Assignment Topic: Implement the master method theorem, the BFS algorithm, the DFS Algorithm, and the Topological sort algorithm

Course Name: Design and Analysis of Algorithms

Course Code: CSE373

Section: 10

Faculty: Emon Kumar Dey (EKD)

Date of submission: 28th May, 2024

Submitted By:

Name: Mohammad Hossain

ID-2131770042

Implementation of the master method theorem

What is the Master theorem?

Master’s theorem is one way to find an algorithm's time complexity. It offers a quick way to figure out the time complexity of algorithms that follow a specific recursion pattern. This theorem can be applied only to decreasing and dividing functions.

Master theorem’s Algorithm for dividing functions:

Master theorem’s Algorithm for dividing functions can only be applied to the recurrence relations of the form:

$$(n) = (n/b)+f(n) \,, \qquad \text{where } f(\mathrm{n}) = \theta(n^k log^p n)$$

Here,

n = input size

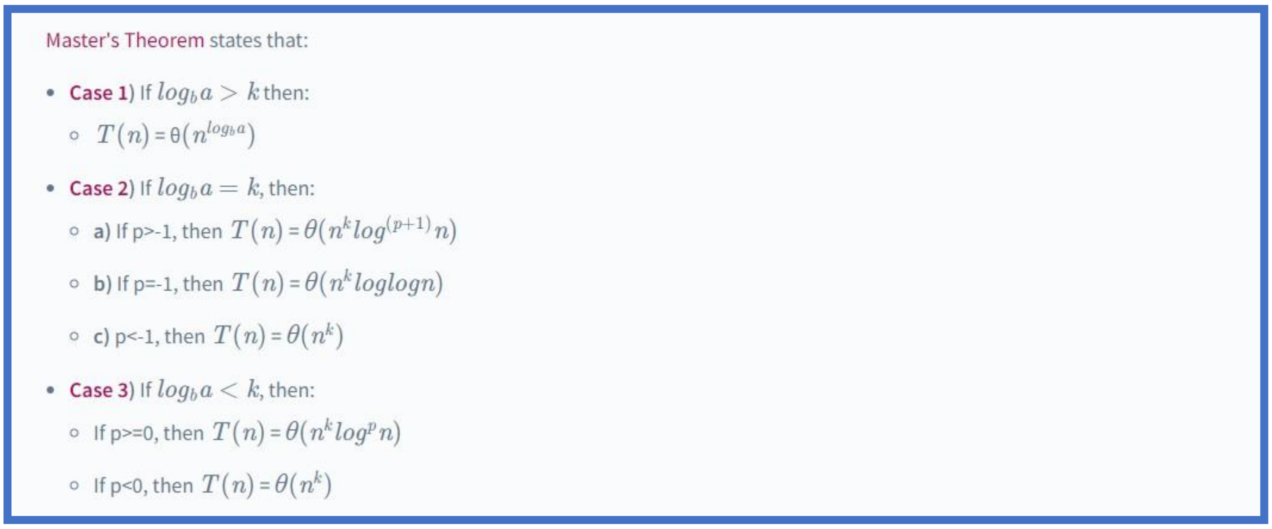
a = number of subproblems in the recursion

n/b = size of each subproblem

Where the values of a, b, and k are fixed and must meet these specific rules:

a>=1, b>1, and k>=0

To find time complexity, first, we need to figure out the values of 'a,' 'b,' and 'k' using the function (n). Then, $log_b\ a = \frac{loga}{logb}$; this equation helps us find the value of $log_b\ a$. Compare $log_b\ a$ and k to decide which case will be applied to the individual algorithm to find time complexity. There are three cases. These three cases are provided here, and we can use them to figure out how long any algorithm takes to run,



Master's Theorem states that:

- Case 1) If $log_b a > k$ then:
  - $T(n) = \theta(n^{log_b a})$
- Case 2) If $log_b a = k$, then:
  - a) If p>-1, then $T(n) = \theta(n^k log^{(p+1)} n)$
  - b) If p=-1, then $T(n) = \theta(n^k loglogn)$
  - c) p<-1, then $T(n) = \theta(n^k)$
- Case 3) If $log_b a < k$, then:
  - If p>=0, then $T(n) = \theta(n^k log^p n)$
  - If p<0, then $T(n) = \theta(n^k)$

Master theorem's cases

Although the Master theorem can be applied to decreasing and dividing functions, we only implemented the code for dividing functions as the instruction of faculty, and these cases are only for dividing functions.

Code implementation of Master theorem:

```
#include <stdio.h>

#include <math.h>
```

```
int main(){     char
T[50];     int a, b, k,
p, n, i;     float
logResult;


   printf("Enter T(n)(enter '!' to end the line): ");
for(i=0;i<50;i++){         scanf("%c",&T[i]);
if(T[i]=='!')             break;
   }     a = T[0]-48;     b =
T[5]-48;     logResult =
log(a)/log(b);


   if(T[8]=='n' & T[11]=='l'){
if(T[9]=='^')          k = T[10]-
48;     if(T[14]=='^')          p
= T[15]-48;         else          p
= 1;
   }
   else if(T[8]=='n' && T[9]=='l'){
      k = 1;
if(T[12]=='^')          p =
T[13]-48;         else
p = 1;     }     else
if(T[8]=='l'){         k=0;
if(T[11]=='^')          p =
T[12]-48;         else
p = 1;
```

```
   }     else
if(T[8]=='n'){
p=0;
if(T[9]=='^')          k
= T[10]-48;
else          k=1;
}

   if(logResult>k){                          //case 1 check
printf("\nAccording to case 1: logb(a)>k\n");

       if(logResult>1 || logResult<0)
printf("T(n) = O(n^log%d(%d))",b,a);

       else if(logResult==0)
printf("T(n) = O(1)");         else
if(logResult==1)
printf("T(n) = O(n)");
   }

   else if(logResult<k){                     //case 3 check
printf("\nAccording to case 3: logb(a)<k and ");

       if(p>=0){
printf("p>=0\n");
```

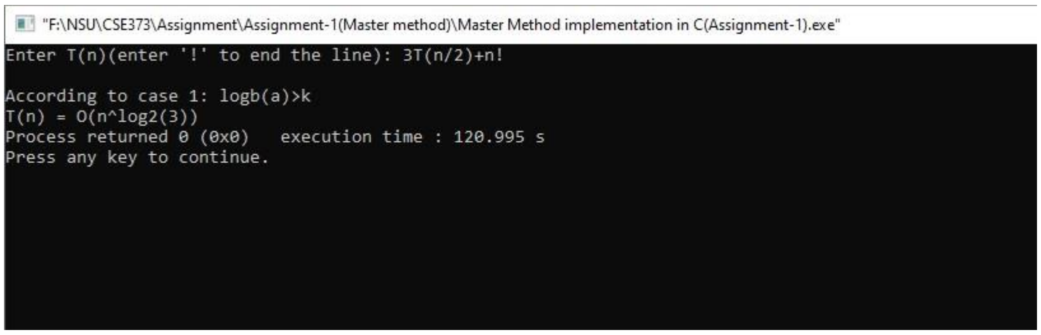
```
     if(k>1 || k<0){              if(p==0)
printf("T(n) = O(n^%d)",k);


      else if(p==1)                printf("T(n) =
0((n^%d)(log(n)))",k);              else if(p>1)
printf("T(n) = O((n^%d)(log^%d(n)))",k,p);
           }


           if(k==0){
if(p==0)                printf("T(n)
= O(1)");


             else if(p==1)
printf("T(n) = 0(log(n))");
else if(p>1)                printf("T(n) =
O(log^%d(n))",k);
           }
if(k==1){
              if(p==0)
printf("T(n) = O(n)");


             else if(p==1)
printf("T(n) = 0(nlog(n))");
              else if(p>1)                printf("T(n)
= O(nlog^%d(n))",k);
```

```
            }         }
else{
printf("p<0\n");

            if(k>1 || k<0)
printf("T(n) = O(n^%d)",k);

            else if(k=0)
printf("T(n) = O(1)");

            else if(k=1)
printf("T(n) = O(n)");         }

    }

    else{         printf("\nAccording to case 2:
logb(a)=k and ");

        if(p>-1){
printf("p>-1\n");

            if(p+1==1){
if(k==1)                    printf("T(n) =
O(nlogn)");

                else if(k>1 || k<0)
printf("T(n) = O((n^%d)logn)",k);
```

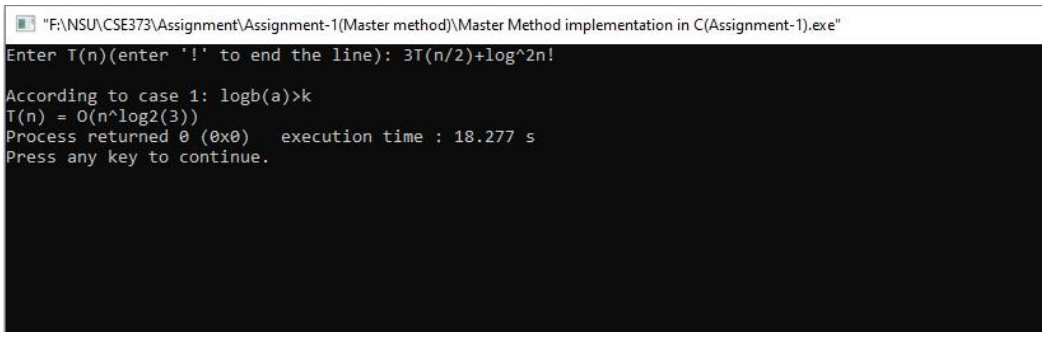
```
               else if(k==0)
printf("T(n) = O(logn)");
            }


            else if(p+1==0){
if(k==1)                     printf("T(n)
= O(n)");


               else if(k>1 || k<0)
printf("T(n) = O(n^%d)",k);
else if(k==0)                     printf("T(n) =
O(1)");
            }


            else if((p+1)>1 || (p+1)<0){
if(k==1)                     printf("T(n) =
O(nlog^%d(n))",(p+1));


               else if(k>1 || k<0)                     printf("T(n) =
O((n^%d)log^%d(n))",k,(p+1));


               else if(k==0)                     printf("T(n)
= O(log^%d(n))",(p+1));
            }
         }     }
return 0;
```

}

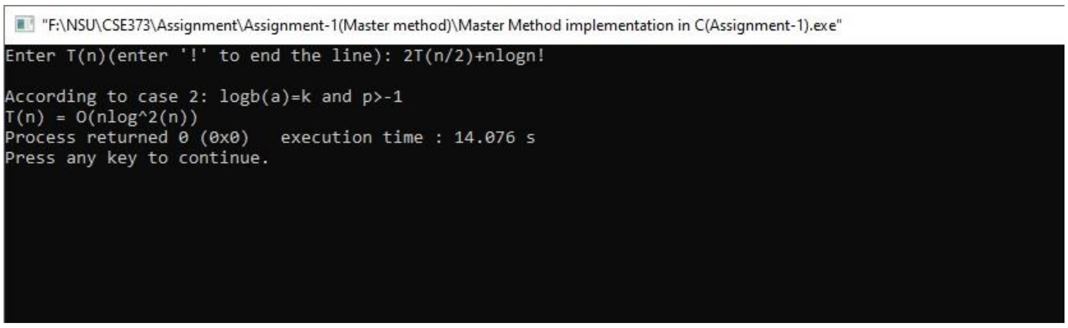
Inputs and Outputs:



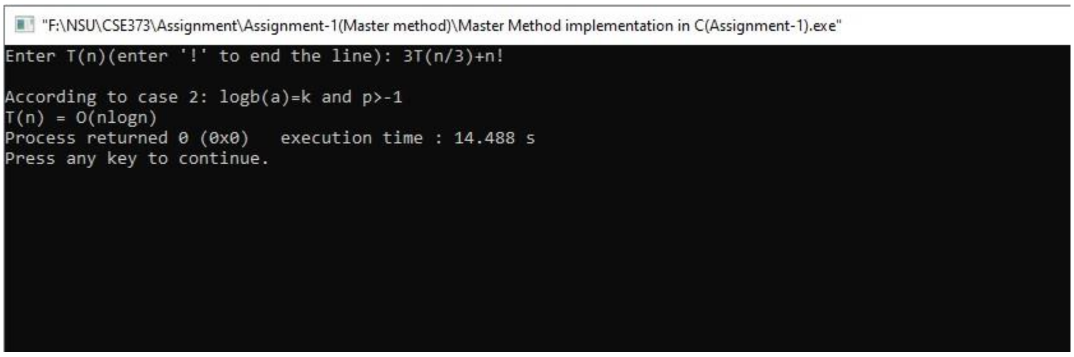
```
"F:\NSU\CSE373\Assignment\Assignment-1(Master method)\Master Method implementation in C(Assignment-1).exe"
Enter T(n)(enter '!' to end the line): 3T(n/2)+n!

According to case 1: logb(a)>k
T(n) = O(n^log2(3))
Process returned 0 (0x0)   execution time : 120.995 s
Press any key to continue.
```



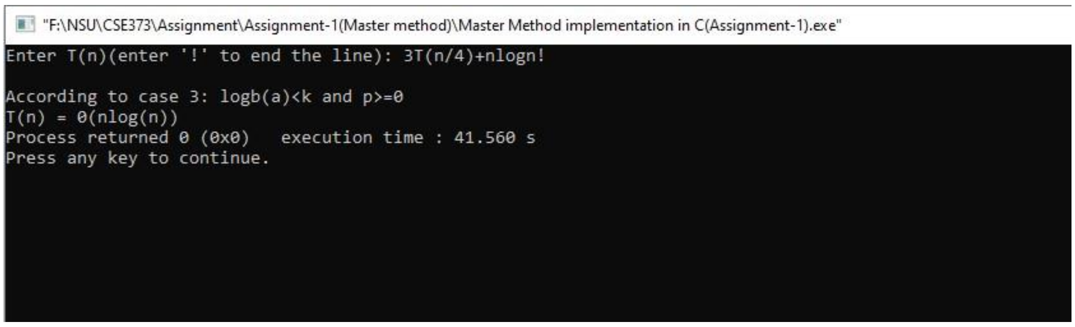
```
"F:\NSU\CSE373\Assignment\Assignment-1(Master method)\Master Method implementation in C(Assignment-1).exe"
Enter T(n)(enter '!' to end the line): 3T(n/2)+log^2n!

According to case 1: logb(a)>k
T(n) = O(n^log2(3))
Process returned 0 (0x0)   execution time : 18.277 s
Press any key to continue.
```



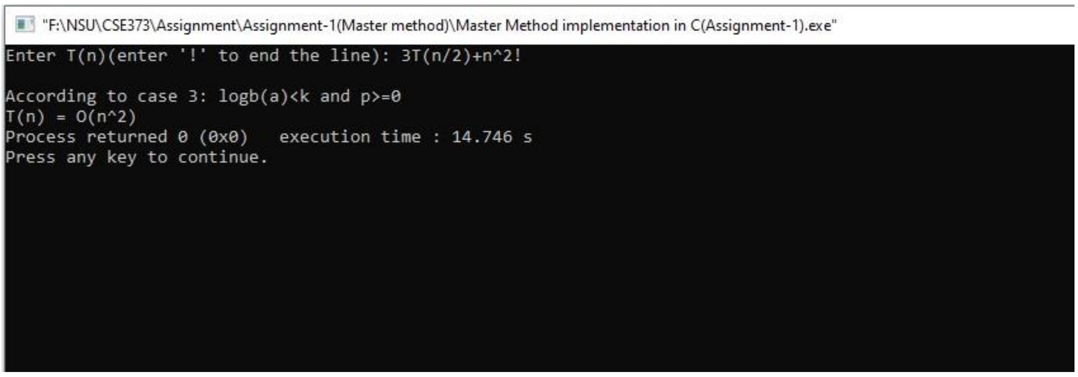
```
"F:\NSU\CSE373\Assignment\Assignment-1(Master method)\Master Method implementation in C(Assignment-1).exe"
Enter T(n)(enter '!' to end the line): 2T(n/2)+nlogn!

According to case 2: logb(a)=k and p>-1
T(n) = O(nlog^2(n))
Process returned 0 (0x0)   execution time : 14.076 s
Press any key to continue.
```



```
"F:\NSU\CSE373\Assignment\Assignment-1(Master method)\Master Method implementation in C(Assignment-1).exe"
Enter T(n)(enter '!' to end the line): 3T(n/3)+n!

According to case 2: logb(a)=k and p>-1
T(n) = O(nlogn)
Process returned 0 (0x0)   execution time : 14.488 s
Press any key to continue.
```



```
"F:\NSU\CSE373\Assignment\Assignment-1(Master method)\Master Method implementation in C(Assignment-1).exe"
Enter T(n)(enter '!' to end the line): 3T(n/4)+nlogn!

According to case 3: logb(a)<k and p>=0
T(n) = 0(nlog(n))
Process returned 0 (0x0)   execution time : 41.560 s
Press any key to continue.
```



```
"F:\NSU\CSE373\Assignment\Assignment-1(Master method)\Master Method implementation in C(Assignment-1).exe"
Enter T(n)(enter '!' to end the line): 3T(n/2)+n^2!

According to case 3: logb(a)<k and p>=0
T(n) = O(n^2)
Process returned 0 (0x0)   execution time : 14.746 s
Press any key to continue.
```

Benefits of the master theorem:

- By using the Master Theorem, one may quickly determine the time complexity of specific recursive algorithm types without checking a detailed recurrence tree or substitution method.

- It provides a direct formula for calculating time complexity, particularly useful when working with algorithms that fit its pattern.

Limitations of the master theorem:

- The Master Theorem only applies to a recurrence relation known as "divide and conquer," in which the problem is divided into equal-sized subproblems.

- It is only applicable in cases where the sizes of the subproblems are equal, and there is a pattern of work completed at each stage of recursion. You cannot apply the Master Theorem to your problem if it does not meet these requirements.

Implementation of the BFS Algorithm

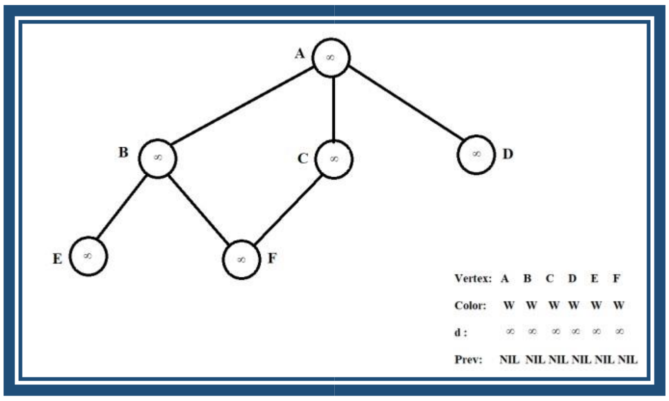
What is the BFS algorithm?

Breadth First Search (BFS) is a graph traversal algorithm that explores all the vertices in a graph at the current depth level before moving on to the vertices at the next depth level. It starts at a specified vertex, called the root, and visits all its neighbor's vertices before moving on to the next level of neighbor's vertices. BFS is commonly used in algorithms for pathfinding, connected components, and shortest path problems in graphs. BFS built a tree for a connected graph and might also make a forest if the graph is not connected. Also, BFS is applied on both weighted and unweighted graphs.

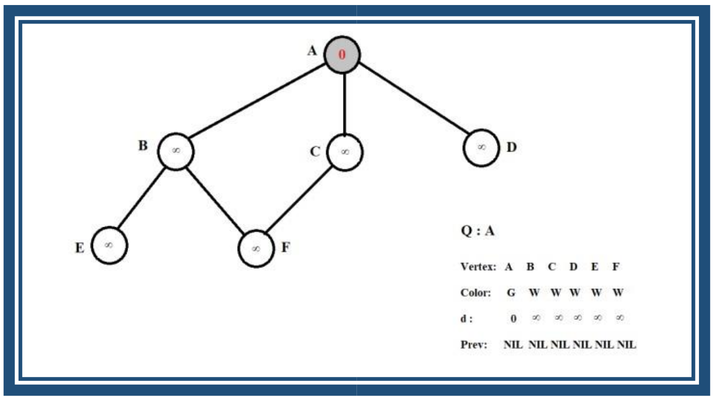
How does the BFS algorithm work?

At the beginning of the BFS function, we have to initialize the color of all vertices to white, previous to NULL, and distance to infinite. Then, initialize the root vertex to grey color, distance to 0, previous to NULL, and create a queue, and then the function enqueue the root vertex. After that, a while loop will be started, continuing until the queue is empty. In this loop, a function called dequeue will be called, and the value of dequeue will be stored in the variable called u. Then, a for loop will be started that will continue until the check of all adjacents of u. In a for loop, an if condition will check whether the adjacent color of u is white. If the color is white, then the color of that adjacent will be colored grey, the distance will be the distance of u +1, and the previous of that adjacent will be set to u. In the end, the color of every vertices will be set to black.

Let's now understand how the BFS algorithm works with a simulation,

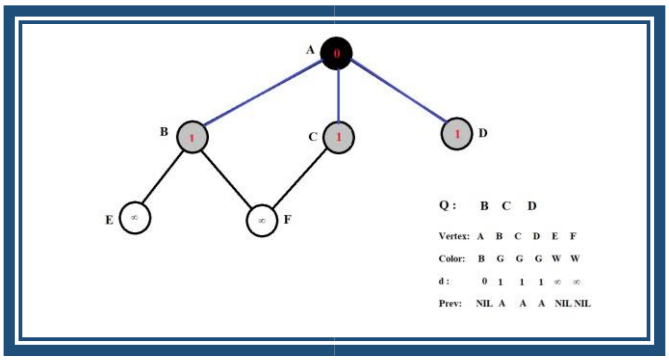


Vertex:	A	B	C	D	E	F
Color:	W	W	W	W	W	W
d :	∞	∞	∞	∞	∞	∞
Prev:	NIL	NIL	NIL	NIL	NIL	NIL



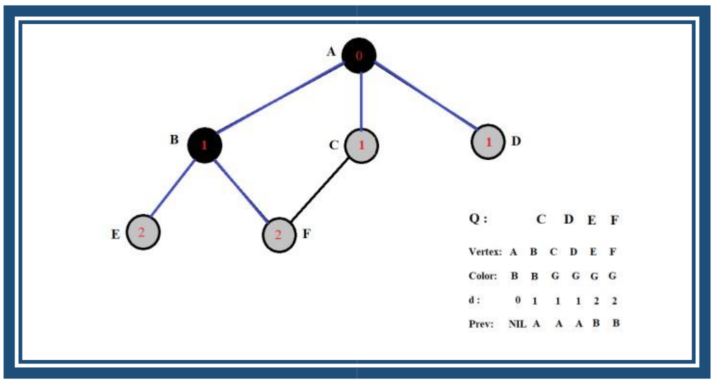
Q : A

Vertex:	A	B	C	D	E	F
Color:	G	W	W	W	W	W
d :	0	∞	∞	∞	∞	∞
Prev:	NIL	NIL	NIL	NIL	NIL	NIL



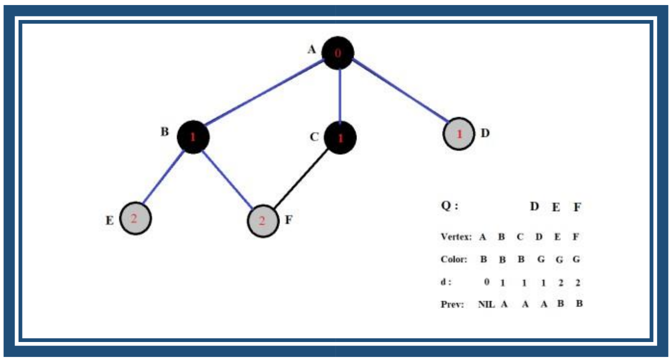
Q : B C D

Vertex:	A	B	C	D	E	F
Color:	B	G	G	G	W	W
d :	0	1	1	1	∞	∞
Prev:	NIL	A	A	A	NIL	NIL



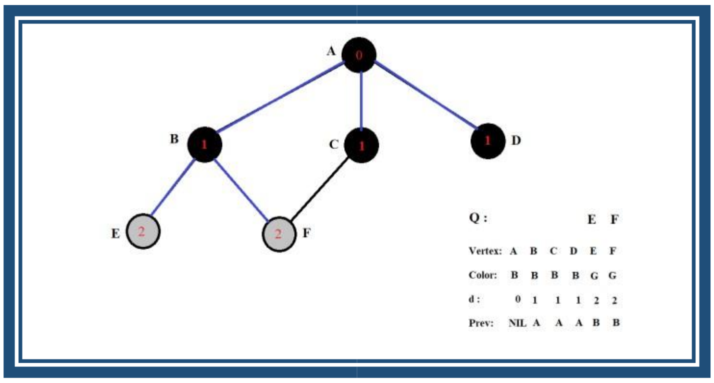
Q : C D E F

Vertex:	A	B	C	D	E	F
Color:	B	B	G	G	G	G
d :	0	1	1	1	2	2
Prev:	NIL	A	A	A	B	B



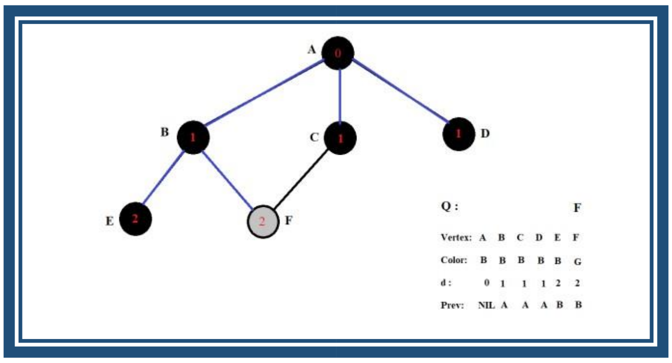
Q : D E F

Vertex:	A	B	C	D	E	F
Color:	B	B	B	G	G	G
d :	0	1	1	1	2	2
Prev:	NIL	A	A	A	B	B



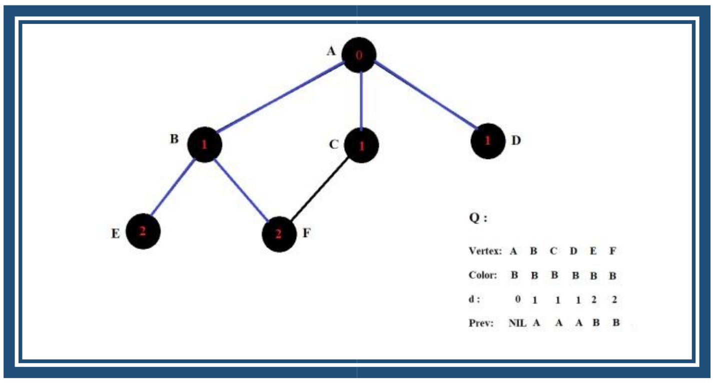
Q : E F

Vertex:	A	B	C	D	E	F
Color:	B	B	B	B	G	G
d :	0	1	1	1	2	2
Prev:	NIL	A	A	A	B	B



Q : F

Vertex:	A	B	C	D	E	F
Color:	B	B	B	B	B	G
d :	0	1	1	1	2	2
Prev:	NIL	A	A	A	B	B



Q :

Vertex:	A	B	C	D	E	F
Color:	B	B	B	B	B	B
d :	0	1	1	1	2	2
Prev:	NIL	A	A	A	B	B

Pseudocode of BFS algorithm:

```
BFS(G) // starts from here
```

```
{    for each vertex u  V-
{s}
  {
    color[u]=WHITE;
                    prev[u]=NIL;
          d[u]=inf;
  }
  color[s]=GRAY;
          d[s]=0;
      prev[s]=NIL;
          Q=empty;
          ENQUEUE(Q,s);

While(Q not empty)
 {
  u = DEQUEUE(Q);

 for each v  adj[u]{      if
(color[v] == WHITE){
color[v] = GREY;
d[v] = d[u] + 1;
prev[v] = u;
      Enqueue(Q, v);
   }
  }
  color[u] = BLACK;
 }
}
```

Code implementation of the BFS algorithm:

```
#include <iostream>

#include <bits/stdc++.h>

#include "Queue.h"

#include "Queue.cpp" using

namespace std;


void BFS(int **AdjMat,char vertex[],int u){

   int i,j;

   char color[u];

char prev[u];     int

d[u];


   for(i=1; i<u; i++){                                   //root = AdjMat[0]

color[i] = 'W';                                  //White = W

prev[i] = 'N';                                  //NULL = N        d[i] =

5000;                                   //Inf = 5000

   }     color[0]

= 'G';     d[0] =

0;     prev[0] =

'N';
```

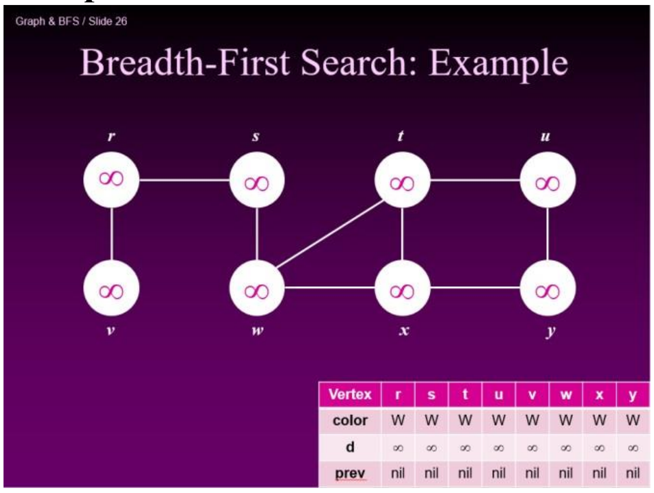
```
   Queue<char> Q(u);
Q.Enqueue(vertex[0]);
while(!Q.IsEmpty()){
       char p;
       Q.Dequeue(p);
int k = 0;
while(vertex[k]!=p){
           k++;
       }


       for(j=0; j<u; j++){
if(AdjMat[k][j]==1 && color[j]=='W'){
              color[j] = 'G';
d[j] = d[k]+1;
prev[j] = p;

Q.Enqueue(vertex[j]);
           }        }        color[k] = 'B';
//Black = B
   }


   cout<<"The result is : "<<endl;
cout<<endl;     cout<<"Vertex: ";
for(i=0; i<u; i++){
cout<<vertex[i]<<" ";

   }
   cout<<endl;
cout<<"Color:  ";


   for(i=0; i<u; i++){
cout<<color[i]<<" ";
   }
   cout<<endl;
cout<<"d:      ";


   for(i=0; i<u; i++){
cout<<d[i]<<" ";
   }
   cout<<endl;
cout<<"Prev:   ";


   for(i=0; i<u; i++){
cout<<prev[i]<<" ";
   }
}


int main()
{     int
a,b;     int
i,j;
   cout<<"Enter the total number of vertices and edges: ";         cin>>a>>b;
             char vertices[a];
cout<<"Enter the vertex: ";
```

```
for(i=0; i<a; i++){
cin>>vertices[i]; }  int
**AdjMatrix;
AdjMatrix = new int*[a];


for(i=0; i<a; i++){
      AdjMatrix[i] = new int[a];
} for(i=0; i<a; i++) {
for(j=0; j<a; j++) {
     AdjMatrix[i][j] = 0;
        }
}


for (i=0; i<b; i++) {
char x,y;         int
m=0,n=0;
cout<<"Enter edges: ";
cin>>x>>y;


     while(vertices[m]!=x){
m++;
}
while(vertices[n]!=y){
          n++;
}
```

```
          AdjMatrix[m][n] = 1;

AdjMatrix[n][m] = 1;



}

cout<<endl;

BFS(AdjMatrix,vertices,a);

return 0;

}
```

Inputs and Outputs:

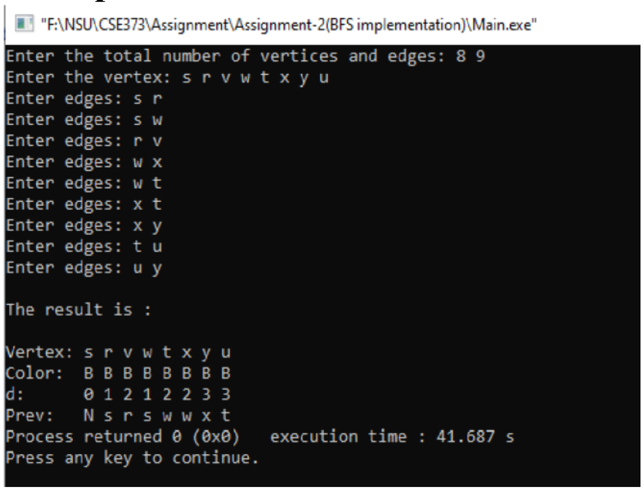
Input:



Breadth-First Search: Example

Vertex	r	s	t	u	v	w	x	y
color	W	W	W	W	W	W	W	W
d	∞	∞	∞	∞	∞	∞	∞	∞
prev	nil	nil	nil	nil	nil	nil	nil	nil

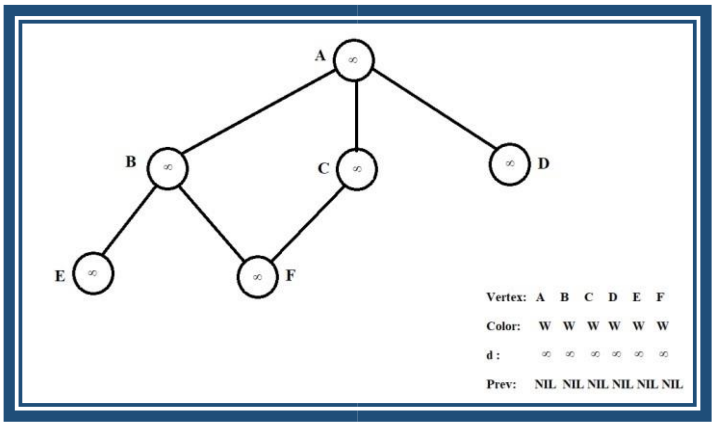
Output:

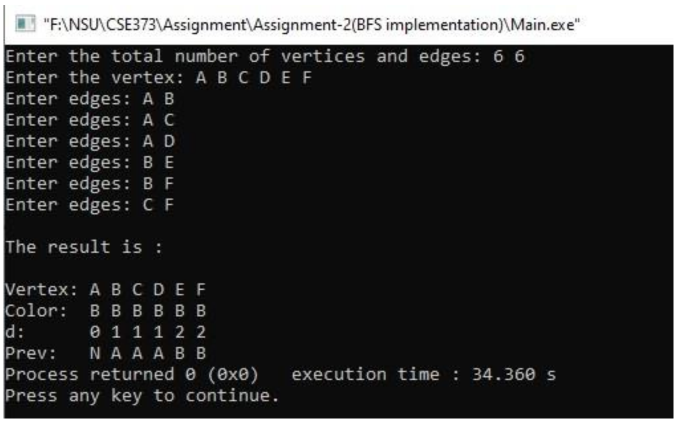


```
"F:\NSU\CSE373\Assignment\Assignment-2(BFS implementation)\Main.exe"
Enter the total number of vertices and edges: 8 9
Enter the vertex: s r v w t x y u
Enter edges: s r
Enter edges: s w
Enter edges: r v
Enter edges: w x
Enter edges: w t
Enter edges: x t
Enter edges: x y
Enter edges: t u
Enter edges: u y

The result is :

Vertex: s r v w t x y u
Color:  B B B B B B B B
d:      0 1 2 1 2 2 3 3
Prev:   N s r s w w x t
Process returned 0 (0x0)   execution time : 41.687 s
Press any key to continue.
```

Input:

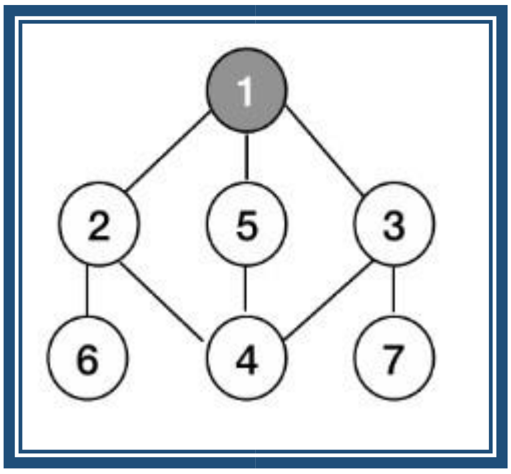
Output:

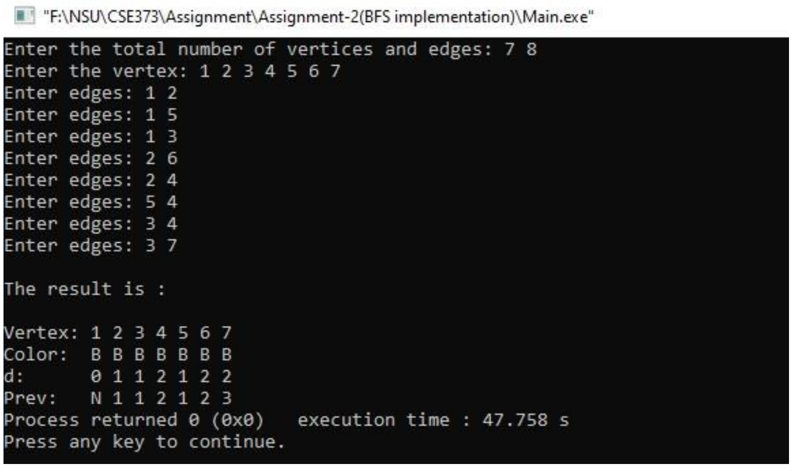




```
"F:\NSU\CSE373\Assignment\Assignment-2(BFS implementation)\Main.exe"
Enter the total number of vertices and edges: 6 6
Enter the vertex: A B C D E F
Enter edges: A B
Enter edges: A C
Enter edges: A D
Enter edges: B E
Enter edges: B F
Enter edges: C F

The result is :

Vertex: A B C D E F
Color:  B B B B B B
d:      0 1 1 1 2 2
Prev:   N A A A B B
Process returned 0 (0x0)   execution time : 34.360 s
Press any key to continue.
```

Input: **Output:**





```
"F:\NSU\CSE373\Assignment\Assignment-2(BFS implementation)\Main.exe"
Enter the total number of vertices and edges: 7 8
Enter the vertex: 1 2 3 4 5 6 7
Enter edges: 1 2
Enter edges: 1 5
Enter edges: 1 3
Enter edges: 2 6
Enter edges: 2 4
Enter edges: 5 4
Enter edges: 3 4
Enter edges: 3 7

The result is :

Vertex: 1 2 3 4 5 6 7
Color:  B B B B B B B
d:      0 1 1 2 1 2 2
Prev:   N 1 1 2 1 2 3
Process returned 0 (0x0)   execution time : 47.758 s
Press any key to continue.
```

The time complexity of BFS:

In the case of an adjacency list representation, where V and E are the numbers of vertices and edges, respectively, the time complexity of the Breadth-First Search (BFS) is O($V + E$). The reason for this complexity is that each edge is examined once (requiring O(V) time), and each vertex is processed once (requiring $O(V)$ time). On the other hand, considering an adjacency matrix representation, BFS has an intime complexity of (V^2) since each vertex must have its adjacent vertices checked, which requires scanning a $V \times V$ matrix. As a result, BFS performs better overall when it has an adjacency list, particularly in sparse graphs where the number of edges is much smaller than the square of the number of vertices.

Benefits of the BFS algorithm:

- When determining the shortest path in an unweighted graph, BFS performs excellently. To ensure the shortest route is identified, it explores every node at the current depth level before going on to nodes at the next level.

- BFS works exceptionally well for unweighted graphs, where the shortest path is only the set of edges with the lowest possible connecting nodes because it investigates every node level by level.

Limitations of the BFS algorithm:

- Because BFS does not consider edge weights, graphs with weighted edges are inappropriate. If path costs differ, it may produce inaccurate results since it treats all edges equally.

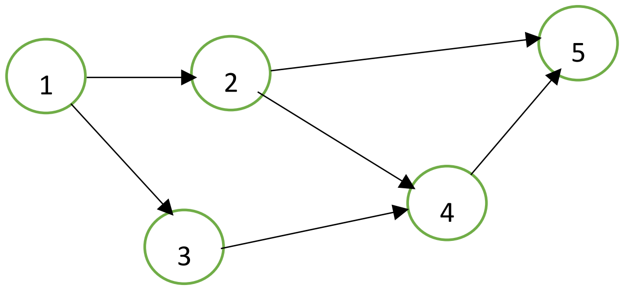
- BFS may become slow in deep graphs, which have many levels since it investigates every node one level at a time. If the graph has a deep structure, BFS may take a while to locate the target node.

Topological Sort Algorithm

What is Topological Sort:

In the Directed Acyclic Graph, Topological sort is a way of the linear ordering of vertices v1, v2, …. vN in such a way that for every directed edge x → y, x will come before y in the ordering.

Example: The topological sort for the below graph is 1, 2, 4, 3, 5



How can we find a topological sort:

Topological Sort Algorithm:

- ✓ Perform a DFS traversal from every vertex in the graph, **not** clearing markings in between traversals.
- ✓ Record DFS postorder along the way.
- ✓ Topological ordering is the reverse of the postorder.

Advantages of the Topological Sort:

- ❖ Efficient ordering.
- ❖ Easy implementation using adjacency lists.
- ❖ Wide range of applications.
- ❖ Low time and space complexity.
- ❖ Scalability for large-scale data processing.

Disadvantages of Topological Sort:

- ❖ Only applicable to directed acyclic graphs (DAGs), not suitable for cyclic graphs.
- ❖ May not be unique, multiple valid topological orderings can exist. ❖ Inefficient for large graphs with many nodes and edges.

The time complexity of the topological sort:

The topological sort has an O(V+E) time complexity, where V is the number of graph vertices and E is the number of graph edges.

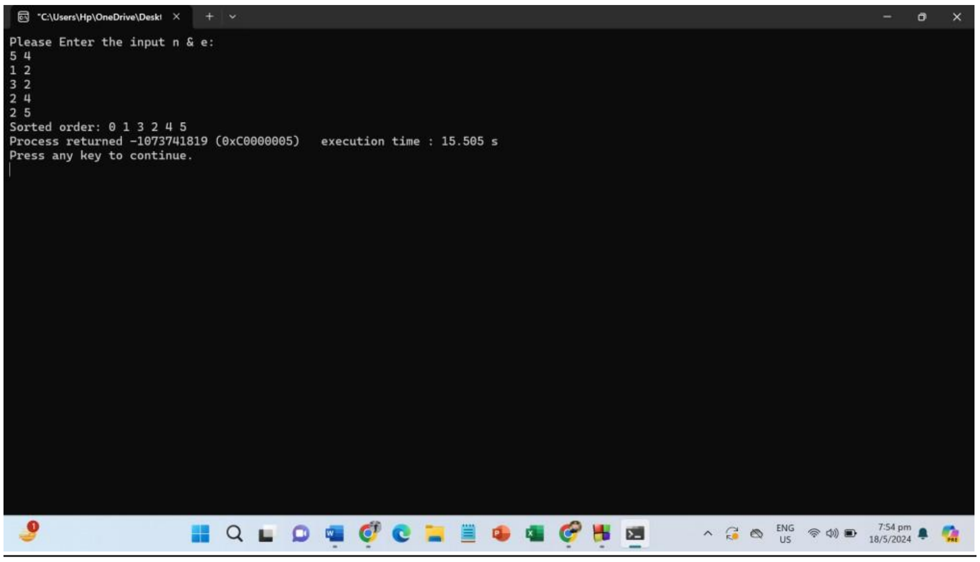
Applications of Topological Sort-

- Scheduling jobs from the given dependencies among jobs
- Instruction Scheduling
- Determining the order of compilation tasks to perform in makefiles
- Data Serialization
- **Deadlock Detection**
- Course Scheduling
- **Event Management**

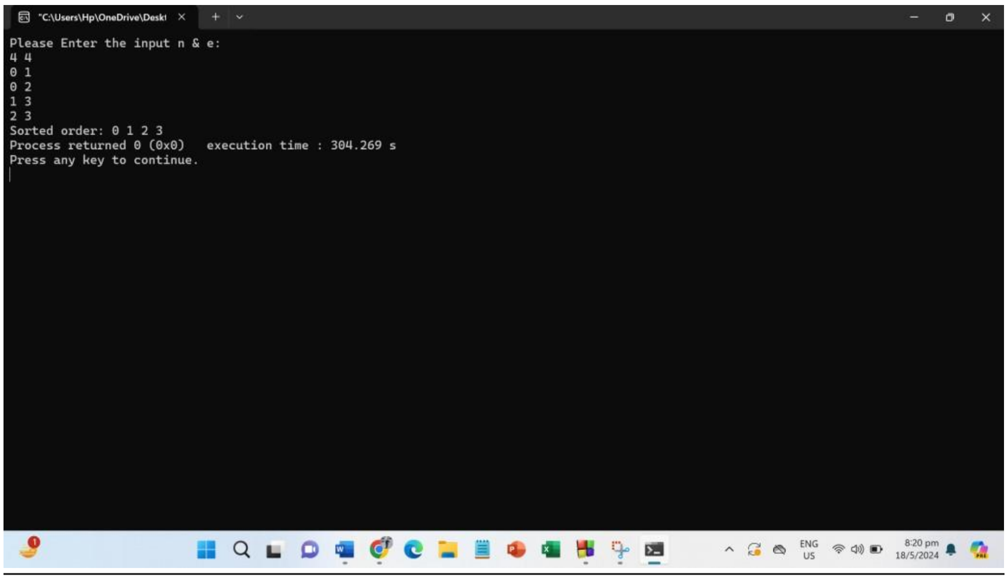
Code Implementation of Topological Sort :

```
#include <bits/stdc++.h>
using namespace std;

int main(){ int
n,e;
int count=0;

cout <<"Please Enter the input n & e: "<<endl; cin>>n>>e;

vector<vector<int>>adjacency(n);
vector<int> indegree(n,0);

for(int i=0;i<e;i++){
   int u,v;
cin>>u>>v;
   adjacency[u].push_back(v);
   indegree[v]++;
}

queue<int>pri_que; for(int
i=0;i<e;i++){
if(indegree[i]==0){
pri_que.push(i);
   }
}

cout<<"Sorted order: ";
while(!pri_que.empty()){
count++;     int
x=pri_que.front();
pri_que.pop();
cout<<x<<" ";     for(auto it
: adjacency[x]){
indegree[it]--;
if(indegree[it]==0)
            pri_que.push(it);
    }
}
}
```

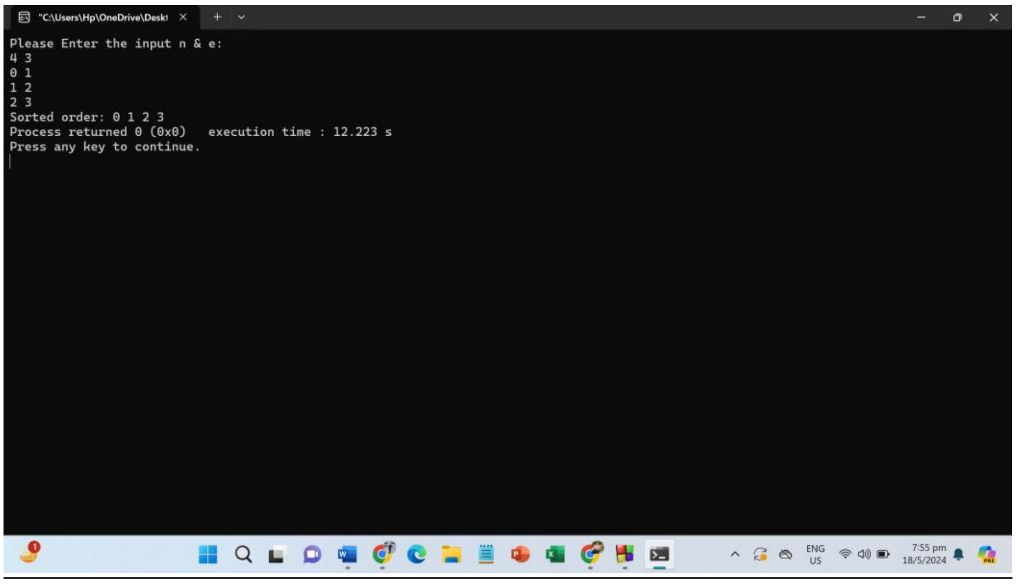
Input and Output:



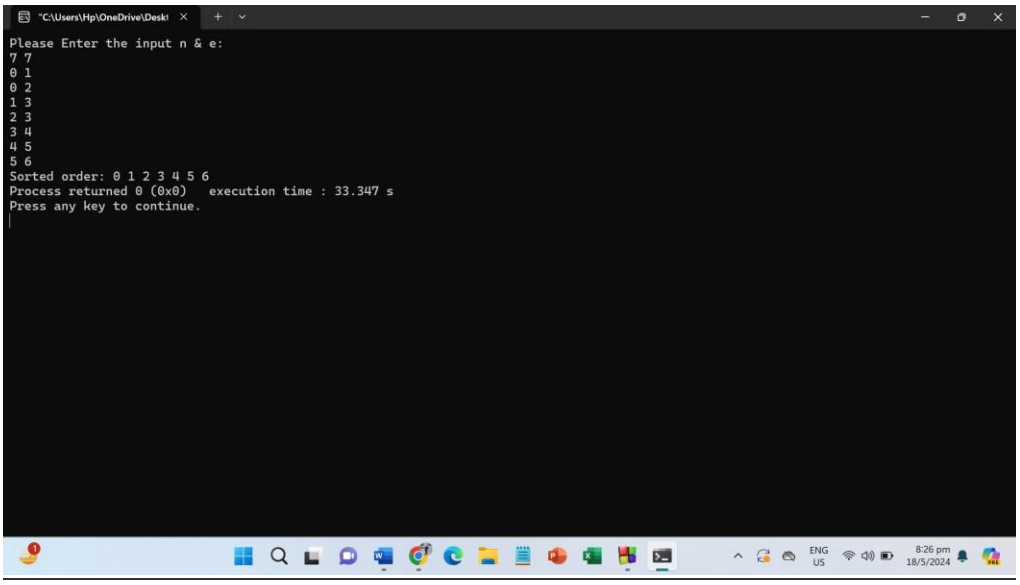
```
Please Enter the input n & e:
5 4
1 2
3 2
2 4
2 5
Sorted order: 0 1 3 2 4 5
Process returned -1073741819 (0xC0000005)   execution time : 15.505 s
Press any key to continue.
```



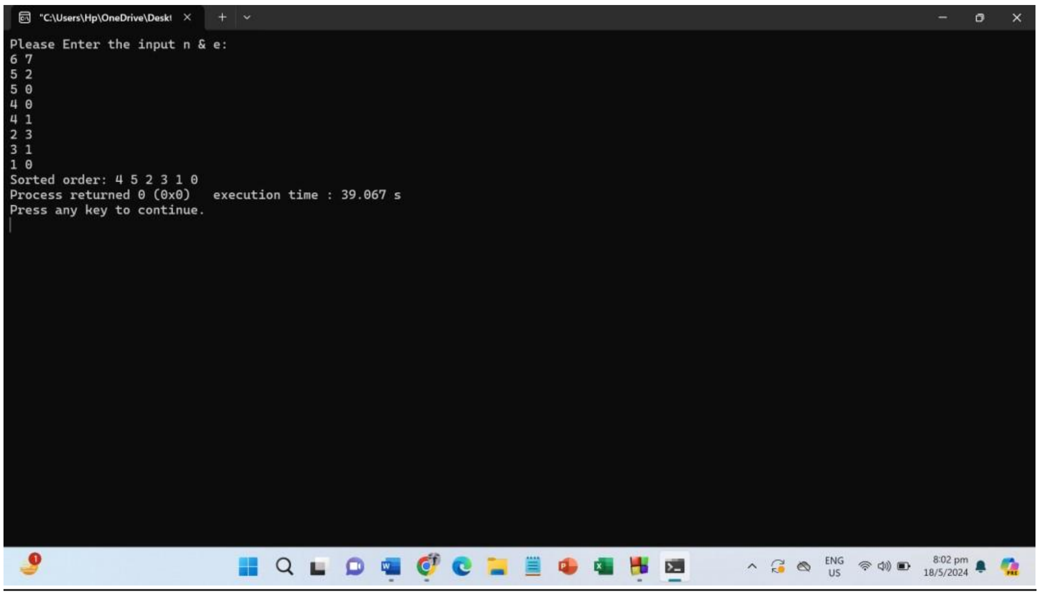
```
Please Enter the input n & e:
4 4
0 1
0 2
1 3
2 3
Sorted order: 0 1 2 3
Process returned 0 (0x0)   execution time : 304.269 s
Press any key to continue.
```



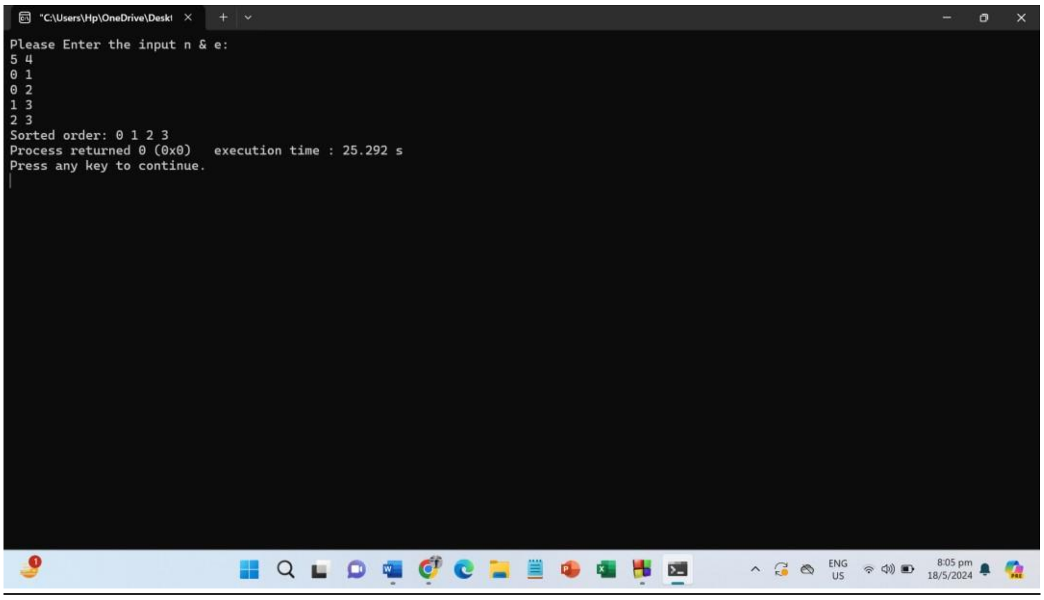
```
Please Enter the input n & e:
4 3
0 1
1 2
2 3
Sorted order: 0 1 2 3
Process returned 0 (0x0)   execution time : 12.223 s
Press any key to continue.
```



```
Please Enter the input n & e:
7 7
0 1
0 2
1 3
2 3
3 4
4 5
5 6
Sorted order: 0 1 2 3 4 5 6
Process returned 0 (0x0)   execution time : 33.347 s
Press any key to continue.
```



```
Please Enter the input n & e:
6 7
5 2
5 0
4 0
4 1
2 3
3 1
1 0
Sorted order: 4 5 2 3 1 0
Process returned 0 (0x0)   execution time : 39.067 s
Press any key to continue.
```



```
Please Enter the input n & e:
5 4
0 1
0 2
1 3
2 3
Sorted order: 0 1 2 3
Process returned 0 (0x0)   execution time : 25.292 s
Press any key to continue.
```

DFS

DFS is a graph data structure’s algorithm technique which is used for traversing or searching in graph data structure .So basically DFS algorithm goes to deep in a path in graph then come back and does the same for all other paths until every path Is finished searching .our slide’s pseudo code explains how we implemented the DFS algorithm. It is an algorithm mainly to explore the tree structure and see the connected paths.it is not efficient to find the shortest path, but its time complexity is O (V+E), where V is the number of vertices and E is the number of edges.

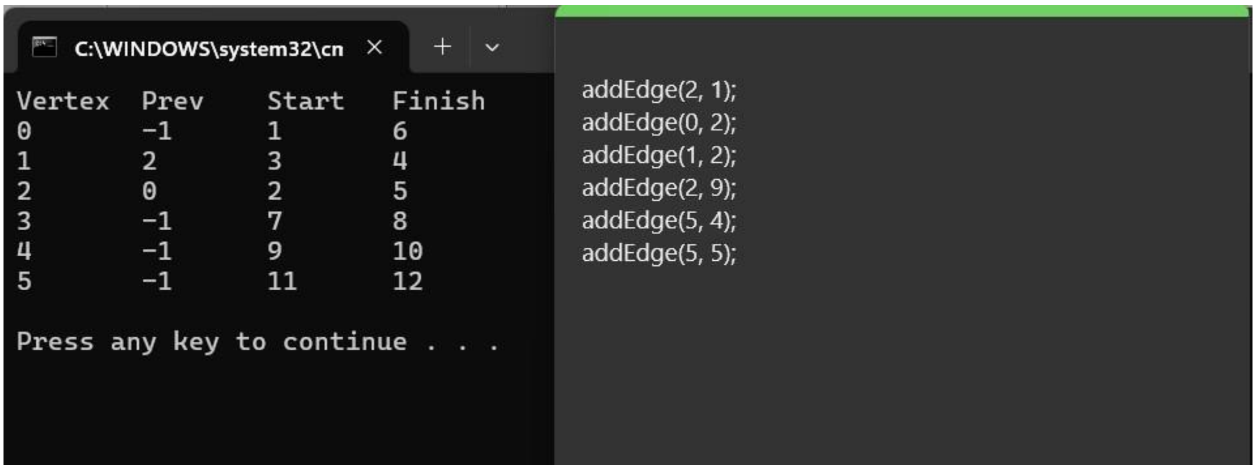
CODE

```
#include <bits/stdc++.h> const int MAX = 1000; //
Maximum number of vertices int V; // Number of
vertices int Adjacency_Matrix[MAX][MAX]; //
Adjacency matrix std::string color[MAX]; int
previous[MAX]; int d[MAX]; int f[MAX]; int
time_equal; using namespace std;



void addEdge(int u, int v);
void DFS_Visit(int u);
void DFS(int u); void
printResults(); int
main()
{
 ios_base::sync_with_stdio(false);
cin.tie(0);
 // cout.tie(0);
```

```
   V = 6; // Number of vertices
for (int i = 0; i < V; ++i) {
for (int j = 0; j < V; ++j) {
            Adjacency_Matrix[i][j] = 0; // Initialize adjacency matrix with 0
       }
   }

   addEdge(0, 1);     addEdge(1, 2);
addEdge(2, 3);     addEdge(3, 4);
addEdge(5, 6);     addEdge(7, 8);     int
startNode = 0; // Starting node for DFS
DFS(startNode);     printResults();

   return 0;
}



void addEdge(int u, int v) {
   Adjacency_Matrix[u][v] = 1; // Add an edge from u to v
}

void DFS_Visit(int u) {
color[u] = "GREY";
time_equal++;     d[u]
= time_equal;     //for
each v ∈ Adj[u]
   for (int v = 0; v < V; ++v) {
       if (Adjacency_Matrix[u][v] == 1 && color[v] == "WHITE") {
previous[v] = u;
            DFS_Visit(v);
       }
   }

   color[u] = "BLACK";
time_equal++;     f[u]
= time_equal;
}

void DFS(int u) {     //for
each vertex u ∈ V     for (int
u = 0; u < V; ++u) {
color[u] = "WHITE";
previous[u] = -1;        d[u] =
INT_MAX;        f[u] =
INT_MAX;
   }

   time_equal = 0;    //for
each vertex u ∈ V     if
(color[u] == "WHITE") {
DFS_Visit(u);
   }

   for (int u = 0; u < V; ++u) {
if (color[u] == "WHITE") {
         DFS_Visit(u);
      }
   }
}

void printResults() {     cout <<
"Vertex\tPrev\tStart\tFinish\n";
```

```
    for (int i = 0; i < V; ++i) {        cout << i << "\t" << previous[i] << "\t" <<
d[i] << "\t" << f[i] << "\n";
    }
}
```

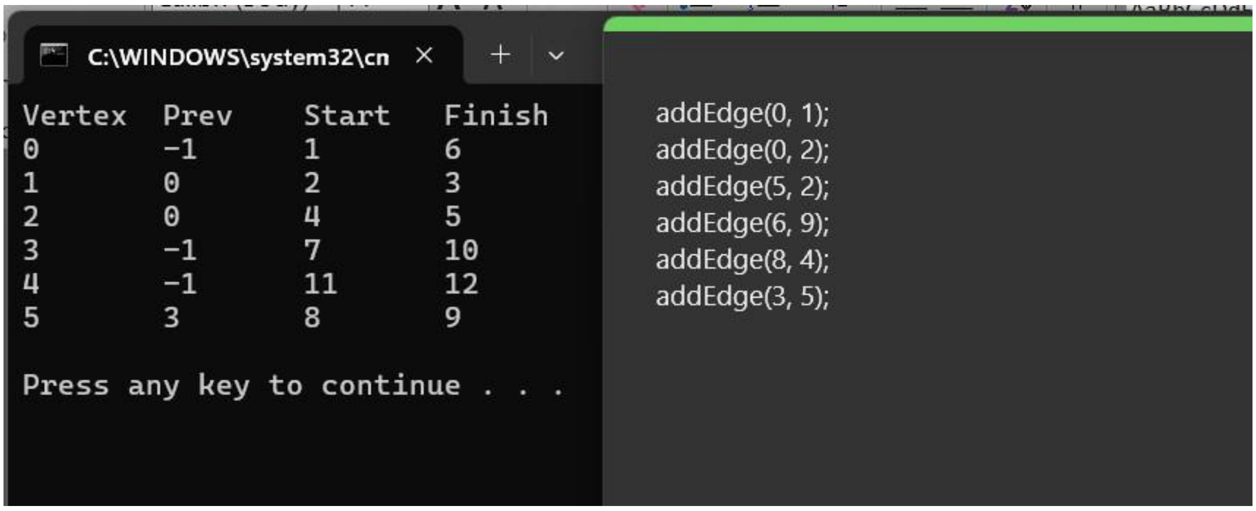
SAMPLE OUTPUT 1



```
Vertex  Prev    Start   Finish
0       -1      1       6
1       2       3       4
2       0       2       5
3       -1      7       8
4       -1      9       10
5       -1      11      12

Press any key to continue . . .
```

```
addEdge(2, 1);
addEdge(0, 2);
addEdge(1, 2);
addEdge(2, 9);
addEdge(5, 4);
addEdge(5, 5);
```

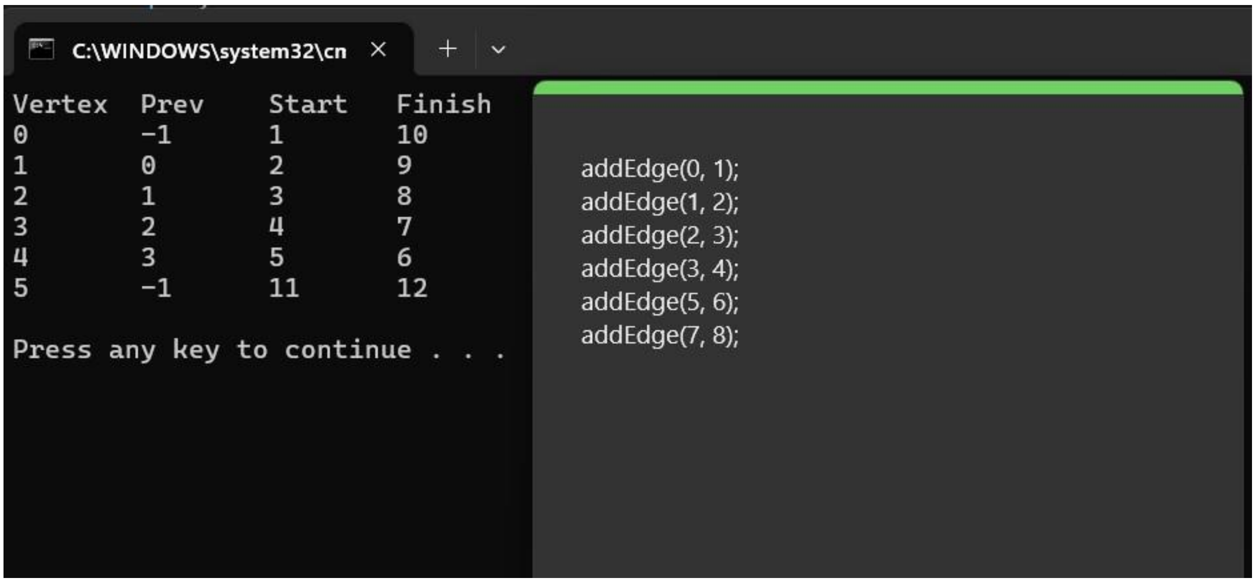
SAMPLE OUTPUT 2



```
Vertex  Prev    Start   Finish
0       -1      1       6
1       0       2       3
2       0       4       5
3       -1      7       10
4       -1      11      12
5       3       8       9

Press any key to continue . . .
```

```
addEdge(0, 1);
addEdge(0, 2);
addEdge(5, 2);
addEdge(6, 9);
addEdge(8, 4);
addEdge(3, 5);
```

SAMPLE OUTPUT 3



C:\WINDOWS\system32\cn

```
Vertex  Prev    Start   Finish
0       -1      1       10
1       0       2       9
2       1       3       8
3       2       4       7
4       3       5       6
5       -1      11      12

Press any key to continue . . .
```

```
addEdge(0, 1);
addEdge(1, 2);
addEdge(2, 3);
addEdge(3, 4);
addEdge(5, 6);
addEdge(7, 8);
```