DFS:

Code:

```
import java.io.*;
import java.util.*;

class Graph {
   private int V;
   private LinkedList<Integer> adj[];
   @SuppressWarnings("unchecked") Graph(int v)
   {
           V = v;
           adj = new LinkedList[v];
           for (int i = 0; i < v; ++i)
                     adj[i] = new LinkedList();
   }
   void addEdge(int v, int w)
   {
           adj[v].add(w); // Add w to v's list.
   }
   void DFSUtil(int v, boolean visited[])
   {
           visited[v] = true;
           System.out.print(v + " ");
           Iterator<Integer> i = adj[v].listIterator();
           while (i.hasNext()) {
                     int n = i.next();
                     if (!visited[n])
                               DFSUtil(n, visited);
           }
   }

   void DFS(int v)
   {
           boolean visited[] = new boolean[V];

           DFSUtil(v, visited);
   }
   public static void main(String args[])
   {
           Graph g = new Graph(4);

           g.addEdge(0, 1);
           g.addEdge(0, 2);
```

```
        g.addEdge(1, 2);
        g.addEdge(2, 0);
        g.addEdge(2, 3);
        g.addEdge(3, 3);

        System.out.println(
            "Following is Depth First Traversal "
            + "(starting from vertex 2)");
        g.DFS(2);
    }
}
```

Output:

```
Depth First Traversal
2 0 1 3
```

BFS:
Code:

```
import java.io.*;
import java.util.*;
class Graph {
    private int V; // No. of vertices
    private LinkedList<Integer> adj[]; // Adjacency Lists
    Graph(int v)
    {
        V = v;
        adj = new LinkedList[v];
        for (int i = 0; i < v; ++i)
            adj[i] = new LinkedList();
    }
    void addEdge(int v, int w) { adj[v].add(w); }
    void BFS(int s)
    {

        boolean visited[] = new boolean[V];
        LinkedList<Integer> queue
            = new LinkedList<Integer>();
        visited[s] = true;
        queue.add(s);

        while (queue.size() != 0) {
            s = queue.poll();
            System.out.print(s + " ");
            Iterator<Integer> i = adj[s].listIterator();
            while (i.hasNext()) {
                int n = i.next();
                if (!visited[n]) {
                    visited[n] = true;
                    queue.add(n);
                }
            }
        }
    }
    public static void main(String args[])
    {
        Graph g = new Graph(4);

        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 2);
        g.addEdge(2, 0);
        g.addEdge(2, 3);
        g.addEdge(3, 3);

        System.out.println(
            "Following is Breadth First Traversal "
            + "(starting from vertex 2)");

        g.BFS(2);
    }
}
```

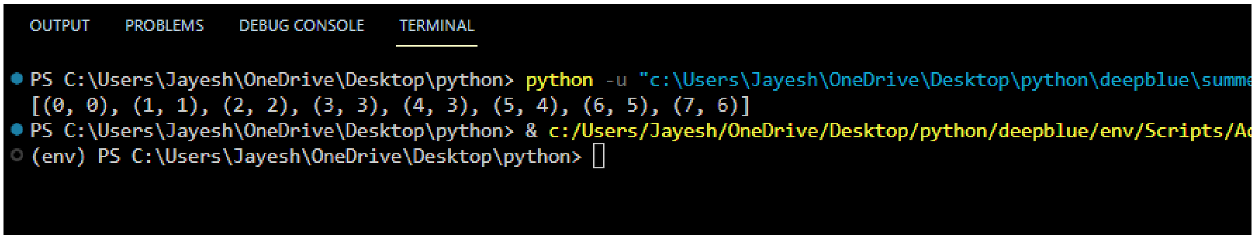
Output:

```
Breadth First Traversal
2 0 3 1
```

A*
Code:

```
class Node():
    """A node class for A* Pathfinding"""

    def __init__(self, parent=None, position=None):
    self.parent = parent
    self.position = position

    self.g = 0
    self.h = 0
    self.f = 0

    def __eq__(self, other):
    return self.position == other.position


def astar(maze, start, end):
    """Returns a list of tuples as a path from the given start to the given end in the given maze"""
    start_node = Node(None, start)
    start_node.g = start_node.h = start_node.f = 0
    end_node = Node(None, end)
    end_node.g = end_node.h = end_node.f = 0

    open_list = []
    closed_list = []
    open_list.append(start_node)
    while len(open_list) > 0:
    current_node = open_list[0]
    current_index = 0
    for index, item in enumerate(open_list):
    if item.f < current_node.f:
    current_node = item
    current_index = index
    closed_list.append(current_node)
    if current_node == end_node:
    path = []
    current = current_node
    while current is not None:
    path.append(current.position)
    current = current.parent
    return path[::-1] # Return reversed path
    children = []
    for new_position in [(0, -1), (0, 1), (-1, 0), (1, 0), (-1, -1), (-1, 1), (1, -1), (1, 1)]: # Adjacent
squares

    node_position = (current_node.position[0] + new_position[0], current_node.position[1] +
new_position[1])

    if node_position[0] > (len(maze) - 1) or node_position[0] < 0 or node_position[1] >
(len(maze[len(maze)-1]) -1) or node_position[1] < 0:
    continue

    if maze[node_position[0]][node_position[1]] != 0:
    continue

    new_node = Node(current_node, node_position)

    children.append(new_node)

    for child in children:

    for closed_child in closed_list:
    if child == closed_child:
        continue

    # Create the f, g, and h values
    child.g = current_node.g + 1
    child.h = ((child.position[0] - end_node.position[0]) ** 2) + ((child.position[1] -
end_node.position[1]) ** 2)
    child.f = child.g + child.h

    for open_node in open_list:
    if child == open_node and child.g > open_node.g:
        continue

    open_list.append(child)


def main():

    maze = [[0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]

    start = (0, 0)
    end = (7, 6)
    path = astar(maze, start, end)
    print(path)
if __name__ == '__main__':
    main()
```

Output:



```
● PS C:\Users\Jayesh\OneDrive\Desktop\python> python -u "c:\Users\Jayesh\OneDrive\Desktop\python\deepblue\summe
  [(0, 0), (1, 1), (2, 2), (3, 3), (4, 3), (5, 4), (6, 5), (7, 6)]
● PS C:\Users\Jayesh\OneDrive\Desktop\python> & c:/Users/Jayesh/OneDrive/Desktop/python/deepblue/env/Scripts/Ac
○ (env) PS C:\Users\Jayesh\OneDrive\Desktop\python> 
```