```cpp
//iterative implementation of BFSs

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

// Data structure to store a graph edge
struct Edge {
    int src, dest;
};

// A class to represent a graph object
class Graph
{
public:
    // a vector of vectors to represent an adjacency list
    vector<vector<int>> adjList;

    // Graph Constructor
    Graph(vector<Edge> const &edges, int n)
    {
        // resize the vector to hold `n` elements of type `vector<int>`
        adjList.resize(n);

        // add edges to the undirected graph
        for (auto &edge: edges)
        {
            adjList[edge.src].push_back(edge.dest);
            adjList[edge.dest].push_back(edge.src);
        }
    }
};

// Perform BFS on the graph starting from vertex `v`
void BFS(Graph const &graph, int v, vector<bool> &discovered)
{
    // create a queue for doing BFS
    queue<int> q;

    // mark the source vertex as discovered
    discovered[v] = true;

    // enqueue source vertex
    q.push(v);

    // loop till queue is empty
    while (!q.empty())
    {
        // dequeue front node and print it
        v = q.front();
        q.pop();
```

```
        cout << v << " ";

        // do for every edge (v, u)
        for (int u: graph.adjList[v])
        {
            if (!discovered[u])
            {
                // mark it as discovered and enqueue it
                discovered[u] = true;
                q.push(u);
            }
        }
    }
}

int main()
{
    // vector of graph edges as per the above diagram
    vector<Edge> edges = {
        {1, 2}, {1, 3}, {1, 4}, {2, 5}, {2, 6}, {5, 9},
        {5, 10}, {4, 7}, {4, 8}, {7, 11}, {7, 12}
        // vertex 0, 13, and 14 are single nodes
    };

    // total number of nodes in the graph (labelled from 0 to 14)
    int n = 15;

    // build a graph from the given edges
    Graph graph(edges, n);

    // to keep track of whether a vertex is discovered or not
    vector<bool> discovered(n, false);

    // Perform BFS traversal from all undiscovered nodes to
    // cover all connected components of a graph
    for (int i = 0; i < n; i++)
    {
        if (discovered[i] == false)
        {
            // start BFS traversal from vertex `i`
            BFS(graph, i, discovered);
        }
    }

    return 0;
}
```

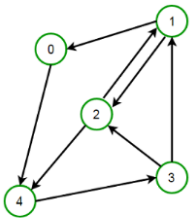
//ADJACENCY MATRIX PRINT

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

// Data structure to store a graph edge
struct Edge {
    int src, dest;
};

// A class to represent a graph object
class Graph
{
public:
    // a vector of vectors to represent an adjacency list
    vector<vector<int>> adjList;

    // Constructor
    Graph(vector<Edge> const &edges, int n)
    {
        adjList.resize(n);

        // add edges to the directed graph
        for (Edge edge: edges)
        {
            int src = edge.src;
            int dest = edge.dest;

            adjList[src].push_back(dest);
        }
    }
};

// `C` is a connectivity matrix and stores transitive closure of a graph
// `root` is the topmost node in DFS tree (it is the starting vertex of DFS)
// `descendant` is current vertex to be explored in DFS.
// Invariant: A path already exists in the graph from `root` to `descendant`
void DFS(Graph const &graph, vector<vector<bool>> &C, int root, int descendant)
{
    for (int child: graph.adjList[descendant])
    {
        // if `child` is an adjacent vertex of descendant, we have
        // found a path from root->child
        if (!C[root][child])
        {
            C[root][child] = true;
            DFS(graph, C, root, child);
        }
    }
```

```
}

int main()
{
    // an array of graph edges as per the above diagram
    vector<Edge> edges = {
        {0, 2}, {1, 0}, {3, 1}
    };

    // total number of nodes in the graph (labelled from 0 to 3)
    int n = 4;

    // build a graph from the given edges
    Graph graph(edges, n);

    // `C` is a connectivity matrix and stores the transitive closure
    // of the graph. The value of `C[i][j]` is 1 only if a directed
    // path exists from vertex `i` to vertex `j`.
    vector<vector<bool>> C(n, vector<bool>(n, false));

    // consider each vertex and start DFS from it
    for (int v = 0; v < n; v++)
    {
        C[v][v] = true;
        DFS(graph, C, v, v);

        // print path info for vertex `v`
        for (int u = 0; u < n; u++) {
            cout << left << setw(4) << C[v][u];
        }
        cout << endl;
    }

    return 0;
}
```

Given a directed graph, check if it is strongly connected or not. A directed graph is said to be strongly connected if every vertex is reachable from every other vertex.

For example, the following graph is strongly connected as a path exists between all pairs of vertices:



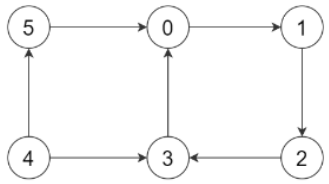

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

// Data structure to store a graph edge
struct Edge {
    int src, dest;
};

// A class to represent a graph object
class Graph
{
public:

    // a vector of vectors to represent an adjacency list
    vector<vector<int>> adjList;

    // Graph Constructor
    Graph(vector<Edge> const &edges, int n)
    {
        // resize the vector to hold `n` elements of type `vector<int>`
        adjList.resize(n);

        // add edges to the directed graph
        for (auto &edge: edges) {
            adjList[edge.src].push_back(edge.dest);
        }
    }
};

// Function to perform DFS traversal on the graph on a graph
void DFS(Graph const &graph, int v, vector<bool> &visited)
{
    // mark current node as visited
    visited[v] = true;

    // do for every edge (v, u)
    for (int u: graph.adjList[v])
```

```
    {
        // `u` is not visited
        if (!visited[u]) {
            DFS(graph, u, visited);
        }
    }
}

// Function to check if the graph is strongly connected or not
bool isStronglyConnected(Graph const &graph, int n)
{
    // do for every vertex
    for (int i = 0; i < n; i++)
    {
        // to keep track of whether a vertex is visited or not
        vector<bool> visited(n);

        // start DFS from the first vertex
        DFS(graph, i, visited);

        // If DFS traversal doesn't visit all vertices,
        // then the graph is not strongly connected
        if (find(visited.begin(), visited.end(), false) != visited.end()) {
            return false;
        }
    }

    return true;
}

int main()
{
    // vector of graph edges as per the above diagram
    vector<Edge> edges = {
        {0, 4}, {1, 0}, {1, 2}, {2, 1}, {2, 4},
        {3, 1}, {3, 2}, {4, 3}
    };

    // total number of nodes in the graph
    int n = 5;
    // build a graph from the given edges
    Graph graph(edges, n);
    // check if the graph is not strongly connected or not
    if (isStronglyConnected(graph, n)) {
        cout << "The graph is strongly connected";
    }
    else {
        cout << "The graph is not strongly connected";
    }

    return 0;
}
```

//finding root vertex of a graph using DFS

A graph can have multiple root vertices. For example, each vertex in a **strongly connected component** is a root vertex. In such cases, the solution should return anyone of them. If the graph has no root vertices, the solution should return `-1`.

The root vertex is `4` since it has a path to every other vertex in the following graph:



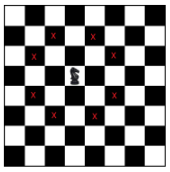

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

// Data structure to store a graph edge
struct Edge {
    int src, dest;
};

// A class to represent a graph object
class Graph
{
public:

    // a vector of vectors to represent an adjacency list
    vector<vector<int>> adjList;

    // Graph Constructor
    Graph(vector<Edge> const &edges, int n)
    {
        // resize the vector to hold `n` elements
        adjList.resize(n);

        // add edges to the directed graph
        for (auto &edge: edges) {
            adjList[edge.src].push_back(edge.dest);
        }
    }
};
  // Utility function to perform DFS traversal on the graph on a graph
void DFS(Graph const &graph, int u, vector<bool> &visited)
{
    // mark the current node as visited
    visited[u] = true;

    // do for every edge (u, v)
    for (int v: graph.adjList[u])
```

```
    {
        // if `v` is not visited
        if (!visited[v]) {
            DFS(graph, v, visited);
        }
    }
}

// Function to find the root vertex of a graph
int findRootVertex(Graph const &graph, int n)
{
    // to keep track of all previously visited vertices in DFS
    vector<bool> visited(n);

    // find the last starting vertex `v` in DFS
    int v = 0;
    for (int i = 0; i < n; i++)
    {
        if (!visited[i])
        {
            DFS(graph, i, visited);
            v = i;
        }
    }

    // reset the visited vertices
    fill(visited.begin(), visited.end(), false);

    // perform DFS on the graph from the last starting vertex `v`
    DFS(graph, v, visited);

    // return -1 if all vertices are not reachable from vertex `v`
    for (int i = 0; i < n; i++)
    {
        if (!visited[i]) {
            return -1;
        }
    }

    // we reach here only if `v` is a root vertex
    return v;
}

int main()
{
    // vector of graph edges as per the above diagram
    vector<Edge> edges = {
        {0, 1}, {1, 2}, {2, 3}, {3, 0}, {4, 3}, {4, 5}, {5, 0}
    };

    // total number of nodes in the graph (0 to 5)
    int n = 6;
```

```
    // build a directed graph from the given edges
    Graph graph(edges, n);

    // find the root vertex in the graph
    int root = findRootVertex(graph, n);

    if (root != -1) {
        cout << "The root vertex is " << root << endl;
    }
    else {
        cout << "The root vertex does not exist" << endl;
    }

    return 0;
}
```

//chess night problem BFS

A knight can move in eight possible directions from a given cell, as illustrated in the following figure:



We can find all the possible locations the knight can move to from the given location by using the array that stores the relative position of knight movement from any location. For example, if the current location is `(x, y)`, we can move to `(x + row[k], y + col[k])` for `0 <= k <= 7` using the following array:

```
row[] = { 2, 2, -2, -2, 1, 1, -1, -1 }
col[] = { -1, 1, 1, -1, 2, -2, 2, -2 }
```

So, from position `(x, y)` knight's can move to:

```
(x + 2, y - 1)
(x + 2, y + 1)
(x - 2, y + 1)
(x - 2, y - 1)
(x + 1, y + 2)
(x + 1, y - 2)
(x - 1, y + 2)
(x - 1, y - 2)
```

```cpp
#include <iostream>
#include <set>
#include <queue>
#include <climits>
using namespace std;

// Below arrays detail all eight possible movements
// for a knight
int row[] = { 2, 2, -2, -2, 1, 1, -1, -1 };
int col[] = { -1, 1, 1, -1, 2, -2, 2, -2 };

// Check if (x, y) is valid chessboard coordinates.
// Note that a knight cannot go out of the chessboard
bool isValid(int x, int y, int N) {
   return (x >= 0 && x < N) && (y >= 0 && y < N);
}

// A queue node used in BFS
struct Node
{
   // (x, y) represents chessboard coordinates
   // `dist` represents its minimum distance from the source
   int x, y, dist;

   // Node constructor
   Node(int x, int y, int dist = 0): x(x), y(y), dist(dist) {}

   // As we are using struct as a key in a `std::set`,
   // we need to overload `<` operator.
   // Alternatively, we can use `std::pair<int, int>` as a key
   // to store the matrix coordinates in the set.

   bool operator<(const Node& o) const {
```

```
        return x < o.x || (x == o.x && y < o.y);
    }
};

// Find the minimum number of steps taken by the knight
// from the source to reach the destination using BFS
int findShortestDistance(int N, Node src, Node dest)
{
    // set to check if the matrix cell is visited before or not
    set<Node> visited;

    // create a queue and enqueue the first node
    queue<Node> q;
    q.push(src);

    // loop till queue is empty
    while (!q.empty())
    {
        // dequeue front node and process it
        Node node = q.front();
        q.pop();

        int x = node.x;
        int y = node.y;
        int dist = node.dist;

        // if the destination is reached, return distance
        if (x == dest.x && y == dest.y) {
            return dist;
        }

        // skip if the location is visited before
        if (!visited.count(node))
        {
            // mark the current node as visited
            visited.insert(node);

            // check for all eight possible movements for a knight
            // and enqueue each valid movement
            for (int i = 0; i < 8; i++)
            {
                // get the knight's valid position from the current position on
                // the chessboard and enqueue it with +1 distance
                int x1 = x + row[i];
                int y1 = y + col[i];

                if (isValid(x1, y1, N)) {
                    q.push({x1, y1, dist + 1});
                }
            }
```

```
      }
    }

    // return infinity if the path is not possible
    return INT_MAX;
}

int main()
{
    // N x N matrix
    int N = 8;

    // source coordinates
    Node src = {0, 7};

    // destination coordinates
    Node dest = {7, 0};

    cout << "The minimum number of steps required is " <<
          findShortestDistance(N, src, dest);

    return 0;
}
```

Shortest path in a maze – Lee Algorithm

Given a **maze** in the form of the binary rectangular matrix, find the shortest path's length in a maze from a given source to a given destination.

The path can only be constructed out of cells having value 1, and at any given moment, we can only move one step in one of the four directions. The valid moves are:

```
Go Top:  (x, y) ⟶ (x - 1, y)
Go Left: (x, y) ⟶ (x, y - 1)
Go Down: (x, y) ⟶ (x + 1, y)
Go Right: (x, y) ⟶ (x, y + 1)
```

For example, consider the following binary matrix. If `source = (0, 0)` and `destination = (7, 5)`, the shortest path from source to destination has length 12.

```
[ 1 1 1 1 1 0 0 1 1 1 ]
[ 0 1 1 1 1 1 0 1 0 1 ]
[ 0 0 1 0 1 1 1 0 0 1 ]
[ 1 0 1 1 1 0 1 1 0 1 ]
[ 0 0 0 1 0 0 0 1 0 1 ]
[ 1 0 1 1 1 0 0 1 1 0 ]
[ 0 0 0 0 1 0 0 1 0 1 ]
[ 0 1 1 1 1 1 1 1 0 0 ]
[ 1 1 1 1 1 0 0 1 1 1 ]
[ 0 0 1 0 0 1 1 0 0 1 ]
```

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

// A Queue Node
struct Node
{
   // (x, y) represents matrix cell coordinates, and
   // `dist` represents their minimum distance from the source
   int x, y, dist;
};

// Below arrays detail all four possible movements from a cell
int row[] = { -1, 0, 0, 1 };
int col[] = { 0, -1, 1, 0 };

// Function to check if it is possible to go to position (row, col)
// from the current position. The function returns false if (row, col)
// is not a valid position or has a value 0 or already visited.
bool isValid(vector<vector<int>> const &mat, vector<vector<bool>> &visited,
      int row, int col) {
   return (row >= 0 && row < mat.size()) && (col >= 0 && col < mat[0].size())
      && mat[row][col] && !visited[row][col];
}

// Find the shortest possible route in a matrix `mat` from source
// cell (i, j) to destination cell (x, y)
int findShortestPathLength(vector<vector<int>> const &mat, pair<int, int> &src,
                pair<int, int> &dest)
{
   // base case: invalid input
```

```
if (mat.size() == 0 || mat[src.first][src.second] == 0 ||
      mat[dest.first][dest.second] == 0) {
   return -1;
}

// `M × N` matrix
int M = mat.size();
int N = mat[0].size();

// construct a `M × N` matrix to keep track of visited cells
vector<vector<bool>> visited;
visited.resize(M, vector<bool>(N));

// create an empty queue
queue<Node> q;

// get source cell (i, j)
int i = src.first;
int j = src.second;

// mark the source cell as visited and enqueue the source node
visited[i][j] = true;
q.push({i, j, 0});

// stores length of the longest path from source to destination
int min_dist = INT_MAX;

// loop till queue is empty
while (!q.empty())
{
   // dequeue front node and process it
   Node node = q.front();
   q.pop();

   // (i, j) represents a current cell, and `dist` stores its
   // minimum distance from the source
   int i = node.x, j = node.y, dist = node.dist;

   // if the destination is found, update `min_dist` and stop
   if (i == dest.first && j == dest.second)
   {
      min_dist = dist;
      break;
   }

   // check for all four possible movements from the current cell
   // and enqueue each valid movement
   for (int k = 0; k < 4; k++)
   {
      // check if it is possible to go to position
      // (i + row[k], j + col[k]) from current position
      if (isValid(mat, visited, i + row[k], j + col[k]))
```

```
            {
                // mark next cell as visited and enqueue it
                visited[i + row[k]][j + col[k]] = true;
                q.push({ i + row[k], j + col[k], dist + 1 });
            }
        }
    }

    if (min_dist != INT_MAX) {
        return min_dist;
    }

    return -1;
}

int main()
{
    vector<vector<int>> mat =
    {
        { 1, 1, 1, 1, 1, 0, 0, 1, 1, 1 },
        { 0, 1, 1, 1, 1, 1, 0, 1, 0, 1 },
        { 0, 0, 1, 0, 1, 1, 1, 0, 0, 1 },
        { 1, 0, 1, 1, 1, 0, 1, 1, 0, 1 },
        { 0, 0, 0, 1, 0, 0, 0, 1, 0, 1 },
        { 1, 0, 1, 1, 1, 0, 0, 1, 1, 0 },
        { 0, 0, 0, 0, 1, 0, 0, 1, 0, 1 },
        { 0, 1, 1, 1, 1, 1, 1, 1, 0, 0 },
        { 1, 1, 1, 1, 1, 0, 0, 1, 1, 1 },
        { 0, 0, 1, 0, 0, 1, 1, 0, 0, 1 },
    };

    pair<int, int> src = make_pair(0, 0);
    pair<int, int> dest = make_pair(7, 5);

    int min_dist = findShortestPathLength(mat, src, dest);
    if (min_dist != -1)
    {
        cout << "The shortest path from source to destination "
             "has length " << min_dist;
    }
    else {
        cout << "Destination cannot be reached from a given source";
    }

    return 0;
}
//op: 12
```

Find the shortest safe route in a field with sensors present

Given a rectangular field with few sensors present, cross it by taking the shortest safe route without activating the sensors.

The rectangular field is in the form of an $M \times N$ matrix, and we need to find the shortest path from any cell in the first column to any cell in the last column of the matrix. The sensors are marked by the value 0 in the matrix, and all its eight adjacent cells can also activate the sensor. The path can only be constructed out of cells having value 1, and at any given moment, we can only move one step in one of the four directions. The valid moves are:

```
Go Up:    (x, y) ⟶ (x - 1, y)
Go Left:  (x, y) ⟶ (x, y - 1)
Go Down:  (x, y) ⟶ (x + 1, y)
Go Right: (x, y) ⟶ (x, y + 1)
```

For example, consider the following matrix:

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

The shortest safe path has a length of 11, and the route is marked in green.

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

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

// A queue node used in BFS
struct Node {
   // (x, y) represents a position inside the field
   // `dist` represents its minimum distance from the source
   int x, y, dist;
};

// Below arrays detail all four possible movements from a cell,
// i.e., (top, right, bottom, left)
int row[] = { -1, 0, 0, 1 };
int col[] = { 0, -1, 1, 0 };

// Function to check if it is safe to go to position (x, y)
// from the current position. The function returns false if (x, y)
// is unsafe or already visited.
bool isSafe(vector<vector<int>> const &field, vector<vector<bool>> const &visited,
            int x, int y) {
   return field[x][y] != 0 && !visited[x][y];
}

// Check if (x, y) is valid field coordinates.
// Note that we cannot go out of the field.
bool isValid(int x, int y, int M, int N) {
```

```
    return (x < M && y < N) && (x >= 0 && y >= 0);
}

// Find the minimum number of steps required to reach the last column
// from the first column using BFS
int BFS(vector<vector<int>> const &field)
{
    // `M × N` matrix
    int M = field.size();
    int N = field[0].size();

    // stores if a cell is visited or not
    vector<vector<bool>> visited;
    visited.resize(M, vector<bool>(N));

    // create an empty queue
    queue<Node> q;

    // process every cell of the first column
    for (int r = 0; r < M; r++)
    {
        // if the cell is safe, mark it as visited and
        // enqueue it by assigning it distance as 0
        if (field[r][0] == 1)
        {
            q.push({r, 0, 0});
            visited[r][0] = true;
        }
    }

    // loop till queue is empty
    while (!q.empty())
    {
        // dequeue front node and process it
        int i = q.front().x;
        int j = q.front().y;
        int dist = q.front().dist;
        q.pop();

        // if the destination is found, return minimum distance
        if (j == N - 1) {
            return dist;
        }

        // check for all four possible movements from the current cell
        // and enqueue each valid movement
        for (int k = 0; k < 4; k++)
        {
            // skip if the location is invalid or visited, or unsafe
            if (isValid(i + row[k], j + col[k], M, N) &&
                isSafe(field, visited, i + row[k], j + col[k]))
            {
```

```
            // mark it as visited and enqueue it with +1 distance
            visited[i + row[k]][j + col[k]] = true;
            q.push({i + row[k], j + col[k], dist + 1});
        }
    }
  }

  return INT_MAX;
}

// Find the shortest path from the first column to the last column in a given field
int findShortestDistance(vector<vector<int>> &mat)
{
  // base case
  if (mat.size() == 0) {
    return 0;
  }

  // `M × N` matrix
  int M = mat.size();
  int N = mat[0].size();

  // `r[]` and `c[]` detail all eight possible movements from a cell
  // (top, right, bottom, left, and four diagonal moves)
  int r[] = { -1, -1, -1, 0, 0, 1, 1, 1 };
  int c[] = { -1, 0, 1, -1, 1, -1, 0, 1 };

  // mark adjacent cells of sensors as unsafe
  for (int i = 0; i < M; i++)
  {
    for (int j = 0; j < N; j++)
    {
      for (int k = 0; k < 8; k++) {
        if (!mat[i][j] && isValid(i + r[k], j + c[k], M, N)
            && mat[i + r[k]][j + c[k]]) {
          mat[i + r[k]][j + c[k]] = INT_MAX;
        }
      }
    }
  }

  // update the mat
  for (int i = 0; i < M; i++)
  {
    for (int j = 0; j < N; j++)
    {
      if (mat[i][j] == INT_MAX) {
        mat[i][j] = 0;
      }
    }
  }
```

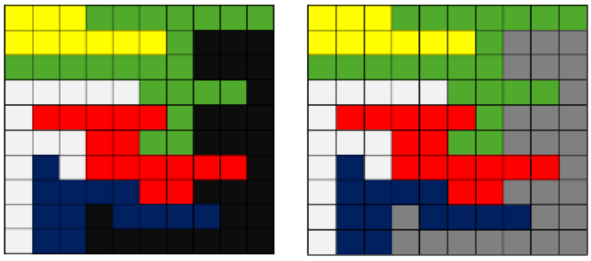
```
    // call BFS and return the shortest distance found by it
    return BFS(mat);
}

int main()
{
    vector<vector<int>> field =
    {
        { 0, 1, 1, 1, 0, 1, 1, 1, 1, 1 },
        { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 },
        { 1, 1, 1, 1, 1, 1, 1, 1, 0, 1 },
        { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 },
        { 1, 1, 1, 1, 1, 0, 1, 1, 1, 1 },
        { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 },
        { 1, 0, 1, 1, 1, 1, 1, 1, 1, 1 },
        { 1, 1, 1, 1, 1, 1, 1, 1, 1, 0 },
        { 1, 1, 1, 1, 1, 0, 1, 1, 1, 1 },
        { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 }
    };

    int dist = findShortestDistance(field);

    if (dist != INT_MAX) {
        cout << "The shortest safe path has a length of " << dist;
    }
    else {
        cout << "No route is safe to reach destination";
    }

    return 0;
}
```

Flood fill (also known as seed fill) is an algorithm that determines the area connected to a given node in a multi-dimensional array.

It is used in the "bucket" fill tool of a paint program to fill connected, similarly colored areas with a different color and in games such as Go and Minesweeper for determining which pieces are cleared. When applied on an image to fill a particular bounded area with color, it is also known as boundary fill.

The flood fill algorithm takes three parameters: a start node, a target color, and a replacement color.

Consider the following matrix to the left – if the start node is `(3, 9)`, target color is **"BLACK"** and replacement color is **"GREY"**, the algorithm looks for all nodes in the matrix that are connected to the start node by a path of the target color and changes them to the replacement color.



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

// Below arrays detail all eight possible movements
int row[] = { -1, -1, -1, 0, 0, 1, 1, 1 };
int col[] = { -1, 0, 1, -1, 1, -1, 0, 1 };

// check if it is possible to go to pixel (x, y) from the
// current pixel. The function returns false if the pixel
// has a different color, or it's not a valid pixel
bool isSafe(vector<vector<char>> const &mat, int x, int y, char target)
{
   return (x >= 0 && x < mat.size()) && (y >= 0 && y < mat[0].size())
      && mat[x][y] == target;
}

// Flood fill using BFS
void floodfill(vector<vector<char>> &mat, int x, int y, char replacement)
{
   // base case
   if (mat.size() == 0) {
      return;
   }
```

```
    // create a queue and enqueue starting pixel
    queue<pair<int, int>> q;
    q.push({x, y});

    // get the target color
    char target = mat[x][y];

    // target color is same as replacement
    if (target == replacement) {
        return;
    }

    // break when the queue becomes empty
    while (!q.empty())
    {
        // dequeue front node and process it
        pair<int, int> node = q.front();
        q.pop();

        // (x, y) represents the current pixel
        int x = node.first, y = node.second;

        // replace the current pixel color with that of replacement
        mat[x][y] = replacement;

        // process all eight adjacent pixels of the current pixel and
        // enqueue each valid pixel
        for (int k = 0; k < 8; k++)
        {
            // if the adjacent pixel at position (x + row[k], y + col[k]) is
            // is valid and has the same color as the current pixel
            if (isSafe(mat, x + row[k], y + col[k], target))
            {
                // enqueue adjacent pixel
                q.push({x + row[k], y + col[k]});
            }
        }
    }
}

// Utility function to print a matrix
void printMatrix(vector<vector<char>> const &mat)
{
    for (int i = 0; i < mat.size(); i++)
    {
        for (int j = 0; j < mat[0].size(); j++) {
            cout << setw(3) << mat[i][j];
        }
        cout << endl;
```

```
    }
}

int main()
{
    // matrix showing portion of the screen having different colors
    vector<vector<char>> mat =
    {
        { 'Y', 'Y', 'Y', 'G', 'G', 'G', 'G', 'G', 'G', 'G' },
        { 'Y', 'Y', 'Y', 'Y', 'Y', 'Y', 'G', 'X', 'X', 'X' },
        { 'G', 'G', 'G', 'G', 'G', 'G', 'G', 'X', 'X', 'X' },
        { 'W', 'W', 'W', 'W', 'W', 'G', 'G', 'G', 'G', 'X' },
        { 'W', 'R', 'R', 'R', 'R', 'R', 'G', 'X', 'X', 'X' },
        { 'W', 'W', 'W', 'R', 'R', 'G', 'G', 'X', 'X', 'X' },
        { 'W', 'B', 'W', 'R', 'R', 'R', 'R', 'R', 'R', 'X' },
        { 'W', 'B', 'B', 'B', 'B', 'R', 'R', 'X', 'X', 'X' },
        { 'W', 'B', 'B', 'X', 'B', 'B', 'B', 'B', 'X', 'X' },
        { 'W', 'B', 'B', 'X', 'X', 'X', 'X', 'X', 'X', 'X' }
    };

    // start node
    int x = 3, y = 9;    // having target color `X`

    // replacement color
    char replacement = 'C';

    // replace the target color with a replacement color
    floodfill(mat, x, y, replacement);

    // print the colors after replacement
    printMatrix(mat);

    return 0;
}
```

Output:

```
Y  Y  Y  G  G  G  G  G  G  G
Y  Y  Y  Y  Y  Y  G  C  C  C
G  G  G  G  G  G  G  C  C  C
W  W  W  W  W  G  G  G  G  C
W  R  R  R  R  R  G  C  C  C
W  W  W  R  R  G  G  C  C  C
W  B  W  R  R  R  R  R  R  C
W  B  B  B  B  R  R  C  C  C
W  B  B  C  B  B  B  B  C  C
W  B  B  C  C  C  C  C  C  C
```

//DFS

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

// Below arrays detail all eight possible movements
int row[] = { -1, -1, -1, 0, 0, 1, 1, 1 };
int col[] = { -1, 0, 1, -1, 1, -1, 0, 1 };

// check if it is possible to go to pixel (x, y) from the
// current pixel. The function returns false if the pixel
// has a different color, or it's not a valid pixel
bool isSafe(vector<vector<char>> const &mat, int x, int y, char target)
{
   return (x >= 0 && x < mat.size()) && (y >= 0 && y < mat[0].size())
      && mat[x][y] == target;
}

// Flood fill using DFS
void floodfill(vector<vector<char>> &mat, int x, int y, char replacement)
{
   // base case
   if (mat.size() == 0) {
      return;
   }

   // get the target color
   char target = mat[x][y];

   // target color is same as replacement
   if (target == replacement) {
      return;
   }

   // replace the current pixel color with that of replacement
   mat[x][y] = replacement;

   // process all eight adjacent pixels of the current pixel and
   // recur for each valid pixel
   for (int k = 0; k < 8; k++)
   {
      // if the adjacent pixel at position (x + row[k], y + col[k]) is
      // a valid pixel and has the same color as that of the current pixel
      if (isSafe(mat, x + row[k], y + col[k], target)) {
         floodfill(mat, x + row[k], y + col[k], replacement);
      }
   }
}
```

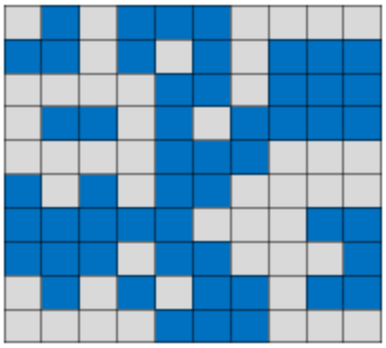
```
// Utility function to print a matrix
void printMatrix(vector<vector<char>> const &mat)
{
    for (int i = 0; i < mat.size(); i++)
    {
        for (int j = 0; j < mat[0].size(); j++) {
            cout << setw(3) << mat[i][j];
        }
        cout << endl;
    }
}

int main()
{
    // matrix showing portion of the screen having different colors
    vector<vector<char>> mat =
    {
        { 'Y', 'Y', 'Y', 'G', 'G', 'G', 'G', 'G', 'G', 'G' },
        { 'Y', 'Y', 'Y', 'Y', 'Y', 'Y', 'G', 'X', 'X', 'X' },
        { 'G', 'G', 'G', 'G', 'G', 'G', 'G', 'X', 'X', 'X' },
        { 'W', 'W', 'W', 'W', 'W', 'G', 'G', 'G', 'G', 'X' },
        { 'W', 'R', 'R', 'R', 'R', 'R', 'G', 'X', 'X', 'X' },
        { 'W', 'W', 'W', 'R', 'R', 'G', 'G', 'X', 'X', 'X' },
        { 'W', 'B', 'W', 'R', 'R', 'R', 'R', 'R', 'R', 'X' },
        { 'W', 'B', 'B', 'B', 'B', 'R', 'R', 'X', 'X', 'X' },
        { 'W', 'B', 'B', 'X', 'B', 'B', 'B', 'B', 'X', 'X' },
        { 'W', 'B', 'B', 'X', 'X', 'X', 'X', 'X', 'X', 'X' }
    };
    // start node
    int x = 3, y = 9;    // having a target color `X`
 // replacement color
    char replacement = 'C';
    // replace the target color with a replacement color using DFS
    floodfill(mat, x, y, replacement);
    // print the colors after replacement
    printMatrix(mat);

    return 0;
}
```

Output:

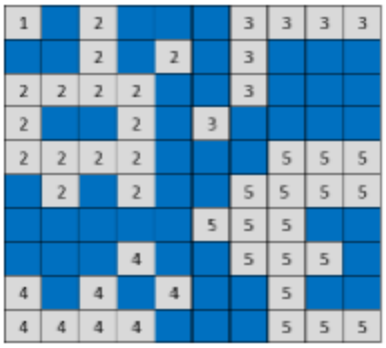
```
 Y   Y   Y   G   G   G   G   G   G   G
 Y   Y   Y   Y   Y   Y   G   C   C   C
 G   G   G   G   G   G   G   C   C   C
 W   W   W   W   W   G   G   G   G   C
 W   R   R   R   R   R   G   C   C   C
 W   W   W   R   R   G   G   C   C   C
 W   B   W   R   R   R   R   R   R   C
 W   B   B   B   B   R   R   C   C   C
 W   B   B   C   B   B   B   B   C   C
 W   B   B   C   C   C   C   C   C   C
```

Given a binary matrix where 0 represents water and 1 represents land, and connected ones form an island, count the total islands.

For example, consider the following image:



The above image highlights water in blue and land in gray in a `10 × 10` matrix. There are a total of **five islands** present in the above matrix. They are marked by the numbers `1-5` in the image below.



//BFS

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

// Below arrays detail all eight possible movements from a cell
// (top, right, bottom, left, and four diagonal moves)
int row[] = { -1, -1, -1, 0, 1, 0, 1, 1 };
int col[] = { -1, 1, 0, -1, -1, 1, 0, 1 };

// Function to check if it is safe to go to position (x, y)
// from the current position. The function returns false if (x, y)
// is not valid matrix coordinates or (x, y) represents water or
// position (x, y) is already processed
```

```
bool isSafe(vector<vector<int>> const &mat, int x, int y,
      vector<vector<bool>> const &processed)
{
   return (x >= 0 && x < mat.size()) && (y >= 0 && y < mat[0].size()) &&
      mat[x][y] && !processed[x][y];
}

void BFS(vector<vector<int>> const &mat, vector<vector<bool>> &processed, int i, int j)
{
   // create an empty queue and enqueue source node
   queue<pair<int, int>> q;
   q.push(make_pair(i, j));

   // mark source node as processed
   processed[i][j] = true;

   // loop till queue is empty
   while (!q.empty())
   {
      // dequeue front node and process it
      int x = q.front().first;
      int y = q.front().second;
      q.pop();

      // check for all eight possible movements from the current cell
      // and enqueue each valid movement
      for (int k = 0; k < 8; k++)
      {
         // skip if the location is invalid, or already
         // processed, or consists of water
         if (isSafe(mat, x + row[k], y + col[k], processed))
         {
            // mark it as processed and enqueue it
            processed[x + row[k]][y + col[k]] = 1;
            q.push(make_pair(x + row[k], y + col[k]));
         }
      }
   }
}

int countIslands(vector<vector<int>> const &mat)
{
   // base case
   if (mat.size() == 0) {
      return 0;
   }

   // `M × N` matrix
   int M = mat.size();
```

```
    int N = mat[0].size();

    // stores if a cell is processed or not
    vector<vector<bool>> processed(M, vector<bool>(N));

    int island = 0;
    for (int i = 0; i < M; i++)
    {
        for (int j = 0; j < N; j++)
        {
            // start BFS from each unprocessed node and increment island count
            if (mat[i][j] && processed[i][j] == 0)
            {
                BFS(mat, processed, i, j);
                island++;
            }
        }
    }

    return island;
}

int main()
{
    vector<vector<int>> mat =
    {
        { 1, 0, 1, 0, 0, 0, 1, 1, 1, 1 },
        { 0, 0, 1, 0, 1, 0, 1, 0, 0, 0 },
        { 1, 1, 1, 1, 0, 0, 1, 0, 0, 0 },
        { 1, 0, 0, 1, 0, 1, 0, 0, 0, 0 },
        { 1, 1, 1, 1, 0, 0, 0, 1, 1, 1 },
        { 0, 1, 0, 1, 0, 0, 1, 1, 1, 1 },
        { 0, 0, 0, 0, 0, 1, 1, 1, 0, 0 },
        { 0, 0, 0, 1, 0, 0, 1, 1, 1, 0 },
        { 1, 0, 1, 0, 1, 0, 0, 1, 0, 0 },
        { 1, 1, 1, 1, 0, 0, 0, 1, 1, 1 }
    };

    cout << "The total number of islands is " << countIslands(mat) << endl;

    return 0;
}
```

Output:

```
The total number of islands is 5
```

Find the shortest path from source to destination in a matrix that satisfies given constraints

Given an `N × N` matrix of positive integers, find the shortest path from the first cell of the matrix to its last cell that satisfies given constraints.

We are allowed to move exactly `k` steps from any cell in the matrix where `k` is the cell's value, i.e., from a cell `(i, j)` having value `k` in a matrix `M`, we can move to `(i+k, j)`, `(i-k, j)`, `(i, j+k)`, or `(i, j-k)`. The diagonal moves are not allowed.

For example,

```
Input:

[ 7  1  3  5  3  6  1  1  7  5 ]
[ 2  3  6  1  1  6  6  6  1  2 ]
[ 6  1  7  2  1  4  7  6  6  2 ]
[ 6  6  7  1  3  3  5  1  3  4 ]
[ 5  5  6  1  5  4  6  1  7  4 ]
[ 3  5  5  2  7  5  3  4  3  6 ]
[ 4  1  4  3  6  4  5  3  2  6 ]
[ 4  4  1  7  4  3  3  1  4  2 ]
[ 4  4  5  1  5  2  3  5  3  5 ]
[ 3  6  3  5  2  2  6  4  2  1 ]

Output:

The shortest path length is 6
The shortest path is (0, 0) (0, 7) (0, 6) (1, 6) (7, 6) (7, 9) (9, 9)
```

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

// A queue node used in BFS
typedef pair<int, int> Node;

// Below arrays detail all four possible movements from a cell
int row[] = { -1, 0, 0, 1 };
int col[] = { 0, -1, 1, 0 };

// Function to check if it is possible to go to position `pt`
// from the current position. The function returns false if `pt` is
// not in a valid position, or it is already visited
bool isValid(Node pt, map<Node, int> visited, int N)
{
    return (pt.first >= 0) && (pt.first < N) &&
        (pt.second >= 0) && (pt.second < N) && !visited.count(pt);
}

// Find the shortest path length in a matrix from source cell (x, y) to
// destination cell (N-1, N-1)
int findPath(vector<vector<int>> const &mat, int x, int y)
{
    // `N × N` matrix
    int N = mat.size();
```

```
// base case
if (N == 0) {
   return -1;
}

// create a queue and enqueue the first node
queue<Node> q;
Node src = {x, y};
q.push(src);

// map to check if the matrix cell is visited before or not. It also
// stores the shortest distance info, i.e., the value corresponding
// to a node in the map represents its shortest distance from the source
map<Node, int> visited;
visited[src] = 0;

// loop till queue is empty
while (!q.empty())
{
   // dequeue front node and process it
   Node node = q.front();
   q.pop();

   int i = node.first;
   int j = node.second;
   int dist = visited[node];

   // if the destination is found, return true
   if (i == N - 1 && j == N - 1) {
      return dist;
   }

   // value of the current cell
   int n = mat[i][j];

   // check all four possible movements from the current cell
   // and recur for each valid movement
   for (int k = 0; k < 4; k++)
   {
      // get the next position using the value of the current cell
      Node next = {(i + row[k] * n), (j + col[k] * n)};

      // check if it is possible to go to position (x, y)
      // from the current position
      if (isValid(next, visited, N))
      {
         q.push(next);
         visited[next] = dist + 1;
      }
   }
```

```
    }

    // return a negative number if the path is not possible
    return -1;
}

int main()
{
    vector<vector<int>> matrix =
    {
        { 7, 1, 3, 5, 3, 6, 1, 1, 7, 5 },
        { 2, 3, 6, 1, 1, 6, 6, 6, 1, 2 },
        { 6, 1, 7, 2, 1, 4, 7, 6, 6, 2 },
        { 6, 6, 7, 1, 3, 3, 5, 1, 3, 4 },
        { 5, 5, 6, 1, 5, 4, 6, 1, 7, 4 },
        { 3, 5, 5, 2, 7, 5, 3, 4, 3, 6 },
        { 4, 1, 4, 3, 6, 4, 5, 3, 2, 6 },
        { 4, 4, 1, 7, 4, 3, 3, 1, 4, 2 },
        { 4, 4, 5, 1, 5, 2, 3, 5, 3, 5 },
        { 3, 6, 3, 5, 2, 2, 6, 4, 2, 1 }
    };

    // Find a route in the matrix from source cell (0, 0) to
    // destination cell (N-1, N-1)
    int len = findPath(matrix, 0, 0);

    if (len != -1) {
        cout << "The shortest path length is " << len;
    }
    else {
        cout << "Destination not possible";
    }

    return 0;
}
```