A ***graph*** *G is* a set *V* of ***vertices*** and a collection *E* of pairs of vertices from *V*, called ***edges***. The aim of this worksheet is to use an implementation of the Graph ADT based on the ***adjacency matrix r***epresentation.

As illustrated in figure 1, with this representation the set ***V* of *vertices*** are stored in an **ArrayList** and the set of edges are represented in a **matrix**.

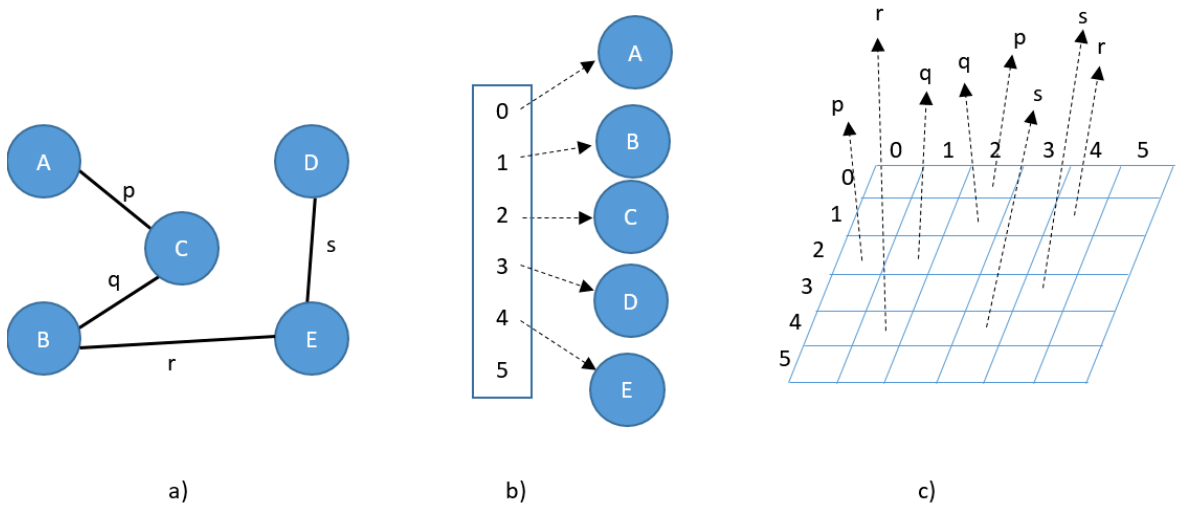


Figure1 - (a) An undirected graph G; (b) vertices' ArrayList; (c) edges' matrix

Consider there is a basic **graph interface** specified in **BasicGraph (**generic parameters V and E designate the element type stored respectively at vertices and edges**)*:***

```
public interface BasicGraph<V,E> {
  int numVertices();
  int numEdges();
  Iterable<V> vertices();
  Iterable<E> edges();
  int outDegree(V vertex);
  int inDegree(V vertex);
  Iterable<E> outgoingEdges(V vertex);
  Iterable<E> incomingEdges(V vertex);
  E getEdge(V va, V vb);
  V[] endVertices(E edge);
  boolean insertVertex(V newVertex);
  boolean insertEdge(V va, V vb, E newEdge);
  boolean removeVertex(V vertex);
  E removeEdge(V va, V vb);
}
```

Which is implemented by a **AdjacencyMatrixGraph<V,E>** generic class, using an adjacency matrix:

```
public class AdjacencyMatrixGraph<V, E> implements BasicGraph<V, E>,
                                                   Cloneable {

    ArrayList<V> vertices;
    E[][] edgeMatrix;

    // .. rest not shown
```

There is also a generic class **GraphAlgorithms** that implements the algorithms for graph visits and paths:

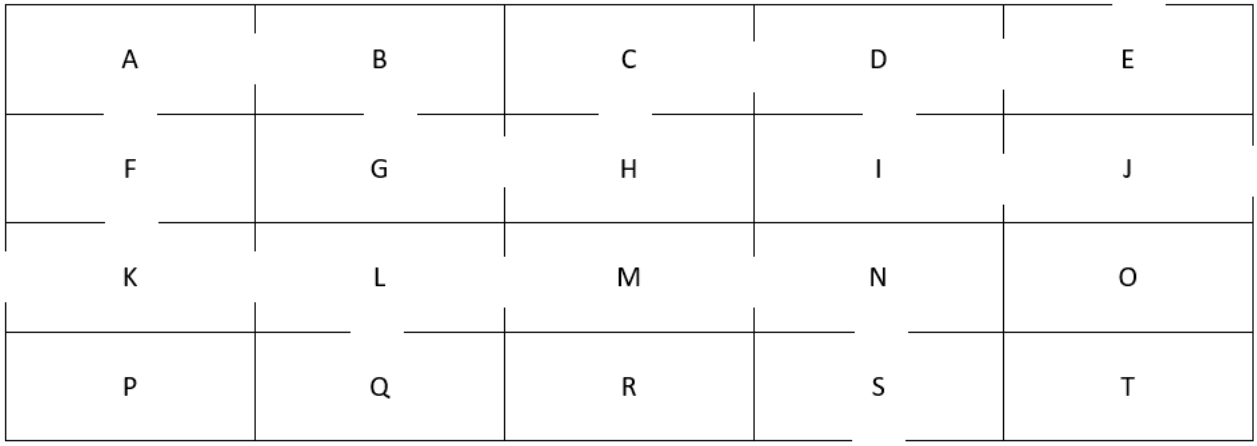
```
public class GraphAlgorithms {

    public static <V,E> LinkedList<V> DFS(AdjacencyMatrixGraph<V,E> graph,
                                          V vertex)
      {// code not shown}

    public static <V,E> LinkedList<V> BFS(AdjacencyMatrixGraph<V,E> graph,
                                          V vertex)
      {// code not shown

    public static <V,E> boolean allPaths(AdjacencyMatrixGraph<V,E> graph,
                        V source, V dest, LinkedList<LinkedList<V>> paths) {
      {// code not shown}
}
```

1. Implement a **LabyrinthCheater** class, which is intended to represent a labyrinth through a map of rooms and doors. Each room has a name and potentially provides an exit. Doors represent connections between rooms.
   - **a)** Declare the class, and its attributes, using an adjacency map graph to represent the labyrinth (suggestion: create private inner classes to represent rooms and connection)
   - **b)** Implement a method to list all rooms which are reachable from a particular room (identified by name)
   - **c)** Implement a (slightly cheating) method to return the name of the nearest room with an exit (from a particular room)
   - **d)** Implement a (really cheating) method to return the path with the sequence of rooms to reach the nearest room with an exit (from a particular room)
   - **e)** Create unit tests for methods developed in b. to d.



Rooms which are reachable from A: A, B, G, H, C, D, E, I, J, F, K, L, M, N, S, Q

Nearest exit from A: K

Path to nearest exit from A: A, F, K

Complementary Exercises

1. There are eight small islands in a lake, and the state wants to build seven bridges to connect them so that each island can be reached from any other one via one or more bridges. The cost of constructing a bridge is proportional to its length. The distances between pairs of islands are given in the following table.

-	240	210	340	280	200	345	120
-	-	265	175	215	180	185	155
-	-	-	260	115	350	435	195
-	-	-	-	160	330	295	230
-	-	-	-	-	360	400	170
-	-	-	-	-	-	175	205
-	-	-	-	-	-	-	305
-	-	-	-	-	-	-	-

Find which bridges to build to minimize the total construction cost.