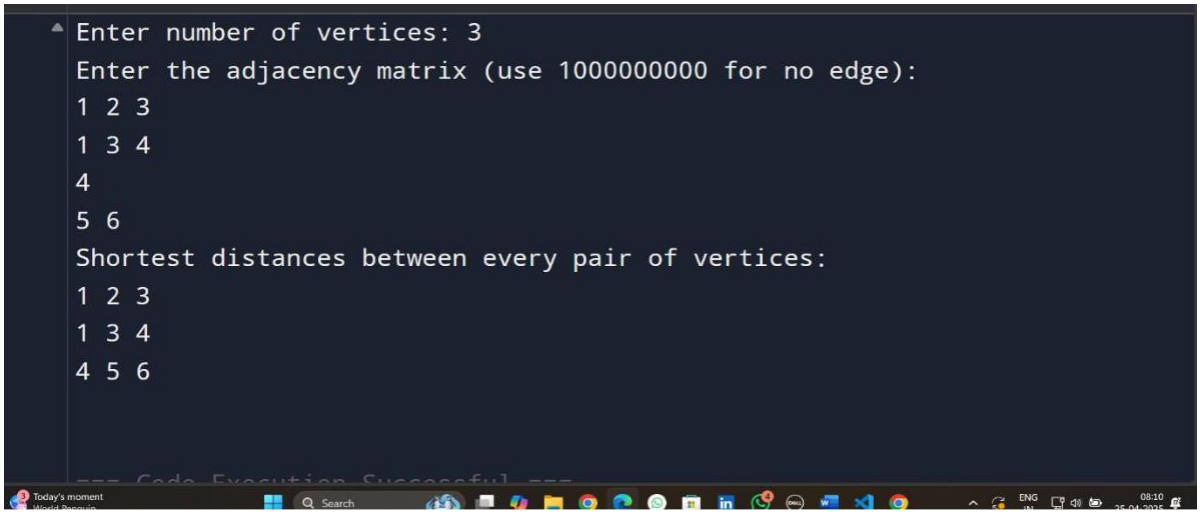
FLOYD-WARSHALL ALGORITHM:

```
#include <iostream>
#include <vector>
using namespace std;

const int INF = 1e9;

void floydWarshall(vector<vector<int>>& dist, int V)
   { for (int k = 0; k < V; ++k) {
      for (int i = 0; i < V; ++i) {
         for (int j = 0; j < V; ++j) {
            if (dist[i][k] < INF && dist[k][j] < INF)
               dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
         }
      }
   }
}

void printDistanceMatrix(const vector<vector<int>>& dist)
   { int V = dist.size();
   for (int i = 0; i < V; ++i) {
      for (int j = 0; j < V; ++j) {
         if (dist[i][j] == INF)
            cout << "INF ";
         else
            cout << dist[i][j] << " ";
      }
      cout << "\n";
   }
}
```

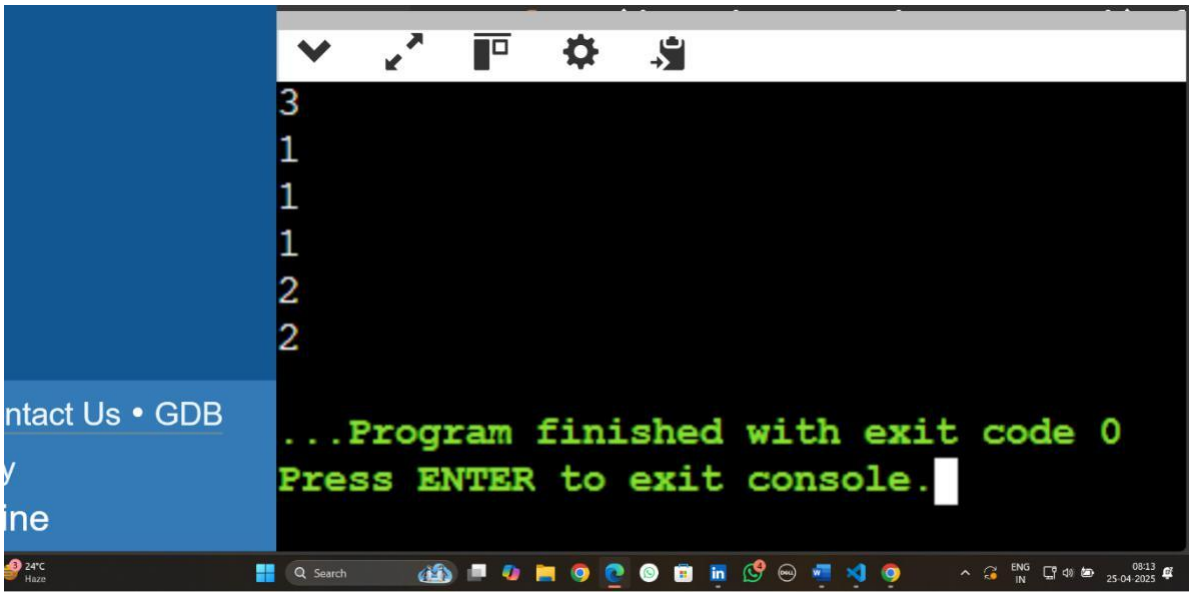
```
int main() {
    int V;
    cin >> V;
    vector<vector<int>> dist(V, vector<int>(V));
    for (int i = 0; i < V; ++i)
        for (int j = 0; j < V; ++j)
            cin >> dist[i][j];
    floydWarshall(dist, V);
    printDistanceMatrix(dist);
    return 0;
}
```



BFS IMPLEMENTATION:

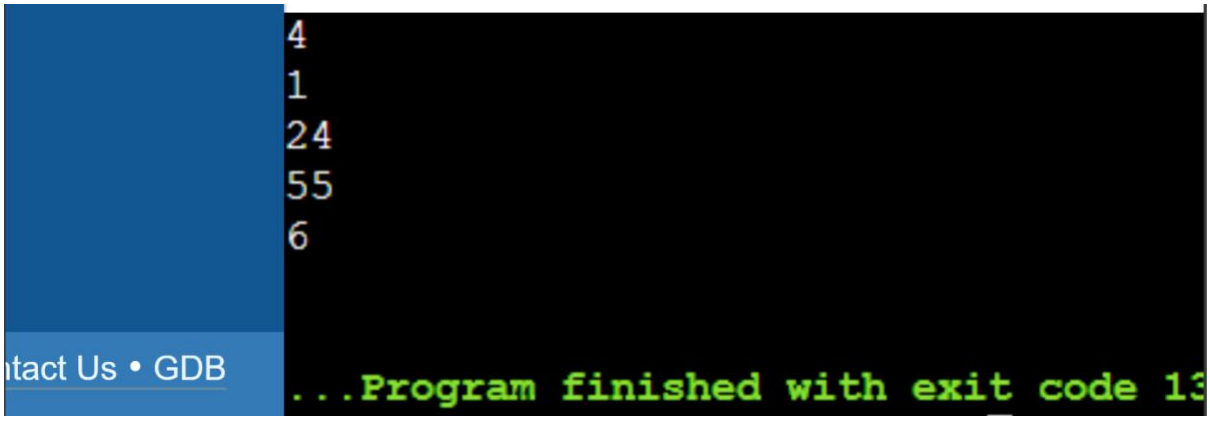
```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

void bfs(int start, vector<vector<int>>& adj, vector<bool>& visited)
   { queue<int> q;
   q.push(start);
   visited[start] = true;

   while (!q.empty()) {
      int node = q.front();
      q.pop();
      cout << node << " ";

      for (int neighbor : adj[node]) {
         if (!visited[neighbor]) {
            visited[neighbor] = true;
            q.push(neighbor);
         }
      }
   }
}

int main() {
   int V, E;
   cin >> V >> E;
   vector<vector<int>> adj(V);
   for (int i = 0; i < E; ++i) {
      int u, v;
```

```
        cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }

    vector<bool> visited(V, false);
    int start;
    cin >> start;
    bfs(start, adj, visited);

    return 0;
}
```



DFS IMPLEMENTATION:

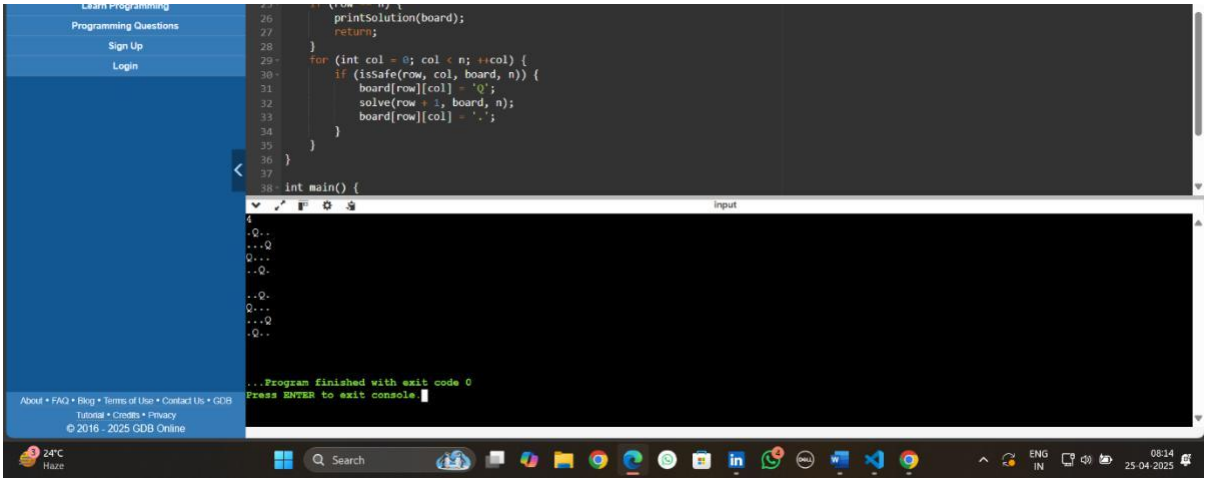
```
#include <iostream>
#include <vector>

using namespace std;
void dfs(int node, vector<vector<int>>& adj, vector<bool>& visited)
   { visited[node] = true;
   cout << node << " ";
   for (int neighbor : adj[node]) {
      if (!visited[neighbor])
         dfs(neighbor, adj, visited);
   }
}

int main() {
   int V, E;
   cin >> V >> E;
   vector<vector<int>> adj(V);
   for (int i = 0; i < E; ++i) {
      int u, v;
      cin >> u >> v;
      adj[u].push_back(v);
      adj[v].push_back(u); // Remove for directed graph
   }
   vector<bool> visited(V, false);
   int start;
   cin >> start;
   dfs(start, adj, visited);
   return 0;
}
```



N QUEENS :

```
#include <iostream>
#include <vector>
using namespace std;
void printSolution(const vector<string>& board) {
   for (const string& row : board)
      cout << row << "\n";
   cout << "\n";
}
bool isSafe(int row, int col, const vector<string>& board, int n)
   { for (int i = 0; i < row; ++i)
      if (board[i][col] == 'Q') return false;
   for (int i = row - 1, j = col - 1; i >= 0 && j >= 0; --i, --j)
      if (board[i][j] == 'Q') return false;
   for (int i = row - 1, j = col + 1; i >= 0 && j < n; --i, ++j)
      if (board[i][j] == 'Q') return false;
   return true;
}
void solve(int row, vector<string>& board, int n) {
   if (row == n) {
      printSolution(board);
      return;
   }
   for (int col = 0; col < n; ++col) {
      if (isSafe(row, col, board, n)) {
         board[row][col] = 'Q';
         solve(row + 1, board, n);
         board[row][col] = '.';
      }
```

```
    }
}

int main() {
    int n;
    cin >> n;
    vector<string> board(n, string(n, '.'));
    solve(0, board, n);
    return 0;
}
```



TRAVELLING SALESMAN PROBLEM:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
   int n;
   cin >> n;
   vector<vector<int>> dist(n, vector<int>(n));
   for (int i = 0; i < n; ++i)
      for (int j = 0; j < n; ++j)
         cin >> dist[i][j];

   vector<int> cities;
   for (int i = 1; i < n; ++i)
      cities.push_back(i);

   int minCost = INT_MAX;

   do {
      int cost = 0;
      int curr = 0;
      for (int i = 0; i < cities.size(); ++i) {
         cost += dist[curr][cities[i]];
         curr = cities[i];
      }
      cost += dist[curr][0];
      minCost = min(minCost, cost);
   } while (next_permutation(cities.begin(), cities.end()));
```

```
    cout << minCost << endl;
    return 0;
}
```