ANSWER NO - 01

A .

a) Advantages of BFS:

   i) Guarantees the shortest path between the starting node and any other reachable node in an unweighted graph.

   ii) Can be used to detect if a graph is bipartite

b) Advantages of DFS:

   i) DFS is more suitable for searching deeper into a graph or tree and can be faster in certain cases.

   ii) DFS can be used to solve problems related to topological sorting

B.

a) Disadvantage of BFS :

   i) BFS requires more memory compared to DFS, as it needs to store all the visited nodes in a queue.

b) Disadvantage of DFS :

   i) DFS may not find the shortest path between two nodes and may get stuck in an infinite loop if the graph has cycles

ANSWER NO - 02

Here are some of the most common ways to represent a graph :

1. **Adjacency Matrix:**
   An adjacency matrix is a 2D array that represents the edges between nodes. The element in the i-th row and j-th column of the matrix is 1 if there is an edge between nodes i and j, and 0 otherwise.
   For an undirected graph, the matrix is symmetric. For example, the following is the adjacency matrix for a simple undirected graph with 4 nodes:

   Example :
   0 1 0 0
   0 1 0 0
   1 0 1 1
   0 1 1 1

2. **Adjacency List:**
   An adjacency list is a collection of linked lists where each list represents the set of nodes adjacent to a particular node. For example, the following is the adjacency list for the same graph as above:

   Example :
   0: 1, 2
   1: 0, 2, 3
   2: 0, 1, 3
   3: 1, 2

3. **Edge List:**
   An edge list is a list of all the edges in a graph, where each edge is represented as a pair of nodes. For example, the edge list for the same graph as above is:

   Example :

   [(0, 1), (0, 2), (1, 2), (1, 3), (2, 3)]

ANSWER NO - 03

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

const int N = 1e5;
vector<int>adj_list[N];
int visited[N];

bool dfs(int src, int parent)
{
   visited[src] = 1;

   for(auto nxt : adj_list[src]){
      if(visited[nxt] == 0){
         if(dfs(nxt, src)){
            return true;
         }
      }
      else if(nxt != parent){
         return true;
      }
   }

   return false;
}

int main()
{
   int v, e;cin>>v>>e;

   for(int i=0; i<e; i++){
      int f, t;cin>>f>>t;
```

```
        adj_list[f].push_back(t);
        adj_list[t].push_back(f);
    }

    bool is_cyclic = false;

    for(int i=0; i<v; i++){
        if(visited[i] == 0){
            if(dfs(i, -1)){
                is_cyclic = true;
                break;
            }
        }
    }

    if(is_cyclic){
        cout<<"Cycle Exist"<<endl;
    }
    else
        cout<<"No Cycle"<<endl;

    return 0;
}
```

ANSWER NO - 04

```
#include<bits/stdc++.h>
using namespace std;
int sum(vector<int>num, int n)
{
    if(n == 0){
        return num[0];
    }
    return num[n] + sum(num, n-1);
}
int main()
{
    int n;cin>>n;
    vector<int>num(n);
    for(int i=0; i<n; i++){
        cin>>num[i];
    }
    cout<<sum(num, n-1)<<endl;
    return 0;
}
```

ANSWER NO - 05

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

const int N = 1e5;
int visited[N];
vector<int>adj_list[N];

void BFS(int src)
{
    queue<int>q;
    visited[src] = 1;
    q.push(src);

    while(!q.empty())
    {
        int head = q.front();
        q.pop();
        for(int adj_node : adj_list[head])
        {
            if(visited[adj_node] == 0){
                visited[adj_node] = 1;
                q.push(adj_node);
            }
        }
    }
}
int main()
{
    int v, e;cin>>v>>e;
    for(int i=0; i<e; i++)
    {
        int a, b;cin>>a>>b;
        adj_list[a].push_back(b);
        adj_list[b].push_back(a);
    }

    int src = 1;
    BFS(src);

    int dst = v;

    if(visited[dst] == 0){
        cout<<"NO"<<endl;
    }
    else{
        cout<<"YES"<<endl;
    }

    return 0;
}
```

ANSWER NO - 06

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

const int N = 2002;
char maze[N][N], visited[N][N];

pair<int, int>src, dst, parent[N][N];
int n, m;
string dir = "";

int dx[] = {1, 0, -1, 0};
int dy[] = {0, 1, 0, -1};
char dp[] = {'D', 'R', 'U', 'L'};

void bfs(pair<int, int> src)
{
   queue<pair<int, int>>q;
   visited[src.first][src.second] = true;
   parent[src.first][src.second] = {-1, -1};
   q.push(src);

   while(!q.empty())
   {
      pair<int, int>head = q.front();
      q.pop();

      int x = head.first;
      int y = head.second;


      for(int i=0; i<4; i++)
      {

         int new_x = x+ dx[i];
         int new_y = y+ dy[i];
         if(new_x>=1 && new_x <=n && new_y >=1 && new_y <=m &&
!visited[new_x][new_y] && maze[new_x][new_y] != '#' &&
maze[new_x][new_y] != 'M'){

            visited[new_x][new_y] = true;
            parent[new_x][new_y] = head;
            q.push({new_x, new_y});
         }
      }
   }
}
void path(pair<int, int> start_, pair<int, int> end_)
{
   string path_directions = "";
   pair<int, int> current = end_;

   while(current != start_)
   {
      pair<int, int> prev = parent[current.first][current.second];
      int diff_x = current.first - prev.first;
      int diff_y = current.second - prev.second;

      for(int i=0; i<4; i++)
      {
         if(dx[i] == diff_x && dy[i] == diff_y)
         {
            path_directions += dp[i];
            break;
         }
      }

      current = prev;
   }
```

```
    reverse(path_directions.begin(), path_directions.end());
    cout << path_directions.size() << endl;
    cout << path_directions << endl;
}

int main()
{
    cin>>n>>m;

    pair<int, int>start_, end_;

    for(int i=1; i<=n; i++)
    {
        for(int j=1; j<=m; j++){
            cin>>maze[i][j];
            if (maze[i][j] == 'A') {
                start_.first = i;
                start_.second = j;
            }
            if (maze[i][j] == '.' && !visited[i][j] && (i == 1 || j == 1 || i == n || j == m))
{
                bool is_adjacent_to_monster = false;
                for(int k=0; k<4; k++) {
                    int new_x = i + dx[k];
                    int new_y = j + dy[k];
                    if(new_x>=1 && new_x <=n && new_y >=1 && new_y <=m &&
maze[new_x][new_y] == 'M') {
                        is_adjacent_to_monster = true;
                        break;
                    }
                }
                if(!is_adjacent_to_monster) {
                    end_.first = i;
                    end_.second = j;
                }
            }

        }
    }

    bfs(start_);

    if (visited[end_.first][end_.second]){
        cout<<"YES"<<endl;
        path(start_, end_);
    }
    else{
        cout<<"NO"<<endl;
    }

    return 0;
}
```

ANSWER NO - 07

selected	A	B	C	E	F	G	H	I	J
—	∞	∞	∞	0	∞	∞	∞	∞	∞
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

Lastly, The answer will be :

NODE	A	B	C	E	F	G	H	I	J
Distance from Source	10	12	9	0	6	5	27	21	15

ANSWER NO - 08

In a recursive function, the base case is a condition that terminates the recursion, and the recursive case is a condition that causes the function to call itself again. The recursive case is dependent on the base case, as without a base case, the recursive function would keep calling itself indefinitely, leading to an infinite loop.

The recursive case is used to break down a problem into smaller subproblems, each of which can be solved using the same algorithm. This is done by passing a smaller version of the original problem to the same function. The function will keep calling itself with smaller and smaller versions of the problem until the base case is reached, at which point the recursion stops and the function starts returning the values calculated in each recursive call.

For example, consider the following recursive function for calculating the factorial of a number:

```
def factorial(n):
   if n == 0:
      return 1
   else:
      return n * factorial(n-1)
```

In this function, the base case is when n is equal to 0, and the function returns 1. The recursive case is when n is greater than 0, and the function calls itself with n-1 as the argument. This continues until n is equal to 0, at which point the base case is reached, and the function stops calling itself and starts returning the values calculated in each recursive call.

Overall, the recursive case and the base case work together to create a recursive function that can solve complex problems by breaking them down into smaller subproblems that can be solved using the same algorithm.