Answer to the question no-01

Advantage of BFS:

1. Shortest path: when we have to determine shortest route from source to destination in an unweighted graph, it is very useful.
2. Level traversal: when we have to determine a node's level from source node, it works perfectly.
3. Optimization: when there are many solutions for a problem , BFS helps to find the optimal solution.

Advantage of DFS:

1. Topological sorting: when we have to do topological sorting in a directed acyclic graph(DAG), it is more helpful.
2. Single solution: if there is only one solution for a problem DFS performs properly.
3. Memory : DFS runs on path, there is no need to store the nodes. So, it takes less Memory.

Disadvantage of BFS:

1. It stores nodes. For that it needs much more memory than DFS.
2. We can not find the longest path from source node to destination.

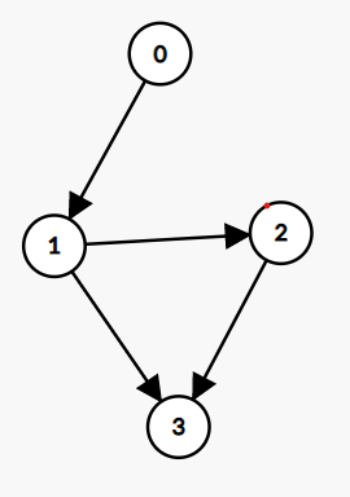
Disadvantage of DFS:

1. In an unweighted cyclic graph, it can not perform the shortest path from source to destination.
2. It can not determine the level properly.

Answer to the question no-02

There are 3 ways to represent the graph to a computer.

1.Adjacency Matrix:
Expression of a graph by 2D Matrix.
If the adjacency matrix is M[node][node], then M[i][j]==1 if there is direct connection between i to j using only 1 edge.

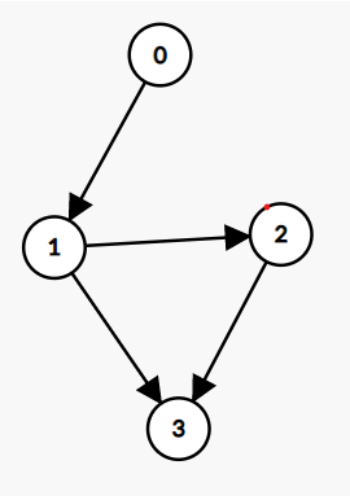


This is a directed graph.
If we want to express it using adjacency matrix then it will look like:

0	1	0	0
0	0	1	1
0	0	0	1
0	0	0	0

If it were a weighted graph then we can write the weight instead of writing 1.

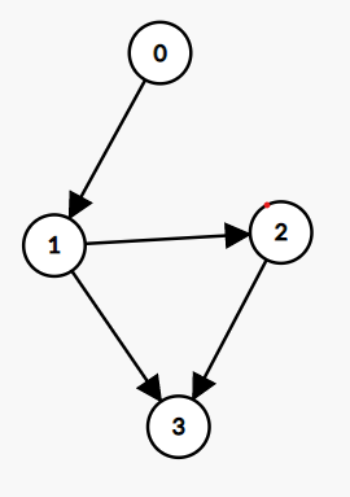
2.Adjacency List:
Expression of a graph by linked list or vector. It is a separate list of every node.



For this unweighted directed graph, we can write the list and it will look like:

0->1
1->2,3
2->3
3->
If it were weighted, we can pair the node and value together.

3.Edge List:
Expression of a graph by set.



For this unweighted directed graph, we can write the set and it will look like:
[
  [0,1]
  [1,2]
  [2,3]
  [1,3]
]
If it were a weighted graph then we can add the weight after the 2 nodes in the cell.
So, in this way we can represent all types of graphs.

Answer to the question no-03

```
#include<bits/stdc++.h>
using namespace std;
const int N=1e5+10;
int visited[N];
vector<int>adj_list[N];
int n;
bool DFS(int node)
{
   visited[node]=1;
   for(int adj_node:adj_list[node])
   {
      if(visited[adj_node]==0)
      {
         bool run=DFS(adj_node);
         if(run)
         {

            return true;
         }
      }
      else if(visited[adj_node]==2)
      {
         return true;
      }
   }
   visited[node]=2;
   return false;
}
int main()
{
   int e;
   cin>>n>>e;
   for(int i=1; i<=e; i++)
   {
      int u,v;
      cin>>u>>v;
```

```
        adj_list[u].push_back(v);
        adj_list[v].push_back(u);
    }
    bool run;
    for(int i=1; i<=n; i++)
    {
        if(visited[i]==0)
        {
            run=DFS(i);
            if(run)
            {
                cout<<"Cycle Exist\n";
                break;
            }
        }
    }
    if(!run)
    {
        cout<<"No Cycle\n";
    }
}
```

Answer to the question no-04

```
#include<bits/stdc++.h>
using namespace std;
int array_sum(int b,int a,int A[])
{
    if(a==0)
    {
        return A[a];
    }
    b+=A[a]+array_sum(b,--a,A);
    return b;
}
int main()
{
    int n;
```

```
    cin>>n;
    int A[n];
    for(int i=0;i<n;i++)
    {
        cin>>A[i];
    }
    cout<<array_sum(0,n-1,A)<<endl;
}
```

Answer to the question no-05

```
#include<bits/stdc++.h>
using namespace std;
const int N=1e5 +5;
vector<int>adj_list[N];
int visited[N],level[N],parent[N];
void DFS(int node)
{
    visited[node]=1;
    for(auto adj_node:adj_list[node])
    {
        if(visited[adj_node]==0)
        {
            parent[adj_node]=node;
            level[adj_node]=level[node]+1;
            DFS(adj_node);
        }
    }
}
int main()
{
    int nodes,edges;
    cin>>nodes>>edges;
    for(int i=1;i<=edges;i++)
    {
        int u,v;
        cin>>u>>v;
        adj_list[u].push_back(v);
```

```
        adj_list[v].push_back(u);
    }
    int src=1;
    DFS(src);
    if(visited[nodes]==0)
    {
        cout<<"NO"<<endl;
    }
    else
    {
        cout<<"YES"<<endl;
    }

}
```

Answer to the question no-06

```
#include<bits/stdc++.h>
using namespace std;
const int N=1005;
int maze[N][N];
int level[N][N];
int dx[]= {0,0,-1,1},dy[]= {1,-1,0,0};
int visited[N][N];
int n,m,a,b;
int parent[N][N];
deque<char>r;
char c[]= {'R','L','U','D'};
bool path()
{
    for(int i=0; i<m; i++)
    {
        if(visited[0][i]==1)
        {
            a=0;
            b=i;
            return true;
        }
```

```
    }
    for(int i=0; i<n; i++)
    {
        if(visited[i][0]==1)
        {
            a=i;
            b=0;
            return true;
        }
    }
    for(int i=1; i<n; i++)
    {
        if(visited[i][m-1]==1)
        {
            a=i;
            b=m-1;
            return true;
        }
    }
    for(int i=1; i<m-1; i++)
    {
        if(visited[n-1][i]==1)
        {
            a=n-1;
            b=i;
            return true;
        }
    }
    return false;
}
bool is_inside(pair<int,int>coord)
{
    int x=coord.first;
    int y=coord.second;
    if(x>=0&&x<n && y>=0 && y<m)
    {
        return true;
```

```
    }
    return false;
}
bool is_safe(pair<int,int>coord)
{
    int x=coord.first;
    int y=coord.second;
    if(maze[x][y]==-1)
    {
        return false;
    }
    return true;
}
void BFS(pair<int,int>src)
{
    queue<pair<int,int>>q;
    visited[src.first][src.second]=1;
    level[src.first][src.second]=0;
    parent[src.first][src.second]=0;
    q.push(src);
    while(!q.empty())
    {
        pair<int,int>head=q.front();
        q.pop();
        int x=head.first,y=head.second;
        for(int i=0; i<4; i++)
        {
            int new_x=x+dx[i];
            int new_y=y+dy[i];
            pair<int,int>adj_node= {new_x,new_y};
            if(is_inside(adj_node) && is_safe(adj_node) && visited[new_x][new_y]==0)
            {
                visited[new_x][new_y]=1;
                level[new_x][new_y]=level[x][y]+1;
                parent[new_x][new_y]=i;
                q.push(adj_node);
            }
```

```
        }
    }
}
int main()
{
    cin>>n>>m;
    pair<int,int>src,dst;
    memset(level,-1,sizeof(level));
    memset(visited,0,sizeof(visited));
    for(int i=0; i<n; i++)
    {
        string input;
        cin>>input;
        for(int j=0; j<m; j++)
        {
            if(input[j]=='#'||input[j]=='M')
            {
                maze[i][j]=-1;
            }
            else if(input[j]=='A')
            {
                src= {i,j};
            }
        }
    }
    BFS(src);
    if(!path())
    {
        cout<<"NO\n";
        return 0;
    }
    cout<<"YES\n";
    dst.first=a;dst.second=b;
    cout<<level[a][b]<<endl;
    while(1)
    {
        int d=parent[dst.first][dst.second];
```

```
        if(src==dst)
        {
            break;
        }
        r.push_front(c[d]);
        dst.first-=dx[d];
        dst.second-=dy[d];
    }
    for(auto i:r)
    {
        cout<<i;
    }
}
```

Answer to the question no-07

The Dijkstra algorithm is used for determining the shortest path on a weighted graph. So, the steps are follow:

1. First of all we determine the distance from source to destination is infinity.

	A	B	C	D	E	F	G	H	I	J
	-	-	-	-	-	-	-	-	-	-

2. As our source node is E , so explore the node from E and write down the distance from it.

	A	B	C	D	E	F	G	H	I	J
	-	-	-	-	-	-	-	-	-	-
E	-	20	9	-	0	-	5	-	-	-

3. E is a done node. Now choose the smallest distance which is node G and explore it. Here exploring G we get node E but we cannot update the value as 0<5.

	A	B	C	D	E	F	G	H	I	J
	-	-	-	-	-	-	-	-	-	-
E	-	20	9	-	0	-	5	-	-	-
G	-	20	9	-	0	6	5	-	-	-

4. G is a done node. Now the smallest is node F. Exploring node F we get B,H. as 13+6<20 so we update the distance of B and write down the distance of H.

	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**
	-	-	-	-	-	-	-	-	-	-
E	-	20	9	-	0	-	5	-	-	-
G	-	20	9	-	0	6	5	-	-	-
F	-	19	9	-	0	6	5	27	-	-

5. F is done. Now the smallest is node C. when exploring C we get B . As 16<19 so, we update the distance and write down the distance of A.

	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**
	-	-	-	-	-	-	-	-	-	-

E	-	20	9	-	0	-	5	-	-	-
G	-	20	9	-	0	6	5	-	-	-
F	-	19	9	-	0	6	5	27	-	-
C	10	16	9	-	0	6	5	27	-	-

6. C is done. A is now the smallest. When exploring A we get B,I,J. Here 12<16 so we update the distance of B and write down the distance of I and J.

	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**
	-	-	-	-	-	-	-	-	-	-
E	-	20	9	-	0	-	5	-	-	-
G	-	20	9	-	0	6	5	-	-	-
F	-	19	9	-	0	6	5	27	-	-
C	10	16	9	-	0	6	5	27	-	-
A	10	12	9	-	0	6	5	27	28	15

7. A is done. Now the smallest is node B. From node B we can explore A,C,E,F,G,H. Distance of H is the same, so we do not need to update the value. So write down the values.

	A	B	C	D	E	F	G	H	I	J
	-	-	-	-	-	-	-	-	-	-
E	-	20	9	-	0	-	5	-	-	-
G	-	20	9	-	0	6	5	-	-	-
F	-	19	9	-	0	6	5	27	-	-
C	10	16	9	-	0	6	5	27	-	-
A	10	12	9	-	0	6	5	27	28	15
B	10	12	9	-	0	6	5	27	28	15

8. B is done. Now the smallest is J. we get A,B,I. Here 21<28 , so we update the distance of I and write down the others.

	A	B	C	D	E	F	G	H	I	J
	-	-	-	-	-	-	-	-	-	-
E	-	20	9	-	0	-	5	-	-	-
G	-	20	9	-	0	6	5	-	-	-
F	-	19	9	-	0	6	5	27	-	-
C	10	16	9	-	0	6	5	27	-	-
A	10	12	9	-	0	6	5	27	28	15

B	10	12	9	-	0	6	5	27	28	15
J	10	12	9	-	0	6	5	27	21	15

9. J is done. Now the smallest is I. we get A,H,J. As the value becomes large , we don't need to update it.

	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**
	-	-	-	-	-	-	-	-	-	-
E	-	20	9	-	0	-	5	-	-	-
G	-	20	9	-	0	6	5	-	-	-
F	-	19	9	-	0	6	5	27	-	-
C	10	16	9	-	0	6	5	27	-	-
A	10	12	9	-	0	6	5	27	28	15
B	10	12	9	-	0	6	5	27	28	15
J	10	12	9	-	0	6	5	27	21	15
I	10	12	9	-	0	6	5	27	21	15

10. ‘I’ is done. Last one is H. There is no scope to update any value. So, write down the previous row.

	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**
	-	-	-	-	-	-	-	-	-	-
E	-	20	9	-	0	-	5	-	-	-

G	-	20	9	-	0	6	5	-	-	-
F	-	19	9	-	0	6	5	27	-	-
C	10	16	9	-	0	6	5	27	-	-
A	10	12	9	-	0	6	5	27	28	15
B	10	12	9	-	0	6	5	27	28	15
J	10	12	9	-	0	6	5	27	21	15
I	10	12	9	-	0	6	5	27	21	15
H	10	12	9	-	0	6	5	27	21	15

11. After performing the dijkstra algorithm we get the distance from E to all nodes.

	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**
	-	-	-	-	-	-	-	-	-	-
E	-	20	9	-	0	-	5	-	-	-
G	-	20	9	-	0	6	5	-	-	-
F	-	19	9	-	0	6	5	27	-	-
C	10	16	9	-	0	6	5	27	-	-
A	10	12	9	-	0	6	5	27	28	15
B	10	12	9	-	0	6	5	27	28	15
J	10	12	9	-	0	6	5	27	21	15
I	10	12	9	-	0	6	5	27	21	15
H	10	12	9	-	0	6	5	27	21	15

From E node

A->10
B->12
C->9
E->0
F->6
G->5
H->27
I->21
J->15

Answer to the question no-08

Recursive case:
It is a part of a recursive function. When a function calls itself ,it is called a recursive case and the recursive case will run until the termination condition is met.

Base case:
It is a very important part of a recursive function. Because it is the termination condition of a recursive function. If we do not write a base case the program will fall in infinite position.

The relation between base case and recursive base is very important. Because base case can not terminate the function itself, when a recursive case calls the base case then it terminates the recursion.

If we see an example:

```
int Pow(int n,int m)
{
if(m==1)
{
return n;
}
return n*Pow(n,m-1);
}
```

Here the recursive case is Pow(n,m-1) and the base case is m==1.
The function Pow is calling itself by lessening the number by 1.
As a result when m==1 then it terminates the function , does the calculation and returns the value to the main function.
This is how a recursive function works.