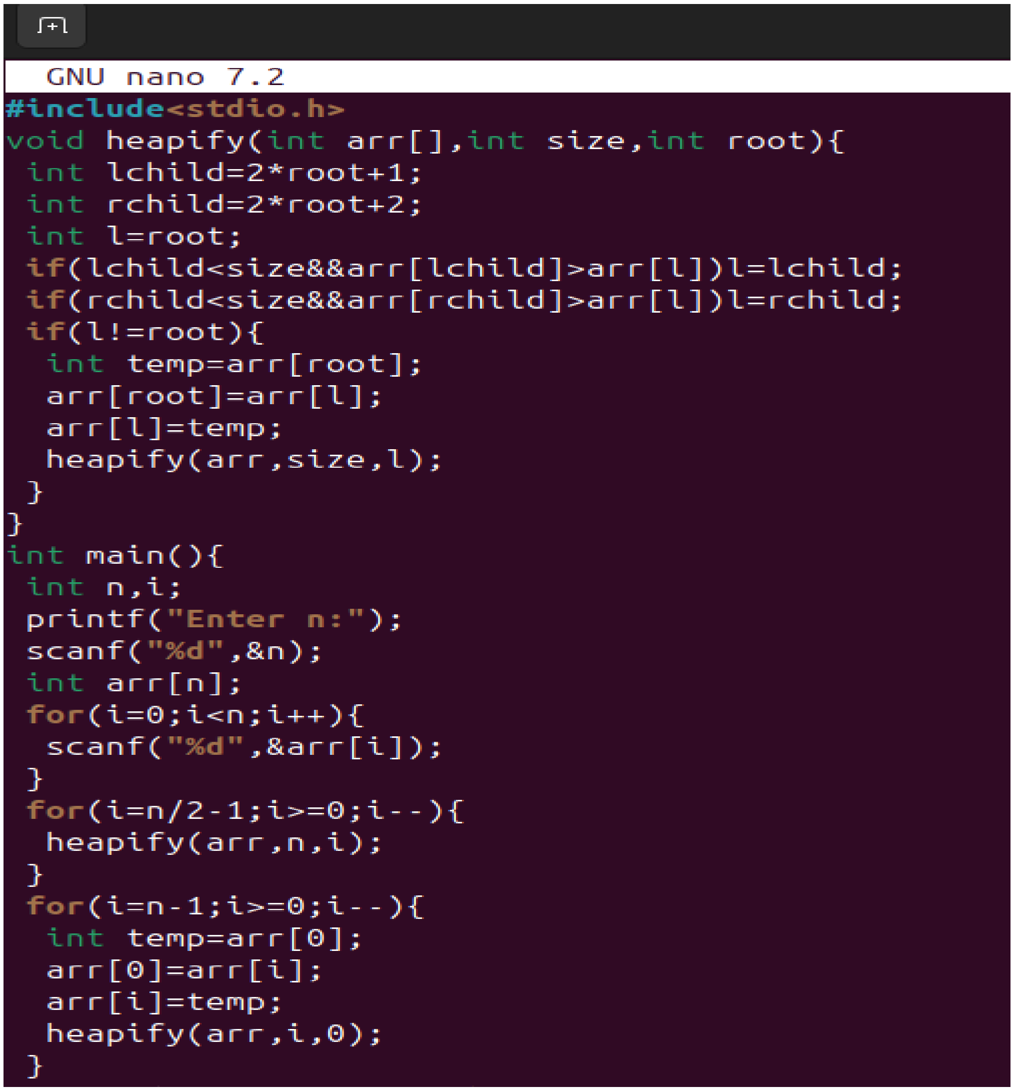
```
amma@amma04:~$ nano bucketsort.c
amma@amma04:~$ gcc bucketsort.c -o bucketsort
amma@amma04:~$ ./bucketsort
Enter n:5
9
8
7
6
5
Sorted array:5 6 7 8 9 amma@amma04:~$ 
```

Interpretation: Distributes elements into buckets, sorts each bucket, and then concatenates them to form the sorted array.

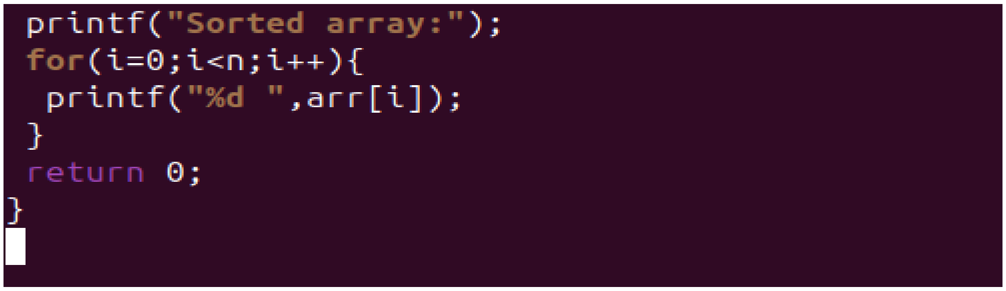
Complexity: Time – O(n^2) Space – O(n + k).

HEAP SORT

CODE:

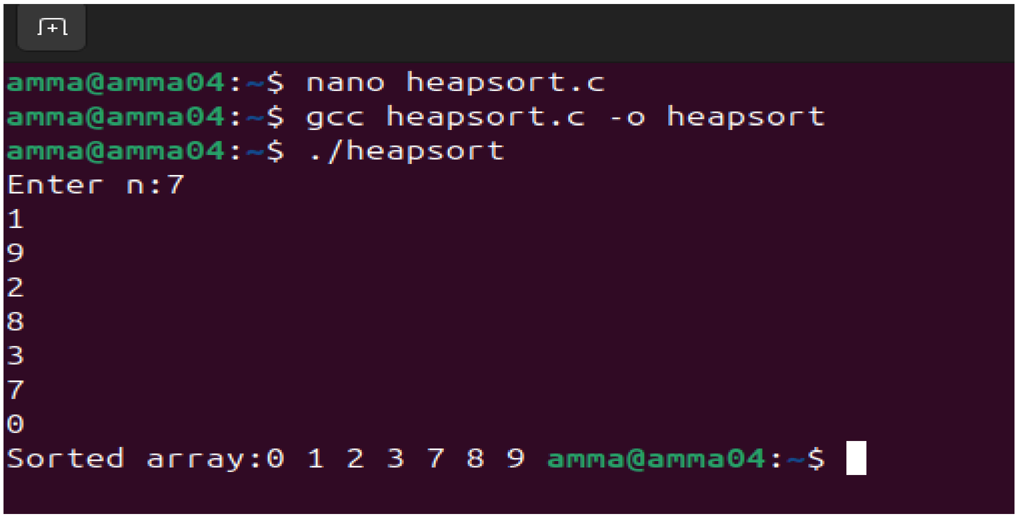


```
#include<stdio.h>
void heapify(int arr[],int size,int root){
 int lchild=2*root+1;
 int rchild=2*root+2;
 int l=root;
 if(lchild<size&&arr[lchild]>arr[l])l=lchild;
 if(rchild<size&&arr[rchild]>arr[l])l=rchild;
 if(l!=root){
  int temp=arr[root];
  arr[root]=arr[l];
  arr[l]=temp;
  heapify(arr,size,l);
 }
}
int main(){
 int n,i;
 printf("Enter n:");
 scanf("%d",&n);
 int arr[n];
 for(i=0;i<n;i++){
  scanf("%d",&arr[i]);
 }
 for(i=n/2-1;i>=0;i--){
  heapify(arr,n,i);
 }
 for(i=n-1;i>=0;i--){
  int temp=arr[0];
  arr[0]=arr[i];
  arr[i]=temp;
  heapify(arr,i,0);
 }
```



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

OUTPUT:

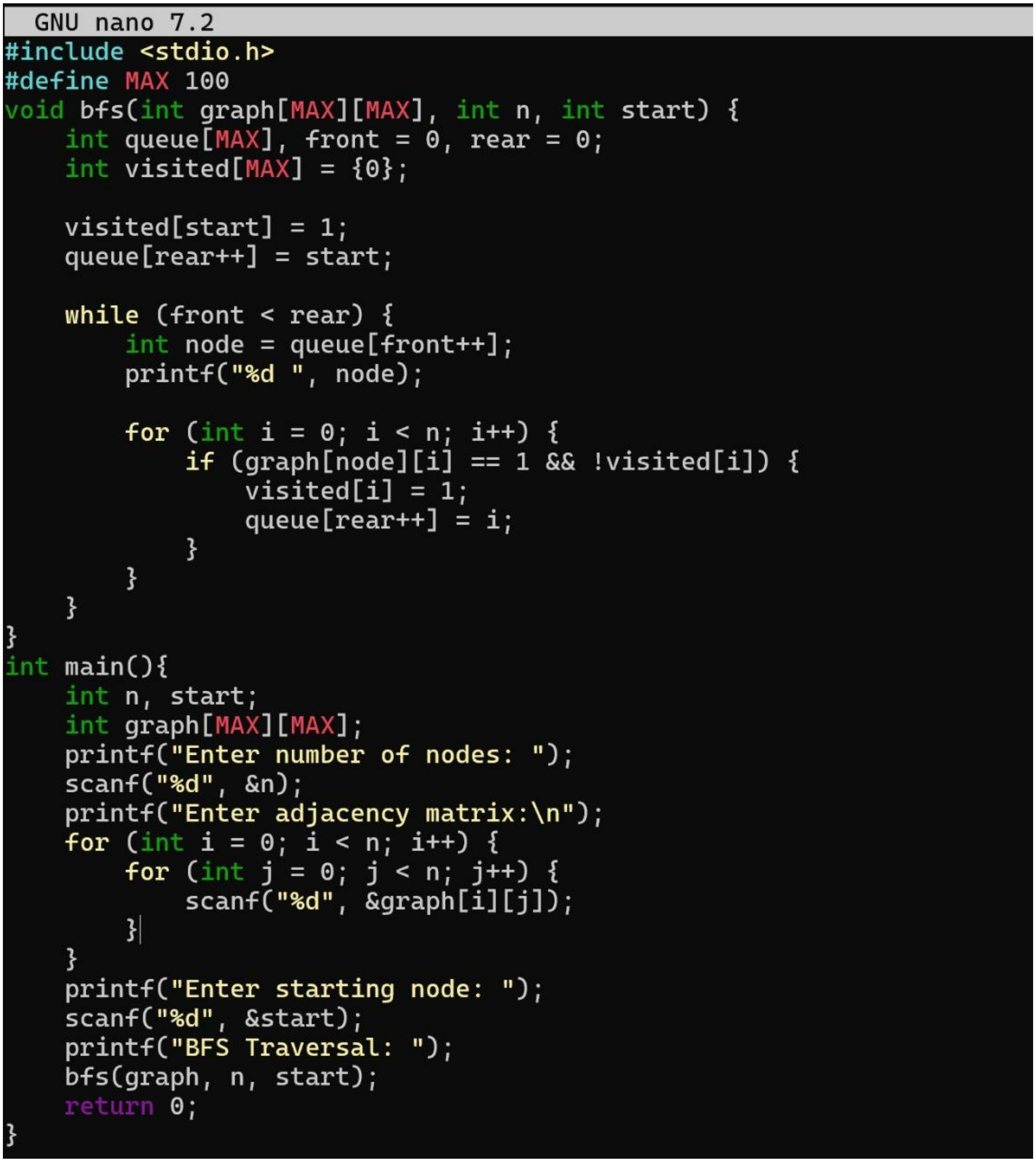


```
amma@amma04:~$ nano heapsort.c
amma@amma04:~$ gcc heapsort.c -o heapsort
amma@amma04:~$ ./heapsort
Enter n:7
1
9
2
8
3
7
0
Sorted array:0 1 2 3 7 8 9 amma@amma04:~$ 
```

Interpretation: Uses a max-heap to repeatedly extract the largest element and place it at the end of the array.

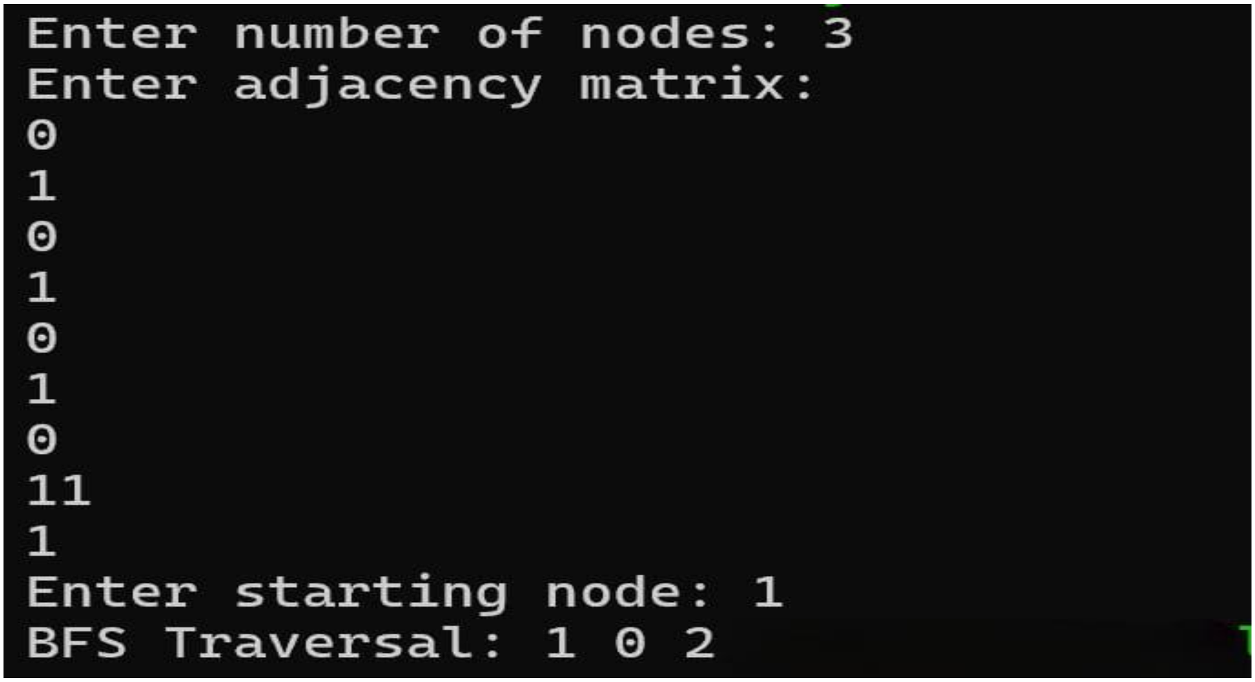
Complexity: Time – O(n log n)   Space – O(1).

BFS :



```c
#include <stdio.h>
#define MAX 100
void bfs(int graph[MAX][MAX], int n, int start) {
    int queue[MAX], front = 0, rear = 0;
    int visited[MAX] = {0};

    visited[start] = 1;
    queue[rear++] = start;

    while (front < rear) {
        int node = queue[front++];
        printf("%d ", node);

        for (int i = 0; i < n; i++) {
            if (graph[node][i] == 1 && !visited[i]) {
                visited[i] = 1;
                queue[rear++] = i;
            }
        }
    }
}
int main(){
    int n, start;
    int graph[MAX][MAX];
    printf("Enter number of nodes: ");
    scanf("%d", &n);
    printf("Enter adjacency matrix:\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            scanf("%d", &graph[i][j]);
        }
    }
    printf("Enter starting node: ");
    scanf("%d", &start);
    printf("BFS Traversal: ");
    bfs(graph, n, start);
    return 0;
}
```

OUTPUT :



```
Enter number of nodes: 3
Enter adjacency matrix:
0
1
0
1
0
1
0
11
1
Enter starting node: 1
BFS Traversal: 1 0 2
```

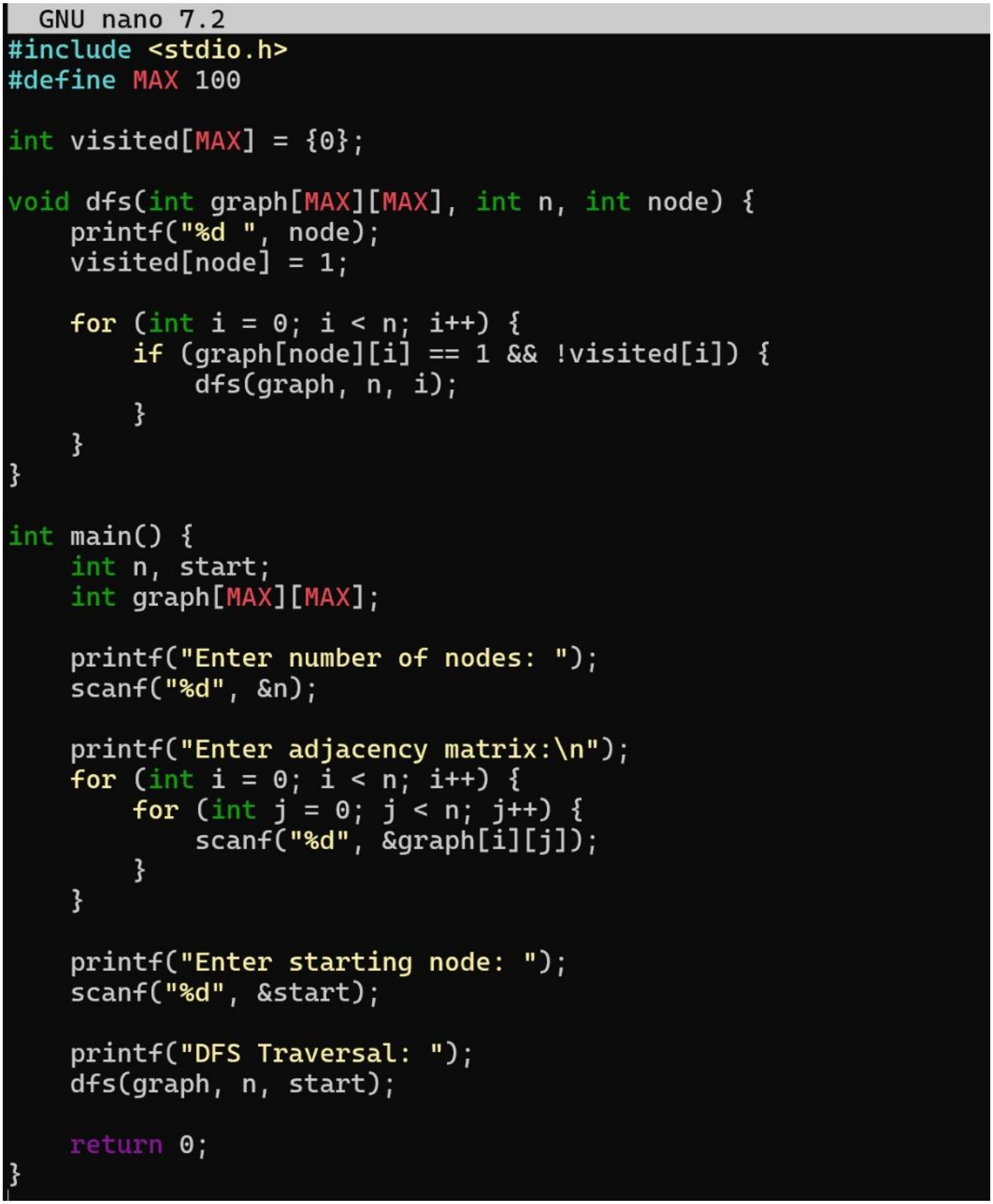
Interpretation

Breadth First Search traverses a graph level by level, visiting all adjacent nodes of a vertex before moving to the next level.

It uses a queue to ensure nodes are explored in the order they are discovered.

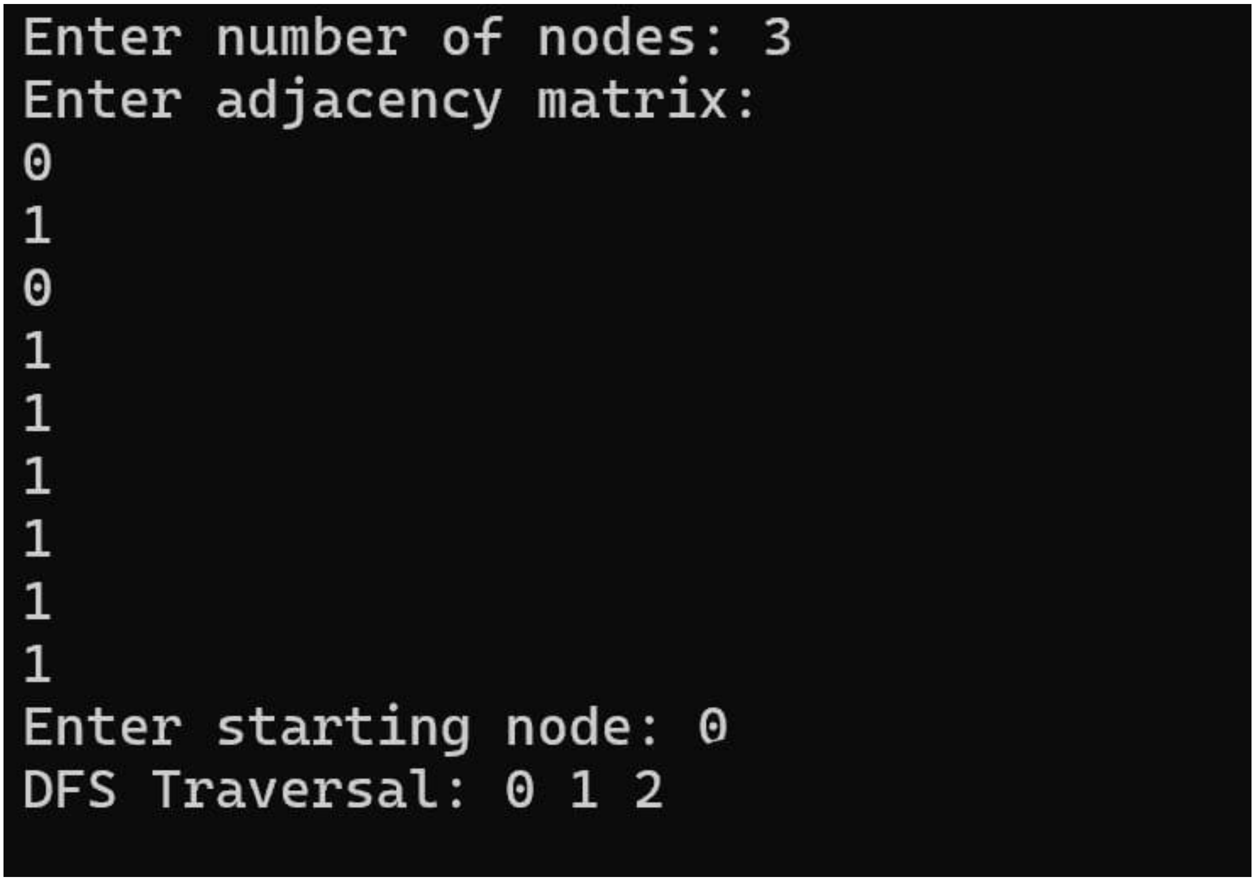
Time Complexity: $O(n^2)$ Space Complexity: $O(n)$

DFS :



```c
#include <stdio.h>
#define MAX 100

int visited[MAX] = {0};

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

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

int main() {
    int n, start;
    int graph[MAX][MAX];

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

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

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

    printf("DFS Traversal: ");
    dfs(graph, n, start);

    return 0;
}
```

OUTPUT :



```
Enter number of nodes: 3
Enter adjacency matrix:
0
1
0
1
1
1
1
1
1
Enter starting node: 0
DFS Traversal: 0 1 2
```

Depth First Search explores a graph by going as deep as possible along each branch before backtracking.
It uses recursion (or stack) to visit unvisited adjacent nodes starting from a given node.

Time Complexity O(n^2)   Space Complexity O(n)