GRAPHS: -

Problem Statement

Sanju loves graph theory and has recently started learning about **Breadth-First Search** (BFS) algorithms. He is given a directed graph represented as an adjacency matrix, and he wants to implement the BFS algorithm to traverse the graph starting from a given vertex.

Input format :

The first line of input consists of the two integers, V and E, separated by a space, representing the number of vertices and edges in the network, respectively.

The next E lines consist of two integers, u and v, separated by a space, representing an edge between vertex u and vertex v.

Output format :

The output displays a single line containing the vertices visited during the BFS traversal, separated by a space.

Code constraints :

The test cases will fall under the following constraints:

1 <= V <= 10

0 <= E <= V*(V-1)/2

0 <= u, v < V

Sample test cases :

Input 1 :

```
5 4
0 1
0 2
0 3
2 4
```

Output 1 :

```
0 1 2 3 4
```

Input 2 :

```
3 2
0 1
0 2
```

Output 2 :

```
0 1 2
```

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

#define MAX_V 100

void enqueue(int* queue, int& rear, int vertex) {
  queue[++rear] = vertex;
}

int dequeue(int* queue, int& front) {
  return queue[++front];
}

void bfsOfGraph(int V, int adjList[MAX_V][MAX_V]) {
  int visited[MAX_V] = {0};
  int queue[MAX_V];
  int front = 0, rear = 0;

  int startVertex = 0;
  enqueue(queue, rear, startVertex);
  visited[startVertex] = 1;

  while (front < rear) {
    int currVertex = dequeue(queue, front);

    cout << currVertex << " ";
```

```
    for (int i = 0; i < V; ++i) {
      if (adjList[currVertex][i] == 1 && visited[i] == 0) {
        enqueue(queue, rear, i);
        visited[i] = 1;
      }
    }
  }
}

int main() {
  int V, E;
  cin >> V >> E;

  int adjList[MAX_V][MAX_V] = {0};

  for (int i = 0; i < E; ++i) {
    int u, v;
    cin >> u >> v;
    adjList[u][v] = 1;
  }

  bfsOfGraph(V, adjList);

  return 0;
}
```

Problem Statement

Siddhu works for a logistics company that handles a complex supply chain. This supply chain includes many suppliers, warehouses, and retail stores, all connected through different routes and transportation methods.

Your job is to assist him in creating a program that utilizes **Breadth-First Search** (BFS) traversal to improve the supply chain's efficiency. The program will find the best route for delivering products from a supplier to a retail store.

Input format :

The first line consists of two integers, V and E, separated by a space, where V represents the number of vertices (locations, suppliers, warehouses, and retail stores), and E represents the number of edges (connections between these locations).

The next E lines each consist of two integers, u and v, separated by a space, representing an edge between location u and location v.

Output format :

The output prints the BFS traversal order of the supply chain, separated by spaces.

Code constraints :

The test cases will fall under the following constraints:

1 <= V <= 10

0 <= E <= V*(V-1)/2

0 <= u, v < V

Sample test cases :

Input 1 :

```
6 8
0 1
0 2
1 3
2 3
2 4
3 4
4 5
5 0
```

Output 1 :

```
0 1 2 3 4 5
```

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

#define MAX_V 100

void enqueue(int* queue, int& rear, int vertex) {
  queue[++rear] = vertex;
}

int dequeue(int* queue, int& front) {
  return queue[++front];
}

void bfsOfGraph(int V, int adjList[MAX_V][MAX_V]) {
  int visited[MAX_V] = {0};
  int queue[MAX_V];
  int front = 0, rear = 0;

  int startVertex = 0;
  enqueue(queue, rear, startVertex);
  visited[startVertex] = 1;

  while (front < rear) {
    int currVertex = dequeue(queue, front);

    cout << currVertex << " ";
```

```
    for (int i = 0; i < V; ++i) {
      if (adjList[currVertex][i] == 1 && visited[i] == 0) {
        enqueue(queue, rear, i);
        visited[i] = 1;
      }
    }
  }
}

int main() {
  int V, E;
  cin >> V >> E;

  int adjList[MAX_V][MAX_V] = {0};

  for (int i = 0; i < E; ++i) {
    int u, v;
    cin >> u >> v;
    adjList[u][v] = 1;
  }

  bfsOfGraph(V, adjList);

  return 0;
}
```

Problem Statement

Raja is in the process of developing a program that aims to analyze a social network and discover the longest chain of connections among its users.

Your task is to help him design a program that utilizes the **Depth-First Search** (DFS) algorithm to identify the most extended chain of friendships within the network.

Input format :

The first line of input consists of two integers "n" and "m", representing the number of users and the number of friendship connections in the social network, respectively.

The second line of input consists of each contain two integers, "u" and "v," where "u" and "v" are the user IDs of two users in the network who are friends.

Output format :

The output displays a single integer representing the length of the longest friendship chain within the network.

Refer to the sample output for the formatting specifications.

Code constraints :

The test cases will fall under the following constraints:

$2 \leq n \leq 10$

$0 \leq m \leq n*(n-1)/2$

$1 \leq u, v \leq n, u \neq v$

Sample test cases :

Input 1 :

```
4 5
1 2
1 3
3 2
2 4
3 4
```

Output 1 :

```
3
```

Input 2 :

```
3 4
1 2
2 1
2 3
1 3
```

Output 2 :

```
2
```

```
#include <iostream>

#define MAXN 100

using namespace std;

int dfs(int node, int adj[][MAXN], int dp[], bool vis[], int n) {
    if (vis[node]) {
        return dp[node];
    }

    vis[node] = true;
    int maxPath = 0;

    for (int i = 0; i < n; i++) {
        if (adj[node][i]) {
            maxPath = max(maxPath, 1 + dfs(i, adj, dp, vis, n));
        }
    }

    dp[node] = maxPath;
    return maxPath;
}

void addEdge(int adj[][MAXN], int u, int v) {
    adj[u][v] = 1;
}

int findLongestPath(int adj[][MAXN], int n) {
```

```
    int dp[MAXN] = {0};
    bool vis[MAXN] = {false};

    int longestPath = 0;

    for (int i = 0; i < n; i++) {
        if (!vis[i]) {
            longestPath = max(longestPath, dfs(i, adj, dp, vis, n));
        }
    }

    return longestPath;
}

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

    int adj[MAXN][MAXN] = {0};
    cin >> m;

    for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;
        addEdge(adj, u - 1, v - 1);
    }

    cout << findLongestPath(adj, n);
    return 0;
}
```

Problem Statement

Manoj is a ship captain on a mission to explore a network of islands. The islands are connected through various routes, and Manoj wants to plan his exploration using **depth-first traversal**.

Given a map of the island network and a starting island, help Manoj write a program using depth-first traversal to explore the islands in a particular order.

Input format :

The first line of input consists of the number of islands in the network, represented by V, and the number of routes between islands, represented by E, separated by a space.

The next E lines consist of two integers each: v and w, representing a route between island v and island w.

The last line of input consists of the starting island for the depth-first traversal, represented by startVertex.

Output format :

The output displays **"Depth First Traversal starting from vertex [startVertex]:"** followed by a space-separated list of island indices representing the order in which Manoj explores the islands using DFS.

Refer to the sample output for the formatting specifications.

Code constraints :

The test cases will fall under the following constraints:

$2 \leq V \leq 10$

$0 \leq E \leq V*(V-1)/2$

$0 \leq v, w < V$

$0 \leq startVertex < V$

Sample test cases :

Input 1 :

```
3 2
0 1
1 2
0
```

Output 1 :

```
Depth First Traversal starting from vertex 0:
0 1 2
```

Input 2 :

```
4 5
0 1
0 2
1 2
2 0
2 1
1
```

Output 2 :

```
Depth First Traversal starting from vertex 1:
```

```
1 2 0
```

```
#include <iostream>
using namespace std;
#define MAX_VERTICES 100

void addEdge(int adj[MAX_VERTICES][MAX_VERTICES], int v, int w) {
  adj[v][w] = 1;
}

void DFS(int adj[MAX_VERTICES][MAX_VERTICES], int visited[MAX_VERTICES], int
V, int v) {
  visited[v] = 1;
  cout << v << " ";

  for (int i = 0; i < V; ++i) {
    if (adj[v][i] && !visited[i]) {
      DFS(adj, visited, V, i);
    }
  }
}

int main() {
  int V, E;
  cin >> V;
  cin >> E;

  int adj[MAX_VERTICES][MAX_VERTICES] = {0};
  int visited[MAX_VERTICES] = {0};

  for (int i = 0; i < E; ++i) {
    int v, w;
    cin >> v >> w;
    addEdge(adj, v, w);
  }

  int startVertex;
  cin >> startVertex;

  cout << "Depth First Traversal starting from vertex " << startVertex << ":\n";
  DFS(adj, visited, V, startVertex);

  return 0;}
```

Problem Statement

Shekin is developing a program to assist a robot in navigating a warehouse. The warehouse is depicted as a grid, with specific cells representing obstacles that the robot must avoid.

Your task is to create a program that uses **Depth-First Search** (DFS) to determine a path for the robot to travel from an initial position to a designated target location within the warehouse.

Input format :

The first line of input consists of two integers separated by a space: V and E, where V represents the number of vertices (cells) in the warehouse grid and E represents the number of edges (connections between cells) indicating possible paths.

The next E lines each consist of two integers, v and w, separated by a space, representing an edge connecting cell v to cell w.

The last line consists of a single integer, startVertex, which is the starting position of the robot within the warehouse.

Output format :

The output should print a message in the format **"Depth First Traversal starting from vertex [startVertex]:",** where [startVertex] is the integer representing the initial cell.

Following that next line, it should list the visited cells in DFS traversal order, separated by spaces.

Refer to the sample output for the exact format.

Code constraints :

The test cases will fall under the following constraints:

$2 \leq V \leq 10$

0 <= E <= V*(V-1)/2

0 <= v, w < V

0 <= startVertex <= V

Sample test cases :

Input 1 :

```
6 6
0 1
0 2
1 3
2 4
3 5
4 5
0
```

Output 1 :

```
Depth First Traversal starting from vertex 0:
0 1 3 5 2 4
```

Input 2 :

```
5 4
0 1
0 2
1 3
3 4
0
```

Output 2 :

```
Depth First Traversal starting from vertex 0:
0 1 3 4 2
```

```
#include <iostream>
using namespace std;
#define MAX_VERTICES 100

void addEdge(int adj[MAX_VERTICES][MAX_VERTICES], int v,
int w) {
  adj[v][w] = 1;
}

void DFS(int adj[MAX_VERTICES][MAX_VERTICES], int
visited[MAX_VERTICES], int V, int v) {
  visited[v] = 1;
  cout << v << " ";

  for (int i = 0; i < V; ++i) {
    if (adj[v][i] && !visited[i]) {
      DFS(adj, visited, V, i);
    }
  }
}

int main() {
  int V, E;
```

```
  cin >> V;
  cin >> E;

  int adj[MAX_VERTICES][MAX_VERTICES] = {0};
  int visited[MAX_VERTICES] = {0};

  for (int i = 0; i < E; ++i) {
    int v, w;
    cin >> v >> w;
    addEdge(adj, v, w);
  }

  int startVertex;
  cin >> startVertex;

  cout << "Depth First Traversal starting from vertex " <<
startVertex << ":\n";
  DFS(adj, visited, V, startVertex);

  return 0;
}
```

Problem Statement

Parthi is interested in graph theory and wants to determine if there exists a valid path from a given starting vertex to an ending vertex in a graph. He decided to use **Depth-First Search** (DFS) to solve this problem.

Given a graph with vertices and edges, your task is to help Parthi write a program that determines if there is a valid path from a specified starting vertex to an ending vertex.

A valid path is a sequence of vertices such that there is an edge between consecutive vertices in the sequence.

Input format :

The first line of input consists of the two integers n and m, representing the number of vertices and edges in the graph, respectively, separated by a space.

The next m lines consist of two integers u and v, representing an undirected edge between vertices u and v.

The last two lines of input consist of the two integers start and end, representing the starting and ending vertices, respectively.

Output format :

The output consists of the following format:

If there is a valid path from the starting vertex to the ending vertex, print: **"There is a path from [start] to [end]".**

If there is no valid path from the starting vertex to the ending vertex, print: **"There is no path from [start] to [end]".**

Refer to the sample output for the formatting specifications.

Code constraints :

The test cases will fall under the following constraints:

$1 \leq n \leq 10$, $0 \leq m \leq n*(n-1)/2$

$1 \leq u, v \leq n$, $u \neq v$

$1 \leq start, end \leq n$, $start \neq end$

Sample test cases :

Input 1 :

```
5 6
0 1
1 2
2 3
3 4
0 3
2 4
1
3
```

Output 1 :

```
There is a path from 1 to 3
```

Input 2 :

```
5 6
0 1
1 2
2 3
3 4
0 3
2 4
0
4
```

Output 2 :

```
There is a path from 0 to 4
```

Input 3 :

```
5 4
0 1
1 2
2 3
3 4
0
5
```

Output 3 :

```
There is no path from 0 to 5
```

```
#include <iostream>

using namespace std;

bool dfs(int adjMatrix[][100], int n, int start, int end, bool visited[]) {
    if (start == end)
        return true;

    visited[start] = true;

    for (int i = 0; i < n; ++i) {
        if (adjMatrix[start][i] && !visited[i] && dfs(adjMatrix, n, i, end,
visited))
            return true;
    }

    return false;
}

bool validPath(int n, int edges[][2], int m, int start, int end) {
    int adjMatrix[100][100] = {0};

    for (int i = 0; i < m; ++i) {
        int u = edges[i][0];
        int v = edges[i][1];
        adjMatrix[u][v] = 1;
        adjMatrix[v][u] = 1;
    }
```

```
    bool visited[100] = {false};

    return dfs(adjMatrix, n, start, end, visited);
}

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

    int edges[m][2];
    for (int i = 0; i < m; ++i) {
        cin >> edges[i][0] >> edges[i][1];
    }

    int start, end;
    cin >> start >> end;

    if (validPath(n, edges, m, start, end)) {
        cout << "There is a path from " << start << " to " << end << endl;
    } else {
        cout << "There is no path from " << start << " to " << end << endl;
    }

    return 0;
}
```

Problem Statement

Rohan is in the process of creating a delivery route optimization application tailored for a courier company. The primary goal of this application is to identify the most efficient route for delivering packages to multiple destinations situated within a city. In this city representation, delivery points serve as vertices, and the streets connecting them are depicted as edges in a graph.

Your task is to develop a program that utilizes the **Breadth-First Search** (BFS) algorithm to calculate the shortest delivery route, starting from the company's warehouse and passing through all specified delivery locations before returning to the warehouse.

Input format :

The first line consists of an integer v, representing the number of locations within the city.
The second line consists of an integer e, representing the number of streets connecting these locations.
The next e-line consists of two space-separated integers src and dest, representing streets connecting location "src" to location "dest."
The next line consists of an integer, representing the starting location (source), which corresponds to the company's warehouse.
The last line consists of an integer, destination, which represents the list of delivery locations that the courier needs to visit, arranged in the order they should be visited.

Output format :

The output is displayed in the following format:

The first line consists of "Shortest path length is: X", where X is the shortest path length.
The second line consists of "Path is: ", followed by the delivery locations and the distance covered, separated by spaces.

Refer to the sample output for the exact format.

Code constraints :

The test cases will fall under the following constraints:
$1 \leq v \leq 10$
$0 \leq e \leq v * (v - 1) / 2$
$0 \leq$ source, destination $< v$

Sample test cases :

Input 1 :

```
8
9
0 1
0 2
1 3
1 4
2 5
2 6
3 7
4 7
5 7
3
7
```

Output 1 :

```
Shortest path length is: 1
Path is: 3 7
```

```
#include <iostream>
#include <climits>

using namespace std;

void add_edge(int adj[][100], int src, int dest)
{
   adj[src][dest] = 1;
   adj[dest][src] = 1;
}

bool BFS(int adj[][100], int src, int dest, int v,
         int pred[], int dist[])
{
   int queue[v];
   int front = -1, rear = -1;

   bool visited[v];

   for (int i = 0; i < v; i++)
   {
      visited[i] = false;
      dist[i] = INT_MAX;
      pred[i] = -1;
   }

   visited[src] = true;
   dist[src] = 0;
   queue[++rear] = src;

   while (front != rear)
   {
      int u = queue[++front];
      for (int i = 0; i < v; i++)
```

```
        {
            if (adj[u][i] && !visited[i])
            {
                visited[i] = true;
                dist[i] = dist[u] + 1;
                pred[i] = u;
                queue[++rear] = i;

                if (i == dest)
                    return true;
            }
        }
    }

    return false;
}

void printShortestDistance(int adj[][100], int s,
                           int dest, int v, int pred[], int dist[])
{
    int path[v];
    int crawl = dest;
    int pathLength = 0;
    path[pathLength++] = crawl;

    while (pred[crawl] != -1)
    {
        path[pathLength++] = pred[crawl];
        crawl = pred[crawl];
    }

    cout << "Shortest path length is: " << dist[dest] << endl;

    cout << "Path is: ";
```

```
    for (int i = pathLength - 1; i >= 0; i--)
        cout << path[i] << " ";
}

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

    int adj[100][100] = {0};

    cin >> e;

    for (int i = 0; i < e; i++)
    {
        int src, dest;
        cin >> src >> dest;
        add_edge(adj, src, dest);
    }

    int source, dest;
    cin >> source >> dest;

    int pred[v], dist[v];

    if (BFS(adj, source, dest, v, pred, dist))
        printShortestDistance(adj, source, dest, v, pred, dist);

    return 0;
}
```

---Unit 1 completed—

Problem Statement

Hannah is given the task of analyzing graphs and determining the reachability of nodes within the graph.

Your goal is to write a program to help Hannah solve this problem efficiently using Warshall's Algorithm.

A graph can be represented using an adjacency matrix. Given such an adjacency matrix and two vertices, **u** and **v,** the program should be able to answer whether there exists a path from vertex **u** to vertex **v**. Additionally, the program should compute the reachability of all vertices from each other within the graph.

Write a program to accomplish these tasks.

Input format :

The first line of input consists of an integer **N,** representing the number of vertices in the graph.

The following **N** lines consist of **N** space-separated integers, forming the adjacency matrix graph, where graph[i][j] is either 0 or 1.

The last line of input consists of two integers, **u** and **v**, representing the pair of vertices for which you need to check the reachability.

Output format :

The output prints the following:

1. An NxN matrix represents the reachability matrix for all pairs of vertices. Each element in the matrix should be 1 if there is a path from the corresponding row vertex to the column vertex, and 0 otherwise.
2. A message indicating whether there is a path from vertex **u** to vertex **v** in the graph If there is a path, print "Path Exists". Otherwise, print "Path does not Exist"

Refer to the sample output for the formatting specifications.

Code constraints :

The test cases will fall under the following constraints:

$1 \leq N \leq 10$

Each element of the adjacency matrix is either 0 or 1.

$0 \leq u, v < N$ (source and destination vertices).

Sample test cases :

Input 1 :

```
4
0 1 0 0
0 0 1 0
0 0 0 1
0 0 0 0
1 3
```

Output 1 :

```
0 1 1 1
0 0 1 1
0 0 0 1
0 0 0 0
Path Exists
```

Input 2 :

```
4
0 1 0 0
0 0 1 0
0 0 0 1
0 0 0 0
3 1
```

Output 2 :

```
0 1 1 1
0 0 1 1
0 0 0 1
0 0 0 0
Path does not Exist
```

```
#include <iostream>
```

```
using namespace std;

bool isReachable(int **graph, int u, int v, int n) {
    return graph[u][v] == 1;
}

void computeReachability(int **graph, int n) {
    for (int k = 0; k < n; ++k) {
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                graph[i][j] = graph[i][j] || (graph[i][k] && graph[k][j]);
            }
        }
    }
}

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

    // Dynamically allocate memory for the adjacency matrix
    int **graph = new int *[n];
    for (int i = 0; i < n; ++i) {
        graph[i] = new int[n];
    }

    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            cin >> graph[i][j];
        }
```

```
    }

    computeReachability(graph, n);

    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            cout << graph[i][j] << " ";
        }
        cout << endl;
    }
    int u, v;
    cin >> u >> v;

    if (isReachable(graph, u, v, n)) {
        cout << "Path Exists";
    } else {
        cout << "Path does not Exist";
    }

    // Deallocate dynamically allocated memory
    for (int i = 0; i < n; ++i) {
        delete[] graph[i];
    }
    delete[] graph;

    return 0;
}
```

Problem Statement

You are given **N** courses numbered from **0** to **N-1.** Some courses have prerequisites that need to be completed before taking the course. The prerequisites are given in pairs (course, prerequisite) in a list.

You need to determine if it is possible to complete all the courses based on the given prerequisites using Warshall's Algorithm to check the course completion possibility.

Example 1

Input:
4
0 1
0 2
1 2
2 3

Output:
It is possible to complete all courses based on the given prerequisites.

Explanation:
Course 0 requires Course 1 and Course 2.
Course 1 requires Course 2.
Course 2 requires Course 3.
There are no circular dependencies or cycles, so it is possible to complete all courses.

Example 2

Input:
3
1 0
2 1
0 2

Output:
It is not possible to complete all courses based on the given prerequisites.

Explanation:
Course 0 requires Course 2.
Course 1 requires Course 0.
Course 2 requires Course 1.

This circular dependency means that you cannot complete all courses because you would be stuck in an infinite loop of prerequisites. So it is not possible to complete all courses.

Input format :

The first line of input consists of an integer **N**, representing the number of courses.
The following **N** lines consist of two space-separated integers each, representing the prerequisite relationship between courses.
For each line, the first integer represents the course number, and the second integer represents its prerequisite course number.

Output format :

If it is possible to complete all courses based on the given prerequisites, print "It is possible to complete all courses based on the given prerequisites."
Otherwise, print "It is not possible to complete all courses based on the given prerequisites."

Refer to the sample output for the formatting specifications.

Code constraints :

The test cases will fall under the following constraints:
$1 \leq N \leq 10$

Sample test cases :

Input 1 :

```
4
0 1
0 2
1 2
2 3
```

Output 1 :

```
It is possible to complete all courses based on the given prerequisites.
```

Input 2 :

```
3
1 0
2 1
0 2
```

Output 2 :

```
It is not possible to complete all courses based on the given prerequisites.
```

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

int MAX_COURSES = 100;

bool canCompleteAllCourses(int n, int prerequisites[][2]) {
    // Initialize the adjacency matrix with zeros (no prerequisites initially)
    bool graph[MAX_COURSES][MAX_COURSES] = {false};

    // Set the prerequisites in the adjacency matrix
    for (int i = 0; i < n; ++i) {
        int course = prerequisites[i][0];
        int prerequisite = prerequisites[i][1];
        graph[course][prerequisite] = true;
    }

    // Applying Warshall's algorithm to update the graph
    for (int k = 0; k < n; ++k) {
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                graph[i][j] = graph[i][j] || (graph[i][k] && graph[k][j]);
            }
        }
    }

    // Check if there is a path from each course to itself (a cycle)
    for (int i = 0; i < n; ++i) {
        if (graph[i][i]) {
            return false;
        }
```

```
    }

    return true;
}

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

    int prerequisites[MAX_COURSES][2];
    for (int i = 0; i < n; ++i) {
        cin >> prerequisites[i][0] >> prerequisites[i][1];
    }

    if (canCompleteAllCourses(n, prerequisites)) {
        cout << "It is possible to complete all courses based on the given prerequisites.";
    } else {
        cout << "It is not possible to complete all courses based on the given
prerequisites.";
    }

    return 0;
}
```

Problem Statement

You are attending a party where there are several people, and some of them know each other.

You are given a matrix representing the acquaintances among the attendees. If cell (i, j) in the matrix is 1, it means that person i knows person j, and if it is 0, it means they do not know each other.

Your task is to find out if there is a celebrity in the party using Warshall's Algorithm.

A celebrity is someone who is known by everyone else at the party but does not know anyone themselves. If such a celebrity exists, your goal is to identify their name in the list of attendees.

Write a program that applies Warshall's Algorithm to determine the presence of a celebrity at the party and, if one exists, output their index in the list. Otherwise, indicate that there is no celebrity at the party.

Input format :

The first line of input consists of an integer **N**, representing the number of people attending the party.

The following **N** lines consist of **N** space-separated integers, representing the acquaintance matrix.

Output format :

If a celebrity is present at the party, output a single line: "A celebrity is present at index X in the list of attendees," where X is the index of the celebrity in the list (0-based index).

If there is no celebrity at the party, output a single line: "There is no celebrity at the party."

Refer to the sample output for the formatting specifications.

Code constraints :

The test cases will fall under the following constraints:

1 <= N <= 10

1 <= i, j <= n

1 <= x <= n

Sample test cases :

Input 1 :

```
4
0 1 1 0
0 0 1 0
0 0 0 0
0 0 1 0
```

Output 1 :

```
A celebrity is present at index 2 in the list of attendees.
```

Input 2 :

```
5
0 1 1 0 0
0 0 0 0 0
0 0 0 1 0
0 0 0 0 1
0 0 0 0 0
```

Output 2 :

```
There is no celebrity at the party.
```

Input 3 :

```
3
0 1 1
0 0 0
0 1 0
```

Output 3 :

```
A celebrity is present at index 1 in the list of attendees.
```

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

#define MAX_N 100

int findCelebrity(int acquaintances[MAX_N][MAX_N], int n) {
   int transitiveClosure[MAX_N][MAX_N];

   for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j) {
         transitiveClosure[i][j] = acquaintances[i][j];
      }
   }

   for (int k = 0; k < n; ++k) {
      for (int i = 0; i < n; ++i) {
         for (int j = 0; j < n; ++j) {
            transitiveClosure[i][j] = transitiveClosure[i][j] || (transitiveClosure[i][k] &&
transitiveClosure[k][j]);
         }
      }
   }

   for (int i = 0; i < n; ++i) {
      bool isCelebrity = true;
      for (int j = 0; j < n; ++j) {
         if (i != j && (transitiveClosure[i][j] || !transitiveClosure[j][i])) {
            isCelebrity = false;
            break;
         }
      }
```

```
        if (isCelebrity)
            return i;
    }

    return -1;
}

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

    int acquaintances[MAX_N][MAX_N];
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            cin >> acquaintances[i][j];
        }
    }

    int celebrityIndex = findCelebrity(acquaintances, n);

    if (celebrityIndex != -1) {
        cout << "A celebrity is present at index " << celebrityIndex << " in the list of attendees.";
    } else {
        cout << "There is no celebrity at the party.";
    }

    return 0;
}
```

Problem Statement

You are given a map of cities and the highways that connect them. Your task is to determine whether there is a chain of highways that connects two specific cities.

Each city is represented by a unique integer from **0** to **N-1,** where **N** is the total number of cities. The highways are represented as a matrix of size NxN, where the cell (i, j) contains a 1 if there is a direct highway connecting city i to city j, and 0 otherwise.

Write a program using Warshall's Algorithm to find out if there exists a chain of highways that connects City 1 to City 2.

Input format :

The first line of input consists of an integer **N,** representing the number of cities.
The following **N** lines consist of **N** space-separated integers (0 or 1), representing the matrix of direct highways between cities.
The last line consists of two integers **City1** and **City2** for which the presence of a chain of highways needs to be determined.

Output format :

If there is a chain of highways connecting City1 and City2, print "There is a chain of highways connecting City 1 and City 2."
Otherwise, print "There is no chain of highways connecting City 1 and City 2."

Refer to the sample output for the formatting specifications.

Code constraints :

The test cases will fall under the following constraints:
$1 \leq N \leq 10$
Each element of the matrix is either 0 or 1.

Sample test cases :

Input 1 :

```
3
0 1 0
0 0 1
1 0 0
1 2
```

Output 1 :

```
There is a chain of highways connecting City 1 and City 2.
```

Input 2 :

```
4
0 1 0 0
0 0 0 1
0 0 0 1
0 0 0 0
2 1
```

Output 2 :

```
There is no chain of highways connecting City 2 and City 1.
```

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

const int MAX_CITIES = 100;

bool hasChainOfHighways(int n, int highways[][MAX_CITIES], int city1, int city2) {
    bool distance[MAX_CITIES][MAX_CITIES];

    // Step 1: Initialize the distance matrix with the given highways matrix
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            distance[i][j] = highways[i][j];
        }
    }

    // Step 2: Applying the Warshall algorithm
    for (int k = 0; k < n; ++k) {
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                distance[i][j] = distance[i][j] || (distance[i][k] && distance[k][j]);
            }
        }
    }

    // Step 3: Return the result (whether there is a chain of highways between city1 and city2)
    return distance[city1][city2];
}

int main() {
    int n; // Number of cities
```

```
    cin >> n;

    int highways[MAX_CITIES][MAX_CITIES] = {0};

    // Taking input for direct highways
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            cin >> highways[i][j];
        }
    }

    int city1, city2;
    cin >> city1 >> city2;

    if (hasChainOfHighways(n, highways, city1, city2)) {
        cout << "There is a chain of highways connecting City " << city1 << " and City "
<< city2 << ".";
    } else {
        cout << "There is no chain of highways connecting City " << city1 << " and City "
<< city2 << ".";
    }

    return 0;
}
```

--unit 2 completed—

Problem Statement

Latha is studying graph theory and is currently learning about detecting negative cycles in a weighted directed graph. She is interested in implementing the Floyd-Warshall algorithm to determine if a given graph contains any negative cycles.

Write a program to help Latha determine whether the given weighted directed graph contains any negative cycles.

Input format :

The first line of input consists of two space-separated integers, **V** and **E**. V represents the number of vertices, and E represents the number of edges in the graph.

The next lines each contain three space-separated integers, **u**, **v**, and **w**, representing a directed edge from vertex **u** to vertex **v** with weight **w**.

Output format :

If there is a negative cycle in the graph, print "Yes"

Otherwise, print "No"

Code constraints :

The test cases will fall under the following constraints:

$1 \leq V \leq 4$

$0 \leq E \leq V*(V-1)/2$

$0 \leq u, v < V$

$-100 \leq w \leq 100$

Sample test cases :

Input 1 :

```
4 3
0 1 1
1 2 -1
2 3 -1
```

Output 1 :

```
Yes
```

Input 2 :

```
4 7
0 1 1
1 2 1
2 3 1
3 1 1
3 0 1
1 0 -3
3 1 -1
```

Output 2 :

```
No
```

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

#define V 4
#define INF 99999

void printSolution(int dist[][V]);

bool negCyclefloydWarshall(int graph[][V])
{
   int dist[V][V], i, j, k;

   for (i = 0; i < V; i++)
      for (j = 0; j < V; j++)
         dist[i][j] = graph[i][j];

   for (k = 0; k < V; k++)
   {
      for (i = 0; i < V; i++)
      {
         for (j = 0; j < V; j++)
         {
            if (dist[i][k] + dist[k][j] < dist[i][j])
               dist[i][j] = dist[i][k] + dist[k][j];
         }
      }
   }
```

```
    for (int i = 0; i < V; i++)
        if (dist[i][i] < 0)
            return true;
    return false;
}

int main()
{
    int graph[V][V];

    for (int i = 0; i < V; i++)
    {
        for (int j = 0; j < V; j++)
        {
            cin >> graph[i][j];
        }
    }

    if (negCyclefloydWarshall(graph))
        cout << "Yes";
    else
        cout << "No";

    return 0;
}
```

Problem Statement

Regi is studying graph theory and algorithms. He wants to determine whether a given directed graph is strongly connected using the Floyd-Warshall algorithm.

A strongly connected graph is a directed graph in which there is a path from every vertex to every other vertex.

Write a program that takes as input an adjacency matrix representing a directed graph and uses the Floyd-Warshall algorithm to determine if the graph is strongly connected.

Example 1

Input:
5
0 1 1 0 0
0 0 0 1 0
0 0 0 0 1
0 0 0 0 1
0 0 0 0 0

Output:
The graph is not strongly connected

Explanation:
This input represents a network with 5 devices and the following connections:
0 -> 1
0 -> 2
1 -> 3
2 -> 4
3 -> 4
Device 0 has a path to Devices 1, 2, 3, and 4.
Device 1 has a path to Devices 3 and 4.
Device 2 has a path to Device 4.
Device 3 has a path to Device 4.
Device 4 has no path to any other device.
As Device 4 does not have a path to any other device, the network is not strongly connected.

Example 2

Input:
4
0 1 0 0
0 0 1 0
0 0 0 1

1 0 0 0

Output:
The graph is strongly connected

Explanation:
This input represents a network with 4 devices and the following connections:
0 -> 1
1 -> 2
2 -> 3
3 -> 0
Device 0 has a path to Devices 1, 2, and 3.
Device 1 has a path to Devices 0, 2, and 3.
Device 2 has a path to Devices 0, 1, and 3.
Device 3 has a path to Devices 0, 1, and 2.
Since there is a path from every device to every other device, the network is strongly connected.

Input format :
The first line of input consists of an integer, **n**, representing the number of vertices in the graph.
The next **n** lines contain **n** space-separated integers, representing the adjacency matrix of the graph. The element graph[i][j] is 1 if there is a directed edge from vertex i to vertex j, and 0 otherwise.

Output format :
The output will be a single line indicating whether the graph is strongly connected or not.

Code constraints :
The test cases will fall under the following constraints:
$1 \leq n \leq 100$
$0 \leq graph[i][j] \leq 1$

Sample test cases :

Input 1 :
```
5
0 1 1 0 0
0 0 0 1 0
0 0 0 0 1
0 0 0 0 1
0 0 0 0 0
```

Output 1 :
```
The graph is not strongly connected
```

Input 2 :
```
4
0 1 0 0
0 0 1 0
0 0 0 1
1 0 0 0
```

Output 2 :
```
The graph is strongly connected
```

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

void floydWarshall(int n, int graph[][100]) {
   int INF = 1e9;

   int reachability[n][100];
   for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j) {
         reachability[i][j] = graph[i][j];
         if (i != j && graph[i][j] == 0) {
            reachability[i][j] = INF;
         }
      }
   }

   for (int k = 0; k < n; ++k) {
      for (int i = 0; i < n; ++i) {
         for (int j = 0; j < n; ++j) {
            if (reachability[i][k] < INF && reachability[k][j] < INF) {
               reachability[i][j] = min(reachability[i][j], reachability[i][k] +
reachability[k][j]);
            }
         }
      }
   }

   bool isStronglyConnected = true;
   for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j) {
         if (i != j && reachability[i][j] == INF) {
```

```
            isStronglyConnected = false;
            break;
        }
    }
}

if (isStronglyConnected) {
    cout << "The graph is strongly connected";
} else {
    cout << "The graph is not strongly connected";
}
}

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

    int graph[100][100];
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            cin >> graph[i][j];
        }
    }

    floydWarshall(n, graph);

    return 0;
}
```

Problem Statement

Varsha is a computer science student working on algorithms. She wants to develop a program that can efficiently find the shortest path from the top-left corner to the bottom-right corner of a 2D grid. Each cell in the grid represents a node, and the weight of the edges between adjacent nodes is given.

Varsha knows that the Floyd-Warshall algorithm can help her solve this problem effectively.

Help Varsha write a program that uses the Floyd-Warshall algorithm to find the shortest path from the top-left corner to the bottom-right corner in the given 2D grid.

Input format :

The first line of input consists of an integer **V**, representing the number of vertices in the grid (VxV grid).

The next **V** lines contain **V** space-separated integers, representing the adjacency matrix of the grid.

Each cell graph [i][j] indicates the weight of the edge between node i and node j.

If there is no direct edge (not reachable), the weight is represented by a value of INF (10000000).

Output format :

The output prints a single line representing the shortest path from the top-left corner (node 0) to the bottom-right corner (node V-1).

Print the nodes in the path, separated by "->".

Code constraints :

The test cases will fall under the following constraints:

$2 \leq V \leq 10$ (number of territories in the grid)

$0 \leq graph[i][j] \leq 10000000$

$0 \leq u, v \leq V-1$

Sample test cases :

Input 1 :

```
4
0 3 10000000 7
8 0 2 10000000
5 10000000 0 1
2 10000000 10000000 0
```

Output 1 :

```
0 -> 1 -> 2 -> 3
```

Input 2 :

```
3
0 1 2
1 0 3
2 3 0
```

Output 2 :

```
0 -> 2
```

Input 3 :

```
4
0 5 10000000 10
10000000 0 3 10000000
10000000 10000000 0 1
10000000 10000000 10000000 0
```

Output 3 :

```
0 -> 1 -> 2 -> 3
```

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

#define INF 10000000
#define MAXN 100

int dis[MAXN][MAXN];
int Next[MAXN][MAXN];

void initialise(int V, int graph[MAXN][MAXN]) {
   for (int i = 0; i < V; i++) {
      for (int j = 0; j < V; j++) {
         dis[i][j] = graph[i][j];
         if (graph[i][j] == INF)
            Next[i][j] = -1;
         else
            Next[i][j] = j;
      }
   }
}

void floydWarshall(int V) {
   for (int k = 0; k < V; k++) {
      for (int i = 0; i < V; i++) {
         for (int j = 0; j < V; j++) {
            if (dis[i][k] == INF || dis[k][j] == INF)
               continue;

            if (dis[i][j] > dis[i][k] + dis[k][j]) {
               dis[i][j] = dis[i][k] + dis[k][j];
               Next[i][j] = Next[i][k];
            }
         }
      }
   }
}
```

```
void printPath(int path[], int n) {
    for (int i = 0; i < n - 1; i++)
        cout << path[i] << " -> ";
    cout << path[n - 1] << endl;
}

int main() {
    int V;
    cin >> V;

    int graph[MAXN][MAXN];
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            cin >> graph[i][j];
        }
    }

    initialise(V, graph);
    floydWarshall(V);

    int u = 0; // Top-left corner
    int v = V - 1; // Bottom-right corner

    int path[MAXN];
    path[0] = u;
    int index = 1;
    while (u != v) {
        u = Next[u][v];
        path[index++] = u;
    }

    printPath(path, index);

    return 0;
}
```

Problem Statement

Varsha is working on optimizing the transportation network in a city, and she needs a program to help her find the shortest path between two locations efficiently.

She decided to implement the Floyd-Warshall algorithm to compute the shortest paths between all pairs of locations in the city.

Write a program that takes a weighted directed graph as input, applies the Floyd-Warshall algorithm to compute the shortest paths between all pairs of locations, and then finds and prints the shortest path from a specified source location to a destination location.

Input format :

The first line of input consists of a single integer, **V**, which is the number of vertices in the graph.

The next **V** lines, each containing **V** integers, represent the adjacency matrix of the weighted directed graph. The value of graph [i][j] represents the weight of the edge from vertex i to vertex j.

The last line of input consists of an integer, **destination**, representing the target vertex.

Output format :

The output displays the shortest path from the source vertex to the destination vertex in the following format: "Shortest path from source to destination S: P1 -> P2 -> P3 ->... -> PN".

where S is the destination index, and P1, P2, P3, ... PN represent the nodes in the path from source to destination.

Code constraints :

$1 \leq V \leq 100$

$0 \leq graph[i][j] \leq 10^7$

$0 \leq destination < V$

Sample test cases :

Input 1 :

```
3
0 1 2
1 0 3
2 3 0
2
```

Output 1 :

```
Shortest path from Source to destination 2: 0 -> 2
```

Input 2 :

```
4
0 5 10000000 10
10000000 0 3 10000000
10000000 10000000 0 1
10000000 10000000 10000000 0
3
```

Output 2 :

```
Shortest path from Source to destination 3: 0 -> 1 -> 2 -> 3
```

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

#define INF 1e7
#define MAXN 100

int dis[MAXN][MAXN];
int Next[MAXN][MAXN];

void initialise(int V, int graph[MAXN][MAXN]) {
   for (int i = 0; i < V; ++i) {
      for (int j = 0; j < V; ++j) {
         dis[i][j] = graph[i][j];
         if (graph[i][j] == INF)
            Next[i][j] = -1;
         else
            Next[i][j] = j;
      }
   }
}

void floydWarshall(int V) {
   for (int k = 0; k < V; ++k) {
      for (int i = 0; i < V; ++i) {
         for (int j = 0; j < V; ++j) {
            if (dis[i][k] == INF || dis[k][j] == INF)
               continue;

            if (dis[i][j] > dis[i][k] + dis[k][j]) {
               dis[i][j] = dis[i][k] + dis[k][j];
               Next[i][j] = Next[i][k];
            }
         }
      }
   }
}
```

```
void printPath(int path[], int n) {
    for (int i = 0; i < n - 1; ++i)
        cout << path[i] << " -> ";
    cout << path[n - 1];
}

int main() {
    int V;
    cin >> V;

    int graph[MAXN][MAXN];
    for (int i = 0; i < V; ++i) {
        for (int j = 0; j < V; ++j) {
            cin >> graph[i][j];
        }
    }

    initialise(V, graph);
    floydWarshall(V);

    int source = 0;

    int dest;
    cin >> dest;

    int path[MAXN];
    path[0] = source;
    int index = 1;
    while (source != dest) {
        source = Next[source][dest];
        path[index++] = source;
    }
    cout << "Shortest path from Source to destination " << dest << ": ";
    printPath(path, index);

    return 0;
}
```

--unit 3 completed—

Problem Statement

Ram is working on a project that requires implementing a string hashing function. He needs to create a program that can calculate a hash value for a given string and a specified table size using the provided hashing function. Help him design this program.

Note:

1. Initialize 'hashVal' to 0.
2. For each character key[i] in the string:
3. Multiply the current 'hashVal' by 37.
4. Add the ASCII value of the character key[i] to hashVal.
5. Finally, return 'hashVal % tSize', which ensures that the hash value falls within the range of the hash table size.

Input format :

The first line of input consists of a string, **text**.

The second line consists of an integer, **tSize**, representing the size of the hash table.

Output format :

The output displays a single integer, which is the hash value of the input string modulo tSize.

Code constraints :

The test cases will fall under the following constraints:

$1 \leq |text| \leq 10^2$

$1 \leq tSize \leq 10^9$

The text contains lowercase and uppercase letters without any space.

Sample test cases :

Input 1 :

```
Welcome
10000
```

Output 1 :

```
680
```

Input 2 :

```
Hello
10
```

Output 2 :

```
4
```

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

unsigned int hashCalc(string key, int tableSize) {
   unsigned int hashVal = 0;

   for (int i = 0; i < key.length(); i++)
      hashVal = 37 * hashVal + key[i];

   return hashVal % tableSize;
}

int main() {
   string text;
   int tSize;
   cin >> text;
   cin >> tSize;

   cout << hashCalc(text, tSize);
}
```

Problem Statement

Jessica is overseeing a supermarket and needs to create a custom hash function to store and organize the names of fruits in her inventory. She has already implemented a hash function and wants to validate its correctness using different fruit names.

Your task is to implement the hash function and help Jessica calculate the hash values for various fruit names.

Note

- hashValue = **(hashValue * 29) + ch**, where 'ch' represents the integer value of the character.
- **hashValue% HASH_TABLE_SIZE** is an operation involving the modulo operator (%).

Input format :

The input consists of strings of fruit names, separated by a line.

The input continues until Jessica enters 'exit' as a string, indicating the end of the input.

Output format :

For each input fruit name (excluding 'exit'), the program should output its corresponding hash value calculated using the hash function.

Each hash value should be printed on a new line.

Code constraints :

Hash table size = 10000

Length of the string = 100 characters

Each input string consists of lowercase and uppercase letters

Sample test cases :

Input 1 :

```
Apple
Banana
Strawberry
exit
```

Output 1 :

```
2258
9745
8553
```

Input 2 :

```
Dragonfruit
Guava
Mango
Papaya
Lychee
Persimmon
exit
```

Output 2 :

```
69
5560
1978
4928
3330
6978
```

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

const int HASH_TABLE_SIZE = 10000;

unsigned int customHash(string& key) {
    unsigned int hashValue = 0;

    for (char ch : key) {
        hashValue = (hashValue * 29) + ch;
    }

    return hashValue % HASH_TABLE_SIZE;
}

int main() {
    string key;
    while (true) {
        cin >> key;

        if (key == "exit") {
            break;
        }

        unsigned int hashValue = customHash(key);
        cout << hashValue << endl;
    }
    return 0;
}
```

Problem Statement

Madhev is working on a project where he needs to implement a hash table using the linear probing technique to store a collection of keys.

He wants to write a program that takes input for the size of the hash table and a list of keys and then uses linear probing to store the keys in the hash table.

After storing the keys, print the keys in ascending order based on their index in the hash table.

Example

Input
10
5
20 35 97 60 85
Output
index: 0, value: 20
index: 1, value: 60
index: 5, value: 35
index: 6, value: 85
index: 7, value: 97

Explanation

Each key is hashed using the modulo operation with the table size, and the initial index is determined. In this case,

- 20 is hashed to index 0.
- 35 is hashed to index 5.
- 97 is hashed to index 7.
- 60 is hashed to index 0 (collision, linear probing moves to the next index).
- 85 is hashed to index 5 (collision, linear probing moves to the next available index).

The keys are inserted into the hash table at the computed indices, resolving collisions using linear probing.
20 is inserted at index 0, 35 is inserted at index 5, 97 is inserted at index 7, 60 is inserted at index 1, and 85 is inserted at index 6.

Then print the minimum indices and their corresponding values in ascending order of the indices.
Input format :

The first line of input consists of an integer **size,** which represents the size of the hash table.
The second line of input consists of an integer **n,** the number of keys to be inserted.
The third line of input consists of n space-separated integers, representing the keys to be inserted into the hash table.

Output format :

The output displays, for each key, its index in the hash table and its value, in ascending order of the indices, in the following format:
"index: <<index>>, value: <<key>>"

Refer to the sample output for the formatting specifications.

Code constraints :

The test cases will fall under the following constraints:
$1 \leq size \leq 10$
$1 \leq n \leq 10$
$1 \leq keys \leq 100$

Sample test cases :

Input 1 :

```
10
5
20 35 97 60 85
```

Output 1 :

```
index: 0, value: 20
index: 1, value: 60
index: 5, value: 35
index: 6, value: 85
index: 7, value: 97
```

Input 2 :

```
5
4
29 13 17 15
```

Output 2 :

```
index: 0, value: 15
index: 2, value: 17
index: 3, value: 13
index: 4, value: 29
```

```
#include <stdio.h>

#define SIZE 100

int calHash(int key, int size)
{
   return key % size;
}

int main() {
   int size, numKeys;
   scanf("%d", &size);
   scanf("%d", &numKeys);

   int keys[numKeys];
   int hashTable[SIZE];
   for (int i = 0; i < SIZE; i++) {
      hashTable[i] = -1;
   }

   for (int i = 0; i < numKeys; ++i) {
      scanf("%d", &keys[i]);
   }

   int minIndices[numKeys];

   for (int i = 0; i < numKeys; ++i)
   {
      int key = keys[i];
      int index = calHash(key, size);
```

```
        while (hashTable[index] != -1) {
            index = (index + 1) % size;
        }

        hashTable[index] = key;

        int mIndex = index;
        while (keys[i] != hashTable[mIndex]) {
            mIndex = (mIndex + 1) % size;
        }

        minIndices[i] = mIndex;
    }

    for (int i = 0; i < numKeys; ++i) {
        for (int j = i + 1; j < numKeys; ++j) {
            if (minIndices[i] > minIndices[j]) {
                int temp = minIndices[i];
                minIndices[i] = minIndices[j];
                minIndices[j] = temp;
                temp = keys[i];
                keys[i] = keys[j];
                keys[j] = temp;
            }
        }

        printf("index: %d, value: %d\n", minIndices[i], keys[i]);
    }

    return 0;
}
```

Problem Statement

Deva works at a bustling supermarket, and he's looking for an efficient way to manage the products on the store shelves.

He wants to use a hash table with linear probing to keep track of the available slots on the shelves and optimize product placement. Help him write a program that manages the product slots using this approach.

Example

Input
6
5
15 30 9 21 12
Output
3 0 4 5 1

Explanation

In the product code allocation process, key 15 is hashed to index 3, resulting in "3" as it's stored in an empty slot. Similarly, key 30 is placed in index 0, yielding "0." Key 9 and 21 both map to index 3, invoking linear probing. Key 9 finds a slot at index 4, resulting in "4," while key 21 is stored at index 5, producing "5." Lastly, key 12 is hashed to index 0, leading to linear probing, and it's managed at index 1, printing "1."

Note

A hash function to determine the initial index for each key in a hash table, utilizing the modulo operation (key%size).
Input format :
The first line of input consists of an integer **N,** representing the size of the hash table.
The second line of input consists of an integer **K,** representing the number of product slots.
The third line of input consists of **K** integers, representing the product slots, separated by a space.
Output format :
The output displays product slots as space-separated integers, each representing the hash index of a product slot.

Refer to the sample output for the formatting specifications.
Code constraints :
The test cases will fall under the following constraints:
$1 \leq N \leq 10$
$1 \leq K \leq 100$
Sample test cases :
Input 1 :

```
6
5
15 30 9 21 12
```

Output 1 :

```
3 0 4 5 1
```

Input 2 :

```
10
5
123 456 789 321 654
```

Output 2 :

```
3 6 9 1 4
```

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

int SIZE = 100;

int calHash(int key, int size)
{
   return key % size;
}

int main() {
   int size, numKeys;
   cin >> size;
   cin >> numKeys;

   int keys[numKeys];
   int hashTable[SIZE];
   for (int i = 0; i < SIZE; i++) {
      hashTable[i] = -1;
   }

   for (int i = 0; i < numKeys; ++i) {
      cin >> keys[i];
   }

   for (int i = 0; i < numKeys; ++i) {
      int index = calHash(keys[i], size);
      while (hashTable[index] != -1) {
         index = (index + 1) % size;
      }
      hashTable[index] = keys[i];
      cout << index << ' ';
   }
   return 0;
}
```

--unit 4 completed—

Problem Statement

Roshene is a diligent student learning about hash tables and the quadratic probing method. She needs your help implementing a program that calculates her best mark using this method.

Write a program to help Roshene find and display this information using quadratic probing to handle collisions in a hash table. Calculate a hash index for each mark using the modulo operation (% tableSize).

Input format :

The first line of input consists of an integer **N,** representing the number of marks Roshene has received.

The second line consists of **N** integers, separated by spaces, representing the marks she received.

Output format :

The output displays the best mark and the index (0-based) in the hash table where the best mark is stored.

Refer to the sample output for the exact text and format.

Code constraints :

tableSize = 10

Sample test cases :

Input 1 :

```
5
98 97 92 78 94
```

Output 1 :

```
Best Mark: 98, Index: 8
```

Input 2 :

```
4
89 94 95 56
```

Output 2 :

```
Best Mark: 95, Index: 5
```

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

int main() {
   int tableSize = 10;
   int numKeys, i;
   cin >> numKeys;
   int keys[numKeys];

   for (i = 0; i < numKeys; i++) {
      cin >> keys[i];
   }

   int hashTable[tableSize];

   for (i = 0; i < tableSize; i++) {
      hashTable[i] = -1;
   }

   for (i = 0; i < numKeys; i++) {
      int key = keys[i];
      int hashIndex = key % tableSize;
      int j = 0;

      while (hashTable[hashIndex] != -1) {
         j++;
         hashIndex = (hashIndex + j * j) % tableSize;
      }

      hashTable[hashIndex] = key;
```

```
    }

    int min = hashTable[0];
    int in = 0;

    for (i = 1; i < tableSize; i++) {
        if (hashTable[i] != -1) {
            if (min < hashTable[i]) {
                min = hashTable[i];
                in = i;
            }
        }
    }

    cout << "Best Mark: " << min << ", Index: " << in ;

    return 0;
}
```

Problem Statement

Akil, a computer science student, is learning about hash tables and collision resolution techniques. He wants to implement a hash table with a size of 100 and use the mid-square hashing method to store a collection of keys. He's looking for a way to efficiently store these keys and resolve collisions using linear probing.

Example

Input

5

80 65 40 60 98

Output

Index 22: Key 65

Index 40: Key 80

Index 60: Key 40

Index 61: Key 60

Index 62: Key 98

Explanation

The mid-square hashing method is used to map each key to an index in a hash table of size 100.

Collision occurs when two keys map to the same index. so linear probing is used to find the next available index.

The final hash table stores the values as follows:

Index 22: Key 65

Index 40: Key 80

Index 60: Key 40 (collision resolved by linear probing)

Index 61: Key 60 (collision resolved by linear probing)

Index 62: Key 98 (collision resolved by linear probing)

The output displays the index and the corresponding key values in the hash table after handling collisions.

Note:

The middle digits are found by dividing the key's square by 10 and then taking the remainder when divided by 100, then returning modulo to the table size (TABLE_SIZE).

Input format :

The first line of input consists of an integer, 'n', representing the number of keys Akil wants to store in the hash table.

The second line of input consists of an integer 'key' representing the keys he wants to insert into the hash table.

Output format :

The output displays each line in the following format: **"Index x: Key y"** where 'x' is the index in the hash table and 'y' is the key.

Refer to the sample output for the formatting specifications.

Code constraints :

The test cases will fall under the following constraints:

hash table size = 100

1 <= n<= 10

10 <= key <= 100

Sample test cases :

Input 1 :

```
5
80 65 40 60 98
```

Output 1 :

```
Index 22: Key 65
Index 40: Key 80
Index 60: Key 40
Index 61: Key 60
Index 62: Key 98
```

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

#define TABLE_SIZE 100

int hashTable[TABLE_SIZE];

void initializeHashTable() {
    for (int i = 0; i < TABLE_SIZE; i++) {
        hashTable[i] = -1;
    }
}

int midSquareHash(int key) {
    int square = key * key;
    int middleDigits = (square / 10) % 100;
    return middleDigits % TABLE_SIZE;
}

int linearProbe(int index) {
    while (hashTable[index] != -1) {
        index = (index + 1) % TABLE_SIZE;
    }
    return index;
}

int main() {
```

```
    initializeHashTable();

    int n, key;
    cin >> n;

    for (int i = 0; i < n; i++) {
        cin >> key;

        int index = midSquareHash(key);

        if (hashTable[index] == -1) {
            hashTable[index] = key;
        } else {
            int newIndex = linearProbe(index);
            hashTable[newIndex] = key;
        }
    }

    for (int i = 0; i < TABLE_SIZE; i++) {
        if (hashTable[i] != -1) {
            cout << "Index " << i << ": Key " << hashTable[i] << endl;
        }
    }

    return 0;
}
```

Problem Statement

Rithika, an aspiring computer scientist, is diving into the world of hash functions and their practical applications. She recently learned about a hash function called "Mid-Square Hashing".

Rithika is implementing the Mid-Square Hashing algorithm to store integer keys in a hash table. She's using linear probing to resolve collisions by finding the next available slot.

Your task is to help her write the code and find the key with the highest value in the hash table, along with its index.

Example

Input
6
42 58 37 29 85 89
Output
Weight: 89, Index: 92

Explanation

- A hash table of size 100 is initialized with all entries set to -1, representing an empty array.
- The Mid-Square Hashing algorithm is used to calculate hash indices for keys.
- Key 42 squared to 1764, and middle digits (76) were used as its hash index.
- Key 58 resulted in a hash index of 36.
- Key 37 faced a collision and was stored at index 37 using linear probing.
- Key 29 had an index of 41.
- Key 85 also experienced a collision, which was resolved by placing it at index 42. The key with the maximum value, 89 (square value 92), was stored at index 92 after a linear search, demonstrating collision resolution for finding the maximum-weighted key.

Note: The formula (squared / 10) extracts the last two digits, and % 100 ensures middleDigit is within 0 to 99, limiting the hash index to the table size.

Input format :
The first line of input consists of an integer, n, representing the number of integer keys to be stored in the hash table.
The second line of input consists of n space-separated integers, where each integer key represents a unique key to be hashed and stored in the table.

Output format :
The output displays a single line with two space-separated integers in the following format:
"Weight: <<X>>, Index: <<Y>>".
where:
"X" represents the weight (value) of the key with the maximum weight in the hash table.
"Y" represents the index (position) in the hash table where the key with the maximum weight is stored.

Refer to the sample output for the formatting specifications.

Code constraints :
table size = 100
10 <= key <= 100

Sample test cases :

Input 1 :
```
5
60 65 70 75 80
```
Output 1 :
```
Weight: 80, Index: 40
```
Input 2 :
```
6
42 58 37 29 85 89
```
Output 2 :
```
Weight: 89, Index: 92
```

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

int midSquareHash(int key, int tableSize)
{
    int squared = key * key;
    int middleDigit = (squared / 10) % 100;
    int hashIndex = middleDigit % tableSize;
    return hashIndex;
}

int main() {
    int tableSize = 100, in, max;
    int numKeys;
    cin >> numKeys;
    int keys[numKeys];
    for (int i = 0; i < numKeys; i++) {
        cin >> keys[i];
    }
    int hashTable[tableSize];
    for (int i = 0; i < tableSize; i++) {
        hashTable[i] = -1;
    }
    for (int i = 0; i < numKeys; i++) {
        int key = keys[i];
        int hashIndex = midSquareHash(key, tableSize);

        while (hashTable[hashIndex] != -1) {
            hashIndex = (hashIndex + 1) % tableSize;
        }
        hashTable[hashIndex] = key;
    }

    max = hashTable[0];
    in = 0;
    for (int i = 1; i < tableSize; i++) {
        if (hashTable[i] != -1 && hashTable[i] > max) {
            max = hashTable[i];
            in = i;
        }
    }
    cout << "Weight: " << max << ", Index: " << in;
    return 0;
}
```

Problem Statement

Reema is organizing a special event and needs to assign guests to specific seats in her venue. She has a table with 10 seats and a list of ticket IDs for the guests.

Reema is looking to create a program that takes user input, including the number of ticket IDs (numKeys) and the list of ticket IDs (keys).

This program uses a hash table (hashTable) and a basic modulo operation as the hash function to assign each guest to a seat based on their ticket ID.

If a seat is already occupied, the program will find the next available seat to ensure that each guest has a unique seating assignment.

Input format :

The first line of input consists of an integer, n, representing the number of ticket IDs.
The second line of input consists of n space-separated integers, each integer representing a unique ticket ID.

Output format :

The output displays the seat allocation for each guest in the following format:

For each occupied seat, it will print **"Seat Number X: Ticket ID Y"**

where X is the seat number and Y is the ticket ID assigned to that seat.

Refer to the sample output for the formatting specifications.

Code constraints :

hash table size = 10
$1 \leq n \leq 10$
$1 \leq ticketID \leq 1000$
$0 \leq X <$ tableSize

Sample test cases :

Input 1 :

```
6
342 190 974 120 415 870
```

Output 1 :

```
Seat Number 0: Ticket ID 190
Seat Number 1: Ticket ID 120
Seat Number 2: Ticket ID 342
Seat Number 3: Ticket ID 870
Seat Number 4: Ticket ID 974
Seat Number 5: Ticket ID 415
```

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

int main() {
    int tableSize = 10;
    int numKeys;
    cin >> numKeys;

    int keys[numKeys];
    for (int i = 0; i < numKeys; i++) {
        cin >> keys[i];
    }

    int hashTable[tableSize];
    for (int i = 0; i < tableSize; i++) {
        hashTable[i] = -1;
    }

    for (int i = 0; i < numKeys; i++) {
        int key = keys[i];
        int hashIndex = key % tableSize;

        while (hashTable[hashIndex] != -1) {
            hashIndex = (hashIndex + 1) % tableSize;
        }

        hashTable[hashIndex] = key;
    }

    for (int i = 0; i < tableSize; i++) {
        if (hashTable[i] != -1) {
            cout << "Seat Number " << i << ": Ticket ID " << hashTable[i] << endl;
        }
    }

    return 0;
}
```

Problem Statement

You are planning a trip with your family to Goa. So you have booked the flight tickets for your family members. The air travel company follows the quadratic probing for making the seating arrangements for the passengers by considering their age.

Assume the hashtable size is 10. Write the code to view the seat numbers (index) for senior citizens, i.e., age ≥ 60. For each senior citizen's age, calculate a hash index using the modulo operation (% tableSize).

Input format :

The first line of input consists of an integer **N,** representing the number of family members.

The second line consists of **N** space-separated integers, representing the age of the family members.

Output format :

The output prints the index of the members with age ≥ 60 (index starts from 0)

If no senior citizens are found, print "No senior citizens in the family".

Refer to the sample output for the exact text and format.

Code constraints :

tableSize = 10

Sample test cases :

Input 1 :

```
5
23 40 45 12 65
```

Output 1 :

```
Age: 65, Seat: 6
```

Input 2 :

```
5
23 40 45 12 25
```

Output 2 :

```
No senior citizens in the family
```

Input 3 :

```
6
24 26 59 60 78 81
```

Output 3 :

```
Age: 60, Seat: 0
Age: 81, Seat: 1
Age: 78, Seat: 8
```

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

int main() {
   int tableSize = 10;
   int numKeys;
   cin >> numKeys;
   int keys[numKeys];
   for (int i = 0; i < numKeys; i++) {
      cin >> keys[i];
   }
   int hashTable[tableSize];
   for (int i = 0; i < tableSize; i++) {
      hashTable[i] = -1;
   }
   for (int i = 0; i < numKeys; i++) {
      int key = keys[i];
      int hashIndex = key % tableSize;
      int j = 0;
      while (hashTable[hashIndex] != -1) {
         j++;
         hashIndex = (hashIndex + j * j) % tableSize;
      }
      hashTable[hashIndex] = key;
   }
   int f = 0;
   for (int i = 0; i < tableSize; i++) {
      if (hashTable[i] != -1 && hashTable[i] >= 60) {
         cout << "Age: " << hashTable[i] <<", Seat: " << i <<"\n";
         f = 1;
      }
   }
   if (f == 0) {
      cout << "No senior citizens in the family";
   }
   return 0;
}
```

Problem Statement

Rishi is working on a program that manages a hash table to store a set of integer keys. He wants to implement a hash table of fixed size (10) and has asked for your help to develop the code.

Your task is to assist him in implementing this functionality with collision resolution and provide the first and last key values, along with their respective indices, in the hash table.

Input format :

The first line of input consists of an integer, N, denoting the number of keys to be inserted into the hash table.

The second line of input consists of a space-separated integer, representing the keys to be inserted.

Output format :

The output displays two lines, each containing the index and value of the first and last keys in the hash table, in the following format:

- **"First index: <index>, Value: <value>"**
- **"Last index: <index>, Value: <value>"**

Refer to the sample output for the formatting specifications.

Code constraints :

table size = 10

1 <= n <= 10

1<= keys <= 10^6

Sample test cases :

Input 1 :

```
6
1024 1056 2045 3145 1210 3512
```

Output 1 :

```
First index: 0, Value: 1210
Last index: 7, Value: 3145
```

Input 2 :

```
6
1230 1123 1450 1256 1425 1520
```

Output 2 :

```
First index: 0, Value: 1230
Last index: 6, Value: 1256
```

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

int main() {
    int tableSize = 10;
    int numKeys;
    cin >> numKeys;

    int keys[numKeys];
    for (int i = 0; i < numKeys; i++) {
        cin >> keys[i];
    }

    int hashTable[tableSize];
    for (int i = 0; i < tableSize; i++) {
        hashTable[i] = -1;
    }

    for (int i = 0; i < numKeys; i++) {
        int key = keys[i];
        int hashIndex = key % tableSize;

        while (hashTable[hashIndex] != -1) {
            hashIndex = (hashIndex + 1) % tableSize;
        }

        hashTable[hashIndex] = key;
```

```
    }

    int f = -1, l = -1;
    int f_index = -1, l_index = -1;

    for (int i = 0; i < tableSize; i++) {
        if (hashTable[i] != -1) {
            if (f == -1) {
                f = hashTable[i];
                f_index = i;
            }
            l = hashTable[i];
            l_index = i;
        }
    }

    if (f != -1 && l != -1) {
        cout << "First index: " << f_index << ", Value: " << f << endl;
        cout << "Last index: " << l_index << ", Value: " << l << endl;
    }

    return 0;
}
```

--unit 5 completed--