# Answer Script

| Question No. 01 | |
|---|---|
| a) Write down the advantages of bfs and dfs (At least 2)?<br>b) Write down the disadvantages of bfs and dfs (At least 1)? | **10** |

**Answer No. 01**

**a) The advantages of bfs and dfs:**

Advantages of Breadth First Search:

1) It guarantees to find the shortest path from the starting point to the goal.
2) It will always find a solution because it searches the entire graph level by level until it finds the goal node.

Advantages of Depth First Search:

1) It consumes less memory.
2) It finds the larger distant element(from source vertex) in less time.

**b) The disadvantages of bfs and dfs:**

Disadvantages of Breadth First Search:

1) All of the connected vertices must be stored in memory. So it consumes more memory.

Disadvantages of Depth First Search:

1) It may not find optimal solution to the problem.

| Question No. 02 |
|---|
| What are the different ways to represent a graph? Describe With Example. **10** |

**Answer No. 02**

There are several ways to represent a graph. Some of them are:

1) Adjacency Matrix
2) Adjacency List
3) Edge List

1) **Adjacency Matrix:** An adjacency matrix is a 2D array where the rows and columns represent the vertices of the graph, and the cells indicate whether there is an edge between the vertices. If there is an edge between vertex i and vertex j, then the cell (i, j) is marked with a 1, otherwise it is marked with a 0.

   **Example:** Considering the following undirected graph:

```
1 — 2
|    |
|    |
3 — 4 — 5
     |
     |
     6
```

The adjacency matrix for this graph would be:

$$\begin{array}{c|ccccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
2 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
3 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
4 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\
5 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
6 & 0 & 0 & 0 & 0 & 1 & 0 & 0
\end{array}$$

2) **Adjacency List:** An adjacency list is a list of lists where each vertex has a list of its adjacent vertices. For example, the adjacency list for the same graph as above would be:

**Example:** Considering the following undirected graph:

```
1 — 2
|     |
|     |
3 — 4 — 5
      |
      |
      6
```

The adjacency list for this graph would be:

```
0: [ ]
1: [2, 3]
2: [1, 4]
3: [1, 4]
4: [2, 3, 5, 6]
5: [4]
6: [4]
```

3) **Edge List:** An edge list is a list of edges, where each edge is represented as a pair of vertices.

**Example:** Considering the following undirected graph:

```
1 — 2
|     |
|     |
3 — 4 — 5
      |
      |
      6
```

The edge list for this graph would be:

(1, 2), (1, 3), (2, 4), (3, 4), (4, 5), (4, 6)

**Question No. 03**

Write down a c++ program to detect cycle in an **undirected** graph. **15**

| **Sample Input-** | **Sample Output-** |
|---|---|
| 4 3<br>1 2<br>2 3<br>1 3<br><br>Explanation:<br>Here,Number of node is 4<br>Number of edge is 3 | Cycle Exist |
| 3 2<br>1 2<br>2 3<br><br>Explanation:<br>Here,Number of node is 3<br>Number of edge is 2 | No Cycle |

**Answer No. 03**

```
#include<bits/stdc++.h>
using namespace std;
const int N = 2e5;

vector<int> adj_list[N];
int visited[N];

bool detect_cycle(int node, int parent)
{
        visited[node] = true;
        for(int adj_node: adj_list[node]) {
        if(!visited[adj_node]) {
        if(detect_cycle(adj_node, node)) {
                return true;
        }
        }
        else if(adj_node != parent) {
```

```
        return true;
        }
        }
        return false;
}

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

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

        bool cycle_exists = false;

        for(int i = 1 ; i <= n ; i++) {
        if(!visited[i]) {
        if(detect_cycle(i, -1)) {
                cycle_exists = true;
                break;
        }
        }
        }

        if(cycle_exists) {
        cout<<"Cycle Exist\n";
        }
        else {
        cout<<"No Cycle\n";
        }
        return 0;
}
```

**Question No. 04**

You are given a positive integer n. The next line will contain n positive integers .Now calculate the total sum of the array.
Implement it using recursion.

**Constraints-**

1<=n<=100 , 1<=A[i]<=1000

**Write a C++ program for this problem**

**10**

| **Sample Input -** | **Sample Output -** |
|---|---|
| 4<br>26 3 17 5 | 51 |

**Answer No. 04**

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

int sum(int arr[], int n)
{
        if(n == 0) {
        return 0;
        }
        return arr[n-1] + sum(arr, n-1);
}

int main()
{
        int n;
        cin >> n;
        int arr[n];
        for(int i=0; i<n; i++) {
        cin >> arr[i];
        }
        int total_sum = sum(arr, n);
        cout << total_sum << endl;
        return 0;
}
```

## Question No. 05

Bangladesh has n cities, and m roads between them.You can go from one city to another if there exists a path between those two cities.
The goal is to  reach from city 1 to n.

**Input -**
The first input line has two integers n and m the number of cities and roads. The cities are numbered 1,2,…,n .After that, there are m lines describing the roads. Each line has two integers a and b. There is a road between those cities.A road always connects two different cities, and there is at most one road between any two cities.

**Output -** Print "YES" if your goal is possible, and "NO" otherwise.

**Constraints-**
2<=n<=10^5 , 1<=m<=2*10^5 , 1<=a,b<=n
**Write a C++ program for this problem**

**15**

| **Sample Input-** | **Sample Output-** |
|---|---|
| 10 8<br>1 3<br>3 4<br>3 6<br>4 6<br>2 5<br>1 7<br>3 10<br>9 8 | YES |
| 8 6<br>7 4<br>7 6<br>4 6 | NO |

|  |  |
|---|---|
| 2 5<br>1 3<br>7 8 |  |

**Answer No. 05**

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

const int N = 1e5 + 5;

vector<int>adj_list[N];
int visited[N];

void bfs(int src)
{
        visited[src] = 1;

        queue<int>q;
        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 n, m;
        cin >> n >> m;
        for(int i = 0 ; i < m ; i++) {
        int u, v;
        cin >> u >> v;
        adj_list[u].push_back(v);
        adj_list[v].push_back(u);
        }

        int src_node = 1, dest_node = n;

        bfs(src_node);

        if(visited[dest_node] == 1) {
        cout<<"YES\n";
        }
        else {
        cout<<"NO\n";
        }

        return 0;

}
```

## Question No. 06

You and some monsters are in a labyrinth.Your goal is to reach from cell A to one of the safe boundary cell.You can walk left, right, up and down.But you can't go to those cells , if there is a monster in that cell or the cell contains a wall.You can only go to the safe(.) cell.

**Input -**

The first input line has two integers n and m the height and width of the map. After this there are n lines of m characters describing the map. Each character is .(safe cell), # (wall), A (start), or M (monster). There is exactly one A in the input.

**Output -**

First print "YES" if your goal is possible, and "NO" otherwise.
If your goal is possible, also print any valid path(the length of the path and its description using characters `D`, `U`, `L`, and `R`).

**Constraints-**

$1 \le n,m \le 1000$

**Write a C++ program for this problem** **15**

| **Sample Input -** | **Sample Output -** |
|---|---|
| 5 8<br>..######<br>#M..A..#<br>#.#.M#.#<br>#M#..#..<br>..#.#### | YES<br>5<br>RRDDR |

**Explanation -**
For the above map , the following are the boundary cells
(1,1),(1,2),(1,3),(1,4)(1,5),(1,6), (1,7), (1,8)
(1,1),(2,1),(3,1),(4,1),(5,1)
(1,8),(2,8),(3,8),(4,8),(5,8)
(5,1),(5,2),(5,3),(5,4),(5,5),(5,6),(5,7),(5,8)
and,
safe boundary cells are (1,1) ,(1,2),(5,1),(5,2),(5,4),(4,8)

**Answer No. 06**

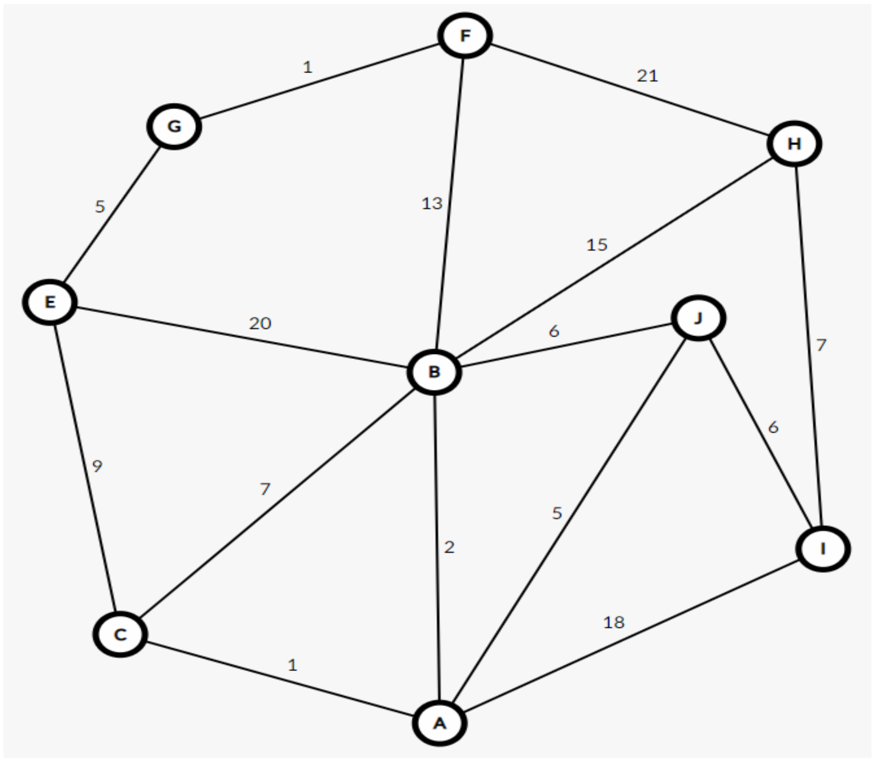
```
#include <bits/stdc++.h>
using namespace std;

const int maxN = 1010;
char grid[maxN][maxN];
bool visited[maxN][maxN];
pair<int, int> parent[maxN][maxN];

int dx[4] = {1, 0, -1, 0};
int dy[4] = {0, 1, 0, -1};
char dir[4] = {'U', 'L', 'D', 'R'};

void bfs(int x, int y, int n, int m) {
    queue<pair<int, int>> q;

    q.push({x, y});
    visited[x][y] = true;
    parent[x][y] = {-1, -1};
    while (!q.empty()) {
            pair<int, int> u = q.front();
            q.pop();
            for (int i = 0; i < 4; i++) {
                        int dx_, dy_;
                        dx_ = u.first + dx[i];
                        dy_ = u.second + dy[i];
                        if (dx_ >= 0 && dx_ < n && dy_ >= 0 && dy_ < m &&
!visited[dx_][dy_] && grid[dx_][dy_] != '#' && grid[dx_][dy_] != 'M') {
                        visited[dx_][dy_] = true;
                        q.push({dx_, dy_});
                        parent[dx_][dy_] = u;
                        }
            }
    }
}

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

    int start_x, start_y;

    for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                        cin >> grid[i][j];
                        if (grid[i][j] == 'A') {
                        start_x = i;
                        start_y = j;
                        }
                        if(grid[i][j] == 'A' && n == 1 && m == 1) {
                       cout << "YES\n";
                       cout << "0\n";
                       return 0;
                        }
            }
    }
    bfs(start_x, start_y, n, m);
```

```
bool found = false;
pair<int, int> end_point = {-1, -1};

for (int i = 0; i < n; i++) {
    if (grid[i][0] == '.' && visited[i][0]) {
        end_point = {i, 0};
        found = true;
        break;
    } else if (grid[i][m-1] == '.' && visited[i][m-1]) {
        end_point = {i, m-1};
        found = true;
        break;
    }
}

for (int j = 0; j < m; j++) {
    if (grid[0][j] == '.' && visited[0][j]) {
        end_point = {0, j};
        found = true;
        break;
    } else if (grid[n-1][j] == '.' && visited[n-1][j]) {
        end_point = {n-1, j};
        found = true;
        break;
    }
}

if (found) {
    cout << "YES\n";
    vector<pair<int, int>> path;
    path.push_back(end_point);
    int x = end_point.first;
    int y = end_point.second;
    while (parent[x][y] != make_pair(-1, -1)) {
        pair<int, int> p = parent[x][y];
        x = p.first, y = p.second;
        path.push_back({x, y});
    }
    reverse(path.begin(), path.end());
    cout << path.size()-1 << "\n";
```

```
        for(auto it : path) {
            int px = it.first, py = it.second;
            for(int i = 0; i < 4; i++) {
            if(px + dx[i] == x && py + dy[i] == y) {
                cout << dir[i];
                break;
            }
            }
            x = px, y = py;
        }
        cout << "\n";
    }
    else {
        cout << "NO" << "\n";
    }
    return 0;
}
```

**Question No. 07**

Write the shortest distance from node E to every other node using the Dijkstra algorithm (the optimized version) for the following graph .You need to write all the steps. **15**



## Answer No. 07

| S.N. | A | B | C | E | F | G | H | I | J |
|---|---|---|---|---|---|---|---|---|---|
| — | inf. | inf. | inf. | 0 | inf. | inf. | inf. | inf. | inf. |
| E | inf. | 20 | 9 | **0** | inf. | 5 | inf. | inf. | inf. |
| G | inf. | 20 | 9 | **0** | 6 | **5** | inf. | inf. | inf. |
| F | inf. | 19 | 9 | **0** | **6** | **5** | inf. | inf. | inf. |
| C | 10 | 18 | **9** | **0** | **6** | **5** | 27 | inf. | inf. |
| 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** |

| Question No. 08 | |
|---|---|
| What is the recursive case in a recursive function and how does it relate to the base case? Explain it. | **10** |

**Answer No. 08**

In a recursive function,

**The recursive case** is the part of the function that calls itself with a smaller or simpler input than the original input. This is done until the function reaches a base case.

**The base case** is the condition in the recursive function where it does not call itself again and instead returns a result.

The relationship between the recursive case and the base case is that the recursive case is used to break down the problem into smaller or simpler sub-problems until it reaches the base case. Without a base case, the recursive function would continue to call itself infinitely and result in an error. Therefore, **a base case is necessary to terminate the recursive function and provide a result.**