ñ

ESTRUCTURAS DE DATOS

2024

TRABAJO PRACTICO N°4
Grafos

Profesores:
HECTOR REINAGA
FERNANDA DANIELA OYARZO
MIRTHA FABIANA MIRANDA

Alumno:
GONZALO ALEJANDRO ULLOA

UNPA
Universidad Nacional
de la Patagonia Austral

Índice

Desarrollo

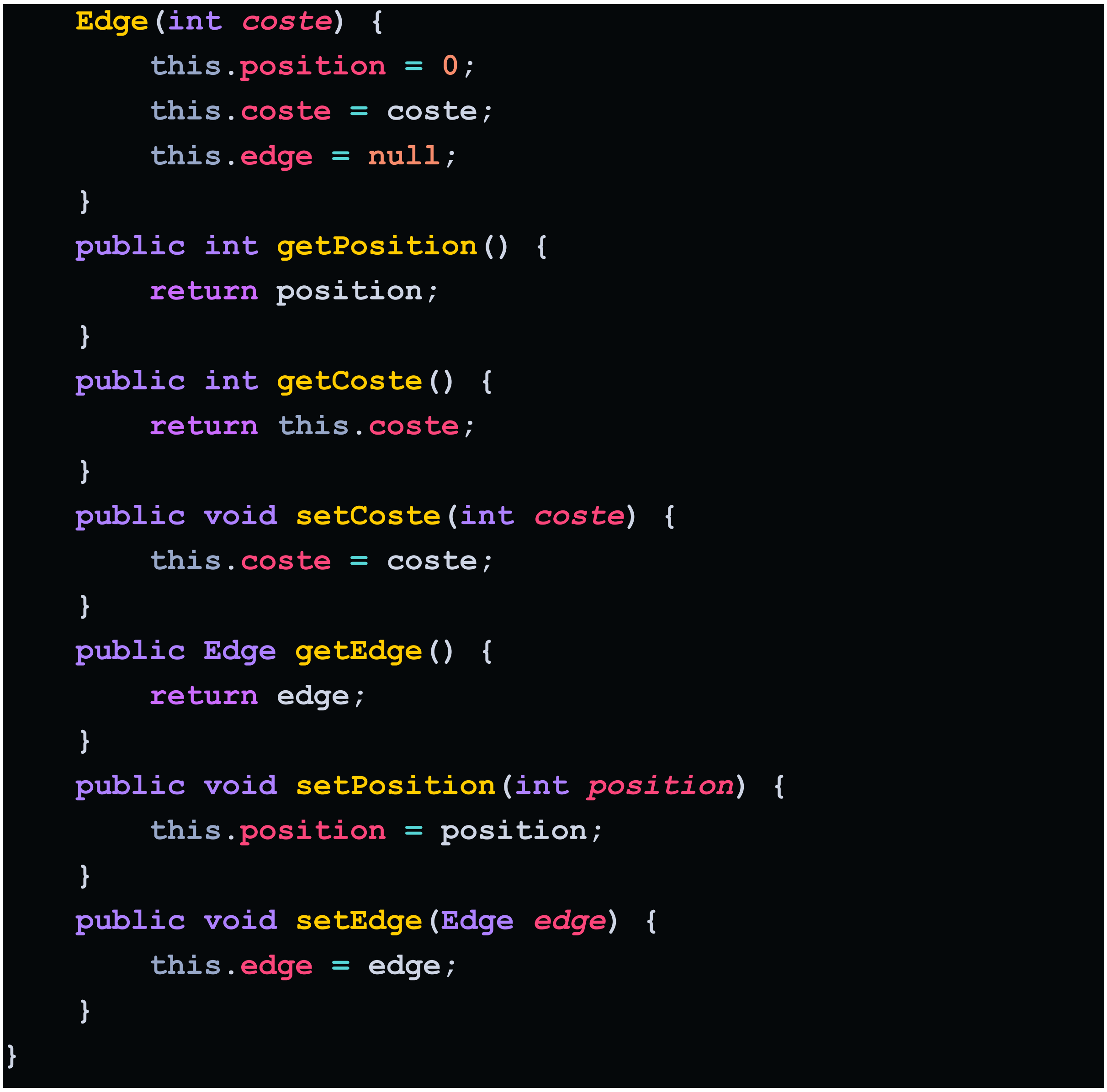
6 y 7. Desarrollé ambos puntos en un mismo main

Clase Vertex

```java
package tp5.MatrizDeAdyacencia;
public class Vertex {
    private Object element;
    private Edge edge;
    Vertex() {
        element = null;
        edge = null;
    }
    public Object getElement() {
        return element;
    }
    public Edge getEdge() {
        return edge;
    }
    public void setElement(Object element) {
        this.element = element;
    }
    public void setEdge(Edge edge) {
        this.edge = edge;
    }
}
```

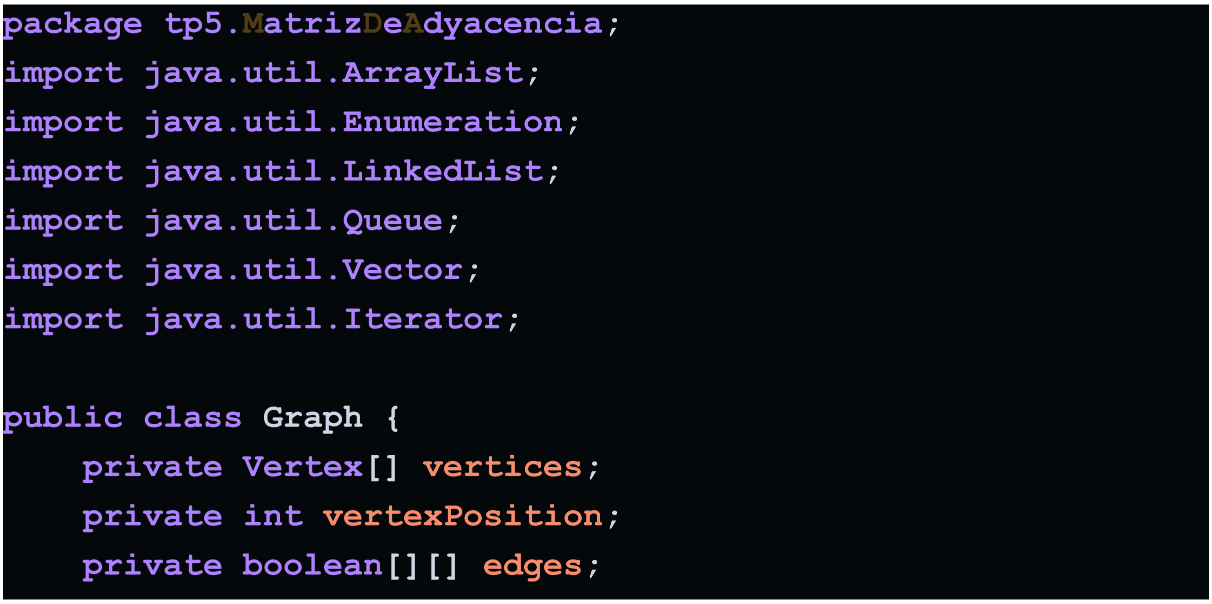
Clase Edge

```java
package tp5.MatrizDeAdyacencia;
public class Edge {
    private int position;
    private Edge edge;
    private int coste;
    Edge() {
        position = 0;
        coste=0;
        edge = null;
    }
```

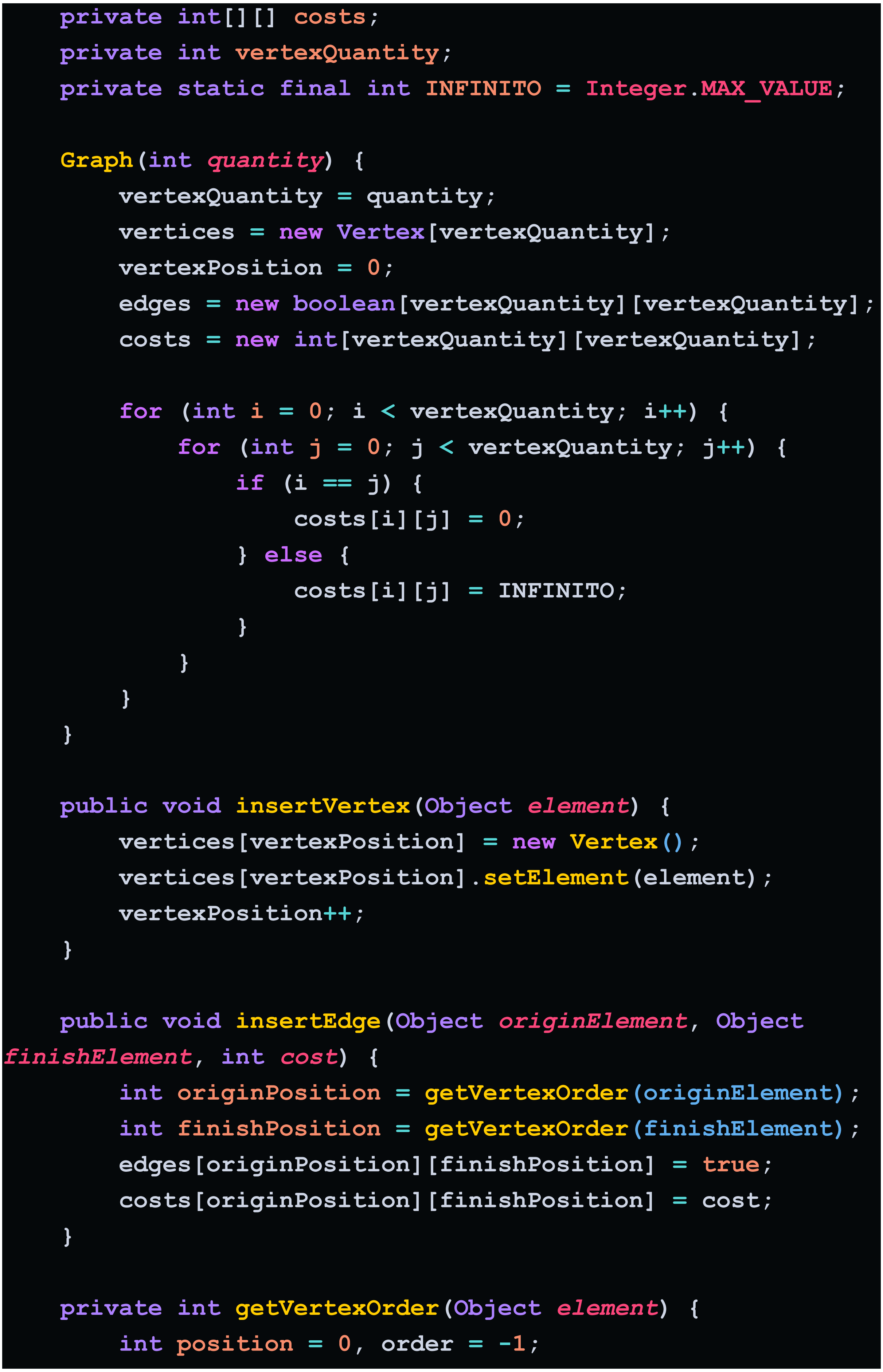


```java
    Edge(int coste) {
        this.position = 0;
        this.coste = coste;
        this.edge = null;
    }
    public int getPosition() {
        return position;
    }
    public int getCoste() {
        return this.coste;
    }
    public void setCoste(int coste) {
        this.coste = coste;
    }
    public Edge getEdge() {
        return edge;
    }
    public void setPosition(int position) {
        this.position = position;
    }
    public void setEdge(Edge edge) {
        this.edge = edge;
    }
}
```

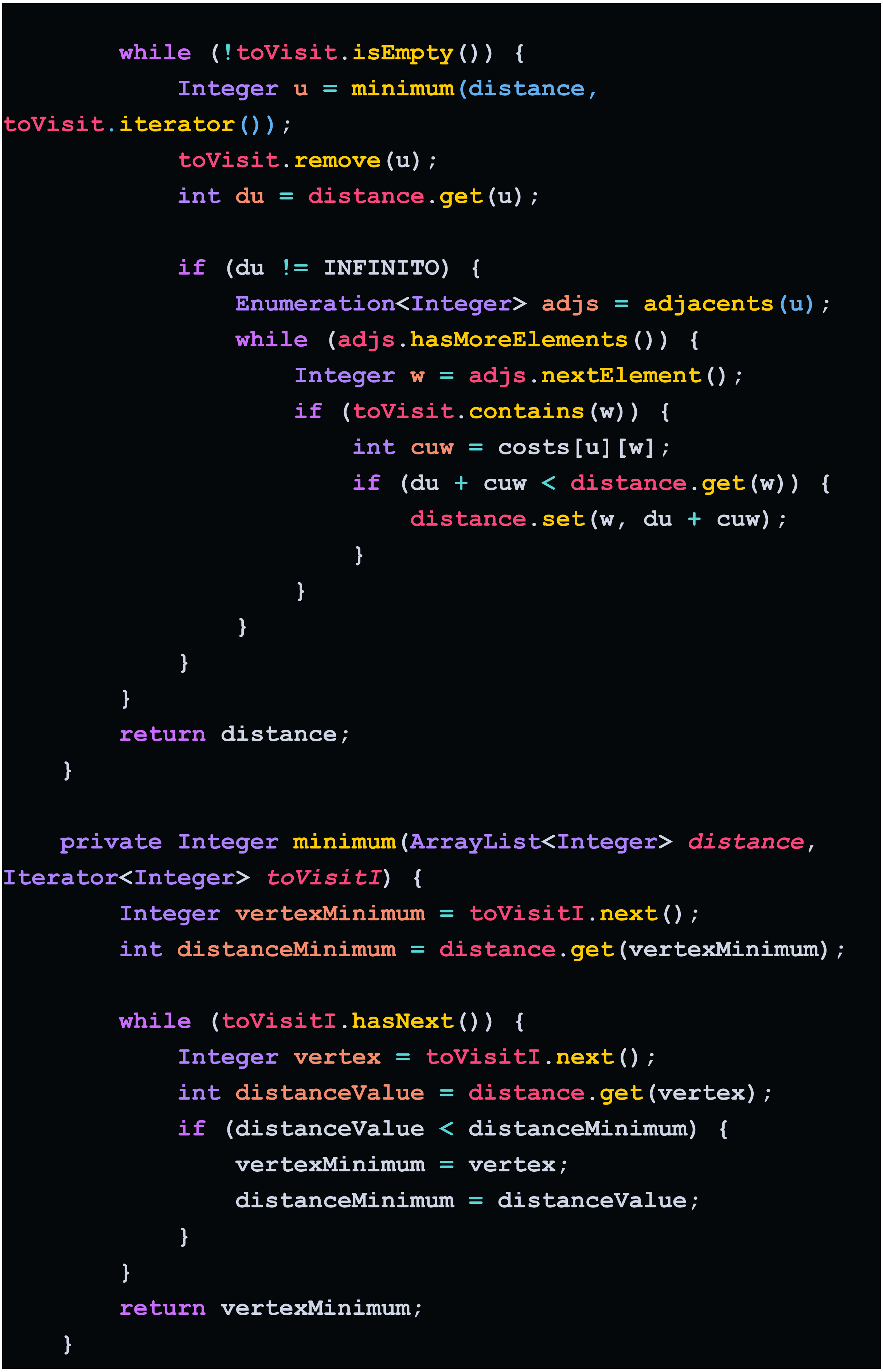
Clase Graph



```java
package tp5.MatrizDeAdyacencia;
import java.util.ArrayList;
import java.util.Enumeration;
import java.util.LinkedList;
import java.util.Queue;
import java.util.Vector;
import java.util.Iterator;

public class Graph {
    private Vertex[] vertices;
    private int vertexPosition;
    private boolean[][] edges;
```



```java
    private int[][] costs;
    private int vertexQuantity;
    private static final int INFINITO = Integer.MAX_VALUE;

    Graph(int quantity) {
        vertexQuantity = quantity;
        vertices = new Vertex[vertexQuantity];
        vertexPosition = 0;
        edges = new boolean[vertexQuantity][vertexQuantity];
        costs = new int[vertexQuantity][vertexQuantity];

        for (int i = 0; i < vertexQuantity; i++) {
            for (int j = 0; j < vertexQuantity; j++) {
                if (i == j) {
                    costs[i][j] = 0;
                } else {
                    costs[i][j] = INFINITO;
                }
            }
        }
    }

    public void insertVertex(Object element) {
        vertices[vertexPosition] = new Vertex();
        vertices[vertexPosition].setElement(element);
        vertexPosition++;
    }

    public void insertEdge(Object originElement, Object
finishElement, int cost) {
        int originPosition = getVertexOrder(originElement);
        int finishPosition = getVertexOrder(finishElement);
        edges[originPosition][finishPosition] = true;
        costs[originPosition][finishPosition] = cost;
    }

    private int getVertexOrder(Object element) {
        int position = 0, order = -1;
        boolean found = false;
        while (position < vertexQuantity & found == false) {
            if
(vertices[position].getElement().equals(element)) {
                found = true;
                order = position;
            }
            position++;
        }
        return order;
    }
    public void depthFirstSearch(Object element){
        Vector visited = new Vector(vertexQuantity);
        depthFirst(getVertexOrder(element), visited);
    }

    private void depthFirst(int element, Vector visited){
        System.out.print(vertices[element].getElement() + "
");
        visited.addElement(new Integer(element));
        Enumeration adjs = adjacents(new Integer(element));
        while (adjs.hasMoreElements()) {
            Integer adjsOther = (Integer)
adjs.nextElement();
            if (!visited.contains(adjsOther)) {
                depthFirst(adjsOther.intValue(), visited);
            }
        }
    }

    public void breadhFirstSearch(Object element) {
        breadhFirst(getVertexOrder(element));
    }
    private void breadhFirst(int element) {
        Vector<Integer> visited = new
Vector(vertexQuantity);
        Queue<Integer> explore = new LinkedList<>();
        explore.add(new Integer(element));
        visited.addElement(new Integer(element));
        do {
            Integer vertexOther = (Integer) explore.poll();

System.out.print(vertices[vertexOther.intValue()].getElement
() + " ");
            Enumeration adjs = adjacents(vertexOther);
            while (adjs.hasMoreElements()) {
                Integer adjsOther = (Integer)
adjs.nextElement();
                if (!visited.contains(adjsOther)) {
                    explore.add(adjsOther);
                    visited.addElement(adjsOther);
                }
            }
        } while (!explore.isEmpty());
    }

    public ArrayList<Integer> dijkstraAlgorithm(Object
vertex) {
        return dijkstra(getVertexOrder(vertex));
    }

    private ArrayList<Integer> dijkstra(int vertex) {
        int vs;
        ArrayList<Integer> distance = new
ArrayList<>(vertexQuantity);
        ArrayList<Integer> toVisit = new
ArrayList<>(vertexQuantity);

        for (vs = 0; vs < vertexQuantity; vs++) {
            if (vs == vertex) {
                distance.add(0);
            } else {
                distance.add(INFINITO);
            }
            toVisit.add(vs);
        }
```

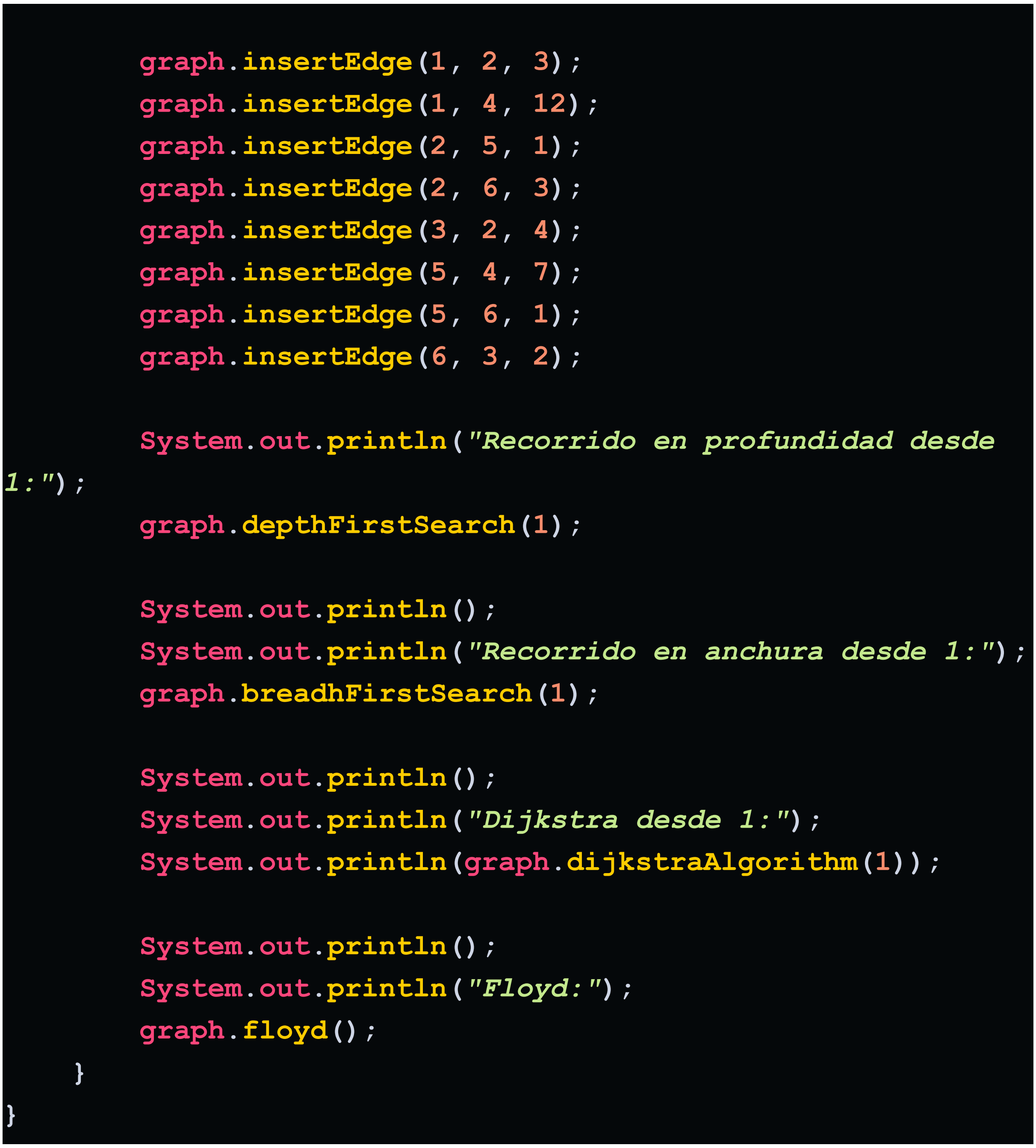


```
        while (!toVisit.isEmpty()) {
            Integer u = minimum(distance,
toVisit.iterator());
            toVisit.remove(u);
            int du = distance.get(u);

            if (du != INFINITO) {
                Enumeration<Integer> adjs = adjacents(u);
                while (adjs.hasMoreElements()) {
                    Integer w = adjs.nextElement();
                    if (toVisit.contains(w)) {
                        int cuw = costs[u][w];
                        if (du + cuw < distance.get(w)) {
                            distance.set(w, du + cuw);
                        }
                    }
                }
            }
        }
        return distance;
    }

    private Integer minimum(ArrayList<Integer> distance,
Iterator<Integer> toVisitI) {
        Integer vertexMinimum = toVisitI.next();
        int distanceMinimum = distance.get(vertexMinimum);

        while (toVisitI.hasNext()) {
            Integer vertex = toVisitI.next();
            int distanceValue = distance.get(vertex);
            if (distanceValue < distanceMinimum) {
                vertexMinimum = vertex;
                distanceMinimum = distanceValue;
            }
        }
        return vertexMinimum;
    }
    private Enumeration<Integer> adjacents(Integer element)
{
        Vector<Integer> adjVertices = new Vector<>();
        for (int i = 0; i < vertexQuantity; i++) {
            if (edges[element][i]) {
                adjVertices.add(i);
            }
        }
        return adjVertices.elements();
    }

    // floyd con matriz P
    public int[][] floyd() {
        int[][] floydMatrix = new
int[vertexQuantity][vertexQuantity];
        int[][] P = new int[vertexQuantity][vertexQuantity];

        for (int i = 0; i < vertexQuantity; i++) {
            for (int j = 0; j < vertexQuantity; j++) {
                floydMatrix[i][j] = costs[i][j];
                if (i != j && costs[i][j] < INFINITO) {
                    P[i][j] = i;
                } else {
                    P[i][j] = -1;
                }
            }
        }
        for (int k = 0; k < vertexQuantity; k++) {
            for (int i = 0; i < vertexQuantity; i++) {
                for (int j = 0; j < vertexQuantity; j++) {
                    if (floydMatrix[i][k] != INFINITO &&
floydMatrix[k][j] != INFINITO &&
                        floydMatrix[i][j] >
floydMatrix[i][k] + floydMatrix[k][j]) {

                        floydMatrix[i][j] =
floydMatrix[i][k] + floydMatrix[k][j];
```

```
                        P[i][j] = P[k][j];
                    }
                }
            }
        }
        // matriz de costos y predecesores
        System.out.println("Matris de costos de Floyd:");
        System.out.println("-  1\t2\t3\t4\t5\t6");
        mostrarMatriz(floydMatrix);
        System.out.println("Matriz de predecesores P:");
        System.out.println("-  1\t2\t3\t4\t5\t6");
        mostrarMatriz(P);

        return floydMatrix;
    }
    private void mostrarMatriz(int[][] matrix) {
        for (int i = 0; i < matrix.length; i++) {
            System.out.print((i+1) + " ");
            for (int j = 0; j < matrix[i].length; j++) {
                System.out.print((matrix[i][j] == INFINITO ?
"INF" : matrix[i][j]) + "\t");
            }
            System.out.println();
        }
    }
}
```

Main

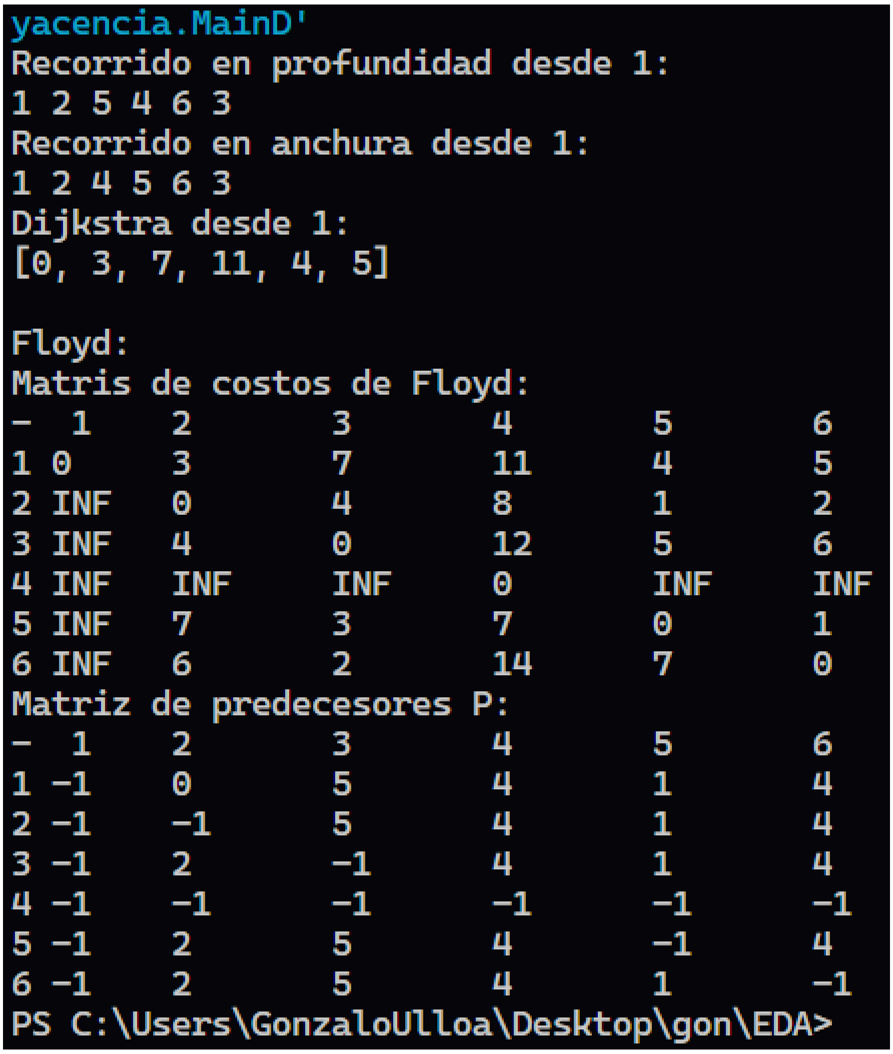
```
package tp5.MatrizDeAdyacencia;
public class MainD {
    public static void main(String[] args) {
        Graph graph = new Graph(6);
        graph.insertVertex(1);
        graph.insertVertex(2);
        graph.insertVertex(3);
        graph.insertVertex(4);
        graph.insertVertex(5);
        graph.insertVertex(6);
```



```
        graph.insertEdge(1, 2, 3);
        graph.insertEdge(1, 4, 12);
        graph.insertEdge(2, 5, 1);
        graph.insertEdge(2, 6, 3);
        graph.insertEdge(3, 2, 4);
        graph.insertEdge(5, 4, 7);
        graph.insertEdge(5, 6, 1);
        graph.insertEdge(6, 3, 2);

        System.out.println("Recorrido en profundidad desde
1:");
        graph.depthFirstSearch(1);

        System.out.println();
        System.out.println("Recorrido en anchura desde 1:");
        graph.breadhFirstSearch(1);

        System.out.println();
        System.out.println("Dijkstra desde 1:");
        System.out.println(graph.dijkstraAlgorithm(1));

        System.out.println();
        System.out.println("Floyd:");
        graph.floyd();
    }
}
```

👇Resultado👇

Resultado consola



```
yacencia.MainD'
Recorrido en profundidad desde 1:
1 2 5 4 6 3
Recorrido en anchura desde 1:
1 2 4 5 6 3
Dijkstra desde 1:
[0, 3, 7, 11, 4, 5]

Floyd:
Matris de costos de Floyd:
-  1    2       3       4       5       6
1 0     3       7       11      4       5
2 INF   0       4       8       1       2
3 INF   4       0       12      5       6
4 INF   INF     INF     0       INF     INF
5 INF   7       3       7       0       1
6 INF   6       2       14      7       0
Matriz de predecesores P:
-  1    2       3       4       5       6
1 -1    0       5       4       1       4
2 -1    -1      5       4       1       4
3 -1    2       -1      4       1       4
4 -1    -1      -1      -1      -1      -1
5 -1    2       5       4       -1      4
6 -1    2       5       4       1       -1
PS C:\Users\GonzaloUlloa\Desktop\gon\EDA>
```