Práctica 03

CURSO 2017-2018

<Programación Dinámica>

FRANCISCO JAVIER MARQUÉS GAONA
JUAN JOSÉ MÉNDEZ CONTRERAS
JOSE ANDRÉS GONZÁLEZ VENTURA

ESTRUCTURA DATOS Y ALGORITMOS II

2º GRADO EN INGENIERÍA INFORMÁTICA

Departamento de Informática
Universidad de Almería

ÍNDICE

PRESENTACIÓN DEL PROBLEMA

Continuamos con el desarrollo de software para la "Metropolitan Transport Authority" (NAMTA) de la expansión urbana de Nueva Almería. En este caso se desea desarrollar la funcionalidad para el cobro por el uso del transporte.

Las condiciones con que se trabaja son:

- a) Las tarifas se fijan anualmente a principios del ejercicio, para ello se generan unas tablas de precios una vez cada año (salvo casos excepcionales).
- b) El usuario paga en función de la distancia recorrida (según el camino óptimo entre el punto de origen y de destino). Hay una cantidad fija de "entrada" (in) y una cantidad correspondiente a la distancia recorrida, calculada según un precio por segmento (sg).
- c) Hay tres tablas de precios (para cada tipo de tarjeta que se describe a continuación) en función de la hora en la que se entra en la red:
    - Horas punta. Ej. 7:00-9:30 y de 17:30-19:30
    - Horas planas. Ej. 6:00-6:59, 9:31-10:30, 16:30-17:29 y 19:31-21:30
    - Horas valles. En este caso, los periodos horarios restantes.

El horario del metro es de 4:00 a 1:00, para entre la 1 y las 4 de la madrugada.
Pueden considerar estos valores como ejemplos, debiendo utilizar parámetros de entrada, permitiendo modificar luego los valores.

Un ejemplo de las tarifas puede ser, según se muestra en la tabla 1, con valores en centavos de dólar (¢).

	Hora punta	Hora plana	Hora valle
Precio entrada (in)	10¢	8¢	5¢
Precio segmento (sg)	12¢	9¢	6¢

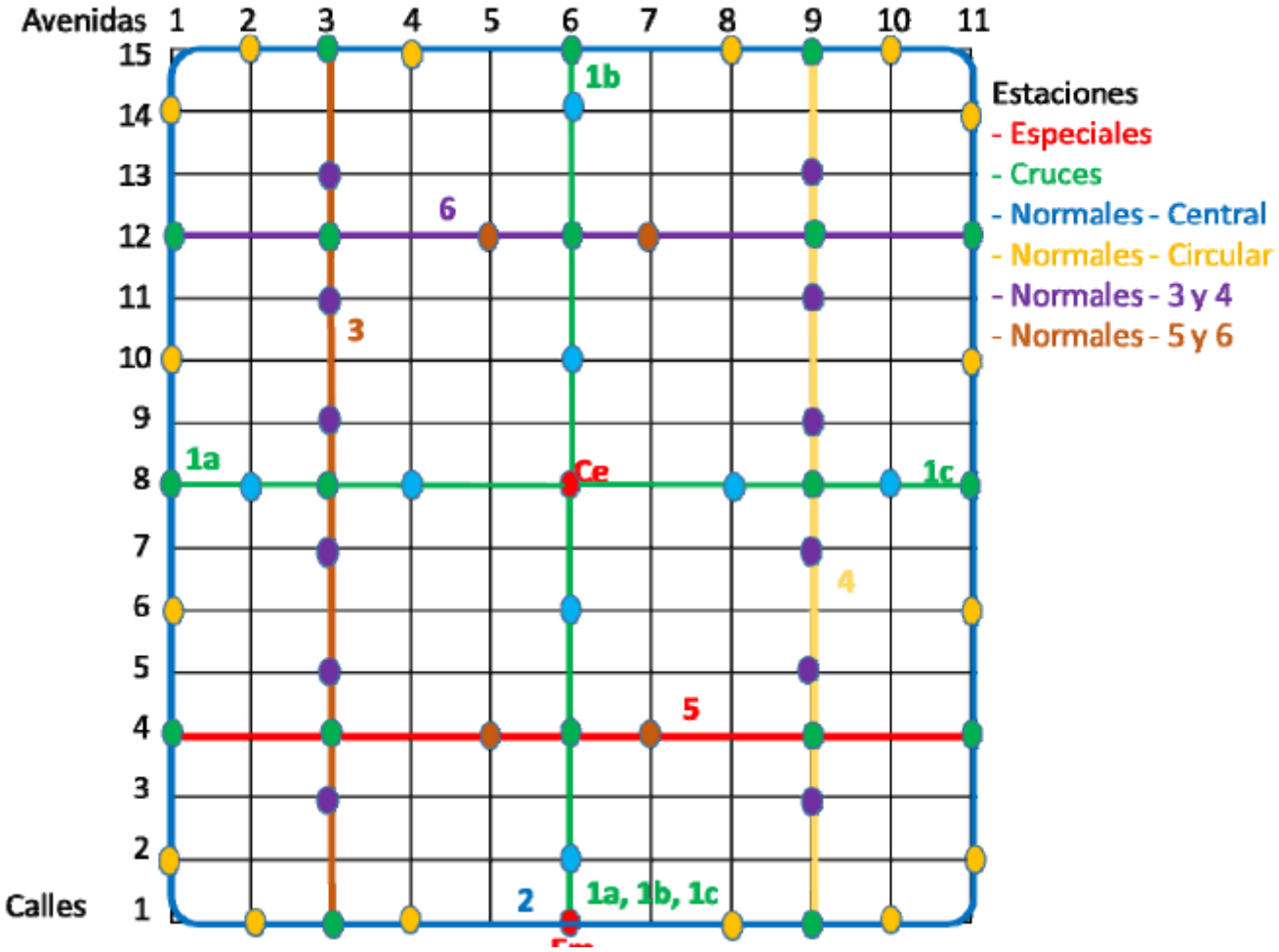
- d) Hay un precio para ticket sencillo, una tarjeta que se compra, indicando en la máquina expendedora (que "sabe" en qué estación está) a qué estación desea ir (utilizando una pantalla táctil). Al salir en la estación designada la tarjeta se retira en el torno. En el torno de entrada se comprueba que la hora es adecuada para la tarjeta (que no ha pasado demasiado tiempo).
- e) Para los usuarios con tarjeta precargada (pueden añadir dinero en cualquier momento en las máquinas expendedoras), hay un precio con descuento de un d% (ejemplo 10%). Se redondea a favor del usuario al centavo de dólar. Al pasar por el torno de entrada se detecta el punto de inicio (y la hora) y al pasar por el torno de salida se descuenta la cantidad correspondiente al trayecto teniendo en cuenta hora de entrada y distancia recorrida.

Así por ejemplo, un usuario que entra en hora plana y recorre la distancia de 10 manzanas (blocks), ej. Ce-(3,1) con una tarjeta simple, pagaría en la máquina expendedora:

Precio: 8 + 10*9 = 98 ¢
Si utilizase una tarjeta abono con 10% de descuento pagaría (descontaría de la tarjeta):

Precio: (8 + 10*9) * 0.9 = 88 ¢

Dada la red de metro descrita en la práctica 2, que para una ciudad con m=11 y n=15 se muestra en la figura 1.



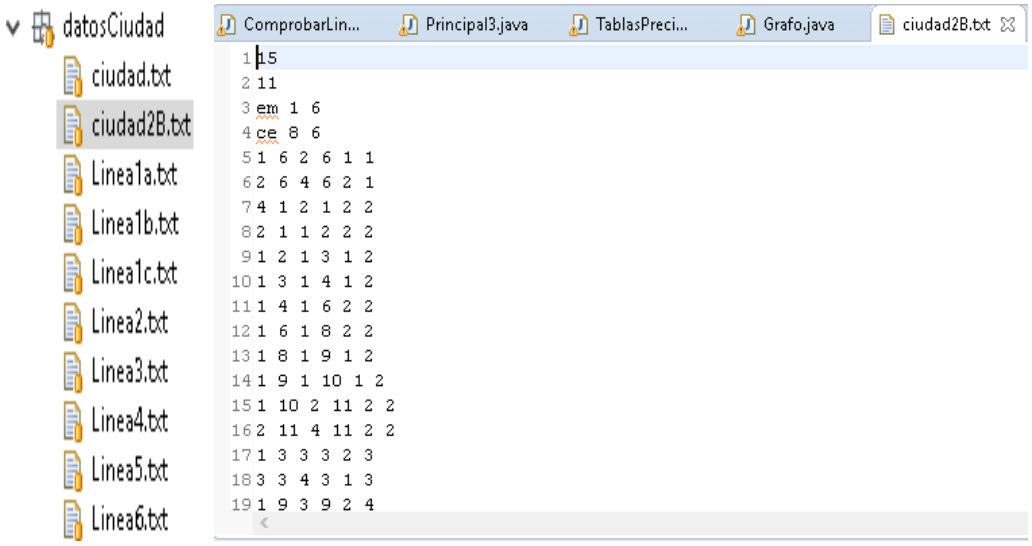
Ha de calcular las tablas de precios que se utilizaran en las máquinas expendedoras para la emisión de las tarjetas de un solo viaje y en los tornos de salida para descontar del saldo de las tarjetas recargables la cantidad consumida. Estas tablas, una para cada periodo horario, se calculan una única vez y se cargan en los distintos dispositivos de la red (máquinas y tornos).

Nota importante: sólo se les pide el algoritmo de cálculo de tarifas, los algoritmos de gestión de tarjetas, de carga y apunte de coste o de comprobación de las tarjetas (saldo y validez) y tiempos (sobre todo para las tarjetas simples) NO han de ser desarrollados.

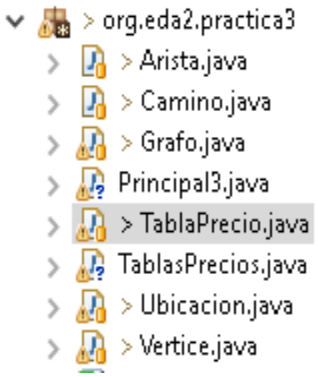
GENERACIÓN DE LOS JUEGOS DE PRUEBA

En esta práctica en concreto hacemos uso de los datos generados en la práctica2 ya que partimos de una ciudad con sus correspondientes líneas para así poder calcular el precio de un determinado recorrido.

A continuación se muestra la carpeta de donde sacamos los datos y el archivo con los caminos posibles determinados por los vértices por los que va pasando:



Al igual que en la práctica anterior hacemos uso de un grafo con sus correspondientes aristas y vértices. Como ya se ha explicado con anterioridad procedemos a explicar las nuevas clases introducidas para la resolución del problema.



Clases práctica03

CLASE TABLAPRECIO

Nuestra clase tablaPrecio tiene 2 atributos, una constante denominada *"DESCUENTO"* que se utilizará para aplicar el descuento (en este caso de un 10%) para las tarjetas con abono. Además también hacemos uso de un array de precios (double) donde se almacenan los precios de una determinada tabla.

CONSTRUCTOR:

En cuanto al constructor no recibe nada como parámetro de entrada. Simplemente establecemos la longitud del array de precios a 6 y para cada elemento del array inicializamos el precio a cero.

```
public TablaPrecio () {
    precio = new double [6];
    for (int i=0; i<precio.length; i++)
        precio[i] = 0;
}
```

METODOS:

- *incrementar(int i, double valor):* Este método aumentará el precio según un valor dado y en la posición del array i.
-

```
public void incrementar (int i, double valor) {
    precio[i] = precio[i] + valor;
}
```

-

-

- *incrementarTodos(double[] importeEntradas):* El método incrementará cada una de las posiciones del array de precios basándose en el array del precio de las entradas.

```
public void incrementarTodos(double[] importeEntradas) {
    for (int i=0; i<precio.length; i++) {
        precio[i] = precio[i] + importeEntradas[i];
    }
}
```

-

- *incrementarTodos(double[] importeSegmentos, double longitud):* Incrementa el array de precios basándose en este caso en el importe de cada segmento que se recorre y la longitud (es decir el nº de segmentos que recorremos).

```
public void incrementarTodos(double[] importeSegmentos,
                             double longitud) {
    for (int i=0; i<precio.length; i++) {
        precio[i] = precio[i] + importeSegmentos[i] * longitud;
    }
}
```

- *reducirAbono():* Reduce el precio de las tarjetas con abono en un 10%.

```
//le resta el 10% a las 3 ultimas tarifas que son las de
//tarjeta abono
public void reducirAbono() {
    for (int i=3; i<precio.length; i++) {
        precio[i] = precio[i] - precio[i] * DESCUENTO;
    }
}
```

CLASE TABLAPRECIOS

Esta clase que extiende de la clase Grafo consta de 2 atributos, una constante *"IMPORTEENTRADAS"* (double[]), este array contendrá los precios de las entradas para cada una de las horas (Punta, plana y simple) y el coste de las entradas tanto para las tarjetas simples como para las tarjetas con abono será el mismo aunque posteriormente se realizará el descuento. También consta de otra constante *"IMPORTESEGMENTOS"* (double[]), que contiene el coste de cada segmento para cada una de las horas.
Por último usamos un TreeMap<Vertice, TreeMap<Vertice, TablaPrecio>> listaPrecios que almacenará el importe de ir de un vértice a otro (de una estación a otra).

CONSTRUCTOR:
En el constructor recibimos el nombre del fichero de donde cogemos los datos como parámetro, en primer lugar ejecutamos floyd() para obtener las distancias mínimas entre vértices. Posteriormente para cada par de vértices distintos calculamos el precio del recorrido, reducimos el precio para las tarjetas con abono y añadimos dicha TablaPrecio a la lista si la listaPrecios contiene el vértice i.

```
public TablasPrecios(String nombreFichero) {
    super(nombreFichero);
    floyd();
    listaPrecios = new TreeMap<Vertice, TreeMap<Vertice, TablaPrecio>>();
    for (int i = 0; i < vertices.size(); i++) {
        for (int j = 0; j < vertices.size(); j++) {
            // calculo el camino desde el vertice i al vertice j
            if (i != j) {
                ArrayList<Vertice> lista = new ArrayList<Vertice>();
                lista.add(vertices.get(i));
                hacerCaminoFloyd(i, j, lista);
                lista.add(vertices.get(j));
                // System.out.println(lista);
                TablaPrecio tabla = calcularPrecio(lista);
                tabla.reducirAbono();
                if (!listaPrecios.containsKey(vertices.get(i)))
                    listaPrecios.put(vertices.get(i), new TreeMap<Vertice, TablaPrecio>());
                listaPrecios.get(vertices.get(i)).put(vertices.get(j), tabla);
            }
        }
    }
}
```

METODOS:

- *calcularPrecio(ArrayList<Vertice> lista):* Este método calcula el precio del recorrido según la lista de vértices por los que va pasando.
-

```
private TablaPrecio calcularPrecio(ArrayList<Vertice> lista) {
    TablaPrecio tabla = new TablaPrecio();
    tabla.incrementarTodos(IMPORTEENTRADAS);
    Vertice anterior = lista.get(0);
    for (int i = 1; i < lista.size(); i++) {
        Vertice actual = lista.get(i);
        // System.out.println("Antes: " + anterior + " " + actual);
        Arista arista = getArista(anterior, actual);
        // System.out.println(arista);
        tabla.incrementarTodos(IMPORTESEGMENTOS, arista.getLongitud());
        anterior = actual;
    }
    return tabla;
}
```

-

- *escribirTablas():* Muestra por pantalla y almacena en un string el cálculo de los precios de las tablas de todos los recorridos posibles. Dicho string se guardará posteriormente en un archivo de texto denominado *tablaPrecios.txt* dentro de la carpeta "tablaP3".

```
public String escribirTablas() {
    String resultado = "";
    System.out.printf   ("%70s", "TABLA PRECIOS");
    System.out.println("\n-----------------------------------------------------------------------------------------------------");
    //System.out.println("Importe en €");
    System.out.printf("%68s", "LEYENDA\n");
    System.out.printf("%1s", leyenda);
    System.out.println("\n-----------------------------------------------------------------------------------------------------");
    System.out.printf("%1s %88s", "Recorrido" ,nombres);
    System.out.println("");
    for (Entry<Vertice, TreeMap<Vertice, TablaPrecio>> e1 : listaPrecios.entrySet()) {
        for (Entry<Vertice, TablaPrecio> e2 : e1.getValue().entrySet()) {
            System.out.println(e1.getKey() + " -> " + e2.getKey() + "  \tImportes: " + "" + e2.getValue());
            resultado = resultado +  e1.getKey() + " -> " + e2.getKey() + "  \tImportes: " + "" + e2.getValue() +"\n";
        }
    }
    return resultado;

}
```

-

Lo que se mostrará por pantalla será lo siguiente:

```
<terminated> Principal3 [Java Application] C:\Program Files\Java\jre1.8.0_144\bin\javaw.exe (1 de ago. de 2018 18:44:32)
                                                    TABLA PRECIOS
-------------------------------------------------------------------------------------------------------------------------------------------

                                                    LEYENDA
Tarjeta simple en hora punta:    TS. Hora Punta
Tarjeta simple en hora plana:    TS. Hora Plana
Tarjeta simple en hora valle:    TS. Hora Valle
Tarjeta abono en hora simple:    TA. Hora Simple
Tarjeta abono en hora plana:     TA. Hora Plana
Tarjeta abono en hora valle:     TA. Hora Valle

-------------------------------------------------------------------------------------------------------------------------------------------
Recorrido                       TSH Punta       TSH Plana       TSH Valle       TAH Punta       TAH Plana       TAH Valle
1,2 -> 1,3      Importes:       0,22 €          0,17 €          0,11 €          0,20 €          0,15 €          0,10 €
1,2 -> 1,4      Importes:       0,34 €          0,26 €          0,17 €          0,31 €          0,23 €          0,15 €
1,2 -> 1,6      Importes:       0,58 €          0,44 €          0,29 €          0,52 €          0,40 €          0,26 €
1,2 -> 1,8      Importes:       0,82 €          0,62 €          0,41 €          0,74 €          0,56 €          0,37 €
1,2 -> 1,9      Importes:       0,94 €          0,71 €          0,47 €          0,85 €          0,64 €          0,42 €
1,2 -> 1,10     Importes:       1,06 €          0,80 €          0,53 €          0,95 €          0,72 €          0,48 €
1,2 -> 2,1      Importes:       0,34 €          0,26 €          0,17 €          0,31 €          0,23 €          0,15 €
1,2 -> 2,6      Importes:       0,70 €          0,53 €          0,35 €          0,63 €          0,48 €          0,32 €
1,2 -> 2,11     Importes:       1,30 €          0,98 €          0,65 €          1,17 €          0,88 €          0,59 €
1,2 -> 3,3      Importes:       0,46 €          0,35 €          0,23 €          0,41 €          0,32 €          0,21 €
1,2 -> 3,9      Importes:       1,18 €          0,89 €          0,59 €          1,06 €          0,80 €          0,53 €
1,2 -> 4,1      Importes:       0,58 €          0,44 €          0,29 €          0,52 €          0,40 €          0,26 €
1,2 -> 4,3      Importes:       0,58 €          0,44 €          0,29 €          0,52 €          0,40 €          0,26 €
1,2 -> 4,5      Importes:       0,82 €          0,62 €          0,41 €          0,74 €          0,56 €          0,37 €
1,2 -> 4,6      Importes:       0,94 €          0,71 €          0,47 €          0,85 €          0,64 €          0,42 €
1,2 -> 4,7      Importes:       1,06 €          0,80 €          0,53 €          0,95 €          0,72 €          0,48 €
1,2 -> 4,9      Importes:       1,30 €          0,98 €          0,65 €          1,17 €          0,88 €          0,59 €
1,2 -> 4,11     Importes:       1,54 €          1,16 €          0,77 €          1,39 €          1,04 €          0,69 €
1,3 -> 1,2      Importes:       0,22 €          0,17 €          0,11 €          0,20 €          0,15 €          0,10 €
```

CLASE GRAFO

A la clase Grafo mencionada en la práctica anterior le añadimos los métodos *floyd(), inicializarFloyd(), ejecutarFloyd() y hacerCaminoFloyd()* todos ellos utilizados para el cálculo de las tablas.

CONSTRUCTOR:

El constructor sigue siendo el mismo que la práctica anterior. A continuación nos concentraremos en los métodos anteriormente mencionados

METODOS:

- *floyd():* Método sencillo que combina *inicializarFloyd()* y *ejecutarFloyd()*

- *inicializarFloyd():* Inicializa el arrayList<Vertice> de vértices con todos los vértices del mapa. Posteriormente inicializamos la matriz de distancias y previo (anteriores) según el nº de vértices totales (mapa.size()). Si i = j (diagonal) inicializamos la matriz de distancias a 0 y la matriz de anteriores todos a null. Obtenemos la arista del vértice i y j, si esta no existe inicializamos la matriz de distancias a infinito y la matriz de anteriores a null. En caso de que exista dicha arista inicializamos distancias a la longitud de la arista y anteriores al vértice de i.

```
private void inicializarFloyd() {
    vertices = new ArrayList<Vertice>();
    int i = 0;
    for (Vertice v : mapa.keySet()) {
        vertices.add(v);
        i++;
    }
    distancias = new double[mapa.size()][mapa.size()];
    anteriores = new Vertice[mapa.size()][mapa.size()];
    for (i = 0; i < distancias.length; i++) {
        for (int j = 0; j < distancias.length; j++) {
            if (i == j) {
                distancias[i][j] = 0;
                anteriores[i][j] = null;
            } else {
                Arista a = getArista(vertices.get(i), vertices.get(j));
                if (a == null) {
                    distancias[i][j] = INFINITO;
                    anteriores[i][j] = null;
                } else {
                    distancias[i][j] = a.getLongitud();
                    anteriores[i][j] = vertices.get(i);
                }
            }
        }
    }
}
```

- *ejecutarFloyd():* Simplemente usamos el algoritmo de Floyd para ir obteniendo las distancias mínimas entre vértices.
-

```
private void ejecutarFloyd() {
    for (int k = 0; k < distancias.length; k++) {
        for (int i = 0; i < distancias.length; i++) {
            for (int j = 0; j < distancias.length; j++) {
                if (distancias[i][k] + distancias[k][j] < distancias[i][j]) {
                    distancias[i][j] = distancias[i][k] + distancias[k][j];
                    anteriores[i][j] = vertices.get(k);
                }
            }
        }
    }
}
```

- *hacerCaminoFloyd(int i, int j, ArrayList<Vertice> lista):* Obtiene los caminos de anteriores[].
-

```
public void hacerCaminoFloyd(int i, int j, ArrayList<Vertice> lista) {
    if (anteriores[i][j] == null || anteriores[i][j].equals(vertices.get(i))) {
        // lista.add(v);
    } else {
        int pos = vertices.indexOf(anteriores[i][j]);
        hacerCaminoFloyd(i, pos, lista);
        lista.add(anteriores[i][j]);
        hacerCaminoFloyd (pos, j, lista);
    }
}
}
```

PRINCIPIO DE OPTIMALIDAD DE BELLMAN APLICADO AL PROBLEMA

El principio de optimalidad de Bellman nos dice que toda sub-secuencia de una secuencia óptima de decisiones también debe ser óptima. Se verifica siempre que toda solución óptima a un problema este compuesta por soluciones óptimas de de sus subproblemas.

Si e_1, e_2, …, e_k, e_{k+1}, …, e_n es un camino mínimo de e_1 a e_n, entonces:

- e_1, e_2, …, e_k, es un camino mínimo de e_1 a e_k
- e_k, e_{k+1}, …, e_n, es un camino mínimo de e_k a e_n

Aplicación del principio:

Si k es un vértice intermedio de mayor índice en el camino óptimo de i a j, entonces el subcamino de i a k es un camino mínimo de i a k que, además, sólo pasa por vértices de índice menor que k. Con el subcamino de k a j ocurre lo mismo.

Definición recursiva de la solución

Para calcular los caminos mínimos pudiendo pasar por los k primeros vértices del grafo, usamos los caminos mínimos pasando por los k – 1 primeros.

Sea $D_k(i, j)$ la distancia del camino más corto de i a j, usando sólo los k primeros vértices del grafo como puntos intermedios, es decir, que no pasa por ningún vértice de índice mayor que k.

Expresión recursiva: $D_k(i, j) = min \{D_{k-1}(i, j), (D_{k-1}(i, k) + D_{k-1}(j, k))\}$

```
private void ejecutarFloyd() {
    for (int k = 0; k < distancias.length; k++) {
        for (int i = 0; i < distancias.length; i++) {
            for (int j = 0; j < distancias.length; j++) {
                if (distancias[i][k] + distancias[k][j] < distancias[i][j]) {
                    distancias[i][j] = distancias[i][k] + distancias[k][j];
                    anteriores[i][j] = vertices.get(k);
                }
            }
        }
    }
}
```

Caso base: $D_0(i, j)$ = *peso arista (i, j) [∞ si no existe la arista, 0 si i = j]*

```
for (i = 0; i < distancias.length; i++) {
    for (int j = 0; j < distancias.length; j++) {
        if (i == j) {
            distancias[i][j] = 0;
            anteriores[i][j] = null;
        } else {
            Arista a = getArista(vertices.get(i), vertices.get(j));
            if (a == null) {
                distancias[i][j] = INFINITO;
                anteriores[i][j] = null;
            } else {
                distancias[i][j] = a.getLongitud();
                anteriores[i][j] = vertices.get(i);
            }
        }
    }
}
```

En ambas expresiones se cumple: $1 \leq i \leq n$, $1 \leq j \leq n$, $1 \leq k \leq n$

Esquema algorítmico

```
Pre: g es un grafo dirigido, ponderado con pesos no negativos.}
método floyd(grafo g): entero array [1..n] [1..n], entero array [1..n] [1..n]
  array [1..n] [1..n] de entero d, p          // d es matriz distancias par vértices
  entero i, j, k                              // p es matriz de vértices intermedios
  para i=1 hasta n hacer                      // Inicialización de d y p
     para j=1 hasta n hacer
        Inicializar d [i] [j]   // A valor nulo para i=j, al peso de la arista (i, j)
                                // si existe, y a ∞ si no existe.
        Inicializar p[i] [j]    // A valor nulo para i=j y para cuando existe
                                // la arista (i, j), ya que en ese caso no habrá
                                // vértice intermedio, y a ∞ si no existe la arista.
     fpara
  fpara
  ……
método floyd(grafo g): entero array [1..n] [1..n], entero array [1..n] [1..n]
  ……
  para k=1 hasta n hacer          // Se va analizando el efecto de cada nodo
     para i=1 hasta n hacer
        para j=1 hasta n hacer
           si d[i][k] + d[k][j] < d[i][j] entonces
              d [i][j] = d[i][k] + d[k][j]
              p [i][j]  = k
           fsi
        fpara
     fpara
  fpara
  retorna d, p
fmétodo
```

{Post: d contiene la distancia de los caminos mínimos entre cada par de vértices de g; p contiene el vértice intermedio en el camino mínimo entre cada par de vértices de g, para poder reconstruir los vértices por los que pasa dicho camino mínimo.}

Se corresponde con *incializarFloyd()* y *ejecutarFloyd()* respectivamente realizado por nosotros en la práctica

Definición de la tabla PD

Hallar el camino mínimo desde el vértice 3 hasta 4 en el grafo con la siguiente matriz de distancias:

$$D = \begin{bmatrix} 0 & 3 & 5 & 1 & \infty & \infty \\ 3 & 0 & \infty & \infty & 9 & \infty \\ 5 & \infty & 0 & 7 & 7 & 1 \\ 1 & \infty & 7 & 0 & \infty & 4 \\ \infty & 9 & 7 & \infty & 0 & \infty \\ \infty & \infty & 1 & 4 & \infty & 0 \end{bmatrix}$$

1ª Iteración:

$d_{23} = \min(d_{23}, d_{21} + d_{13}) = 8$
$d_{24} = \min(d_{24}, d_{21} + d_{14}) = 4$
$d_{25} = \min(d_{25}, d_{21} + d_{15}) = 9$
$d_{26} = \min(d_{26}, d_{21} + d_{16}) = \infty$
$d_{32} = \min(d_{32}, d_{31} + d_{12}) = 8$
$d_{34} = \min(d_{34}, d_{31} + d_{14}) = 6$
$d_{35} = \min(d_{35}, d_{31} + d_{15}) = 7$
$d_{36} = \min(d_{36}, d_{31} + d_{16}) = 1$
$d_{45} = \min(d_{45}, d_{41} + d_{15}) = \infty$
$d_{46} = \min(d_{46}, d_{41} + d_{16}) = 4$
$d_{56} = \min(d_{58}, d_{51} + d_{16}) = \infty$

La matriz de distancia después de esta iteración es:

$$W_1 = \begin{bmatrix} 0 & 3 & 5 & 1 & \infty & \infty \\ 3 & 0 & 8 & 4 & 9 & \infty \\ 5 & 8 & 0 & 6 & 7 & 1 \\ 1 & 4 & 6 & 0 & \infty & 4 \\ \infty & 9 & 7 & \infty & 0 & \infty \\ \infty & \infty & 1 & 4 & \infty & 0 \end{bmatrix}$$

2ª Iteración:

$d_{13} = \min(d_{13}, d_{12} + d_{23}) = 5$
$d_{14} = \min(d_{14}, d_{12} + d_{24}) = 1$
$d_{15} = \min(d_{15}, d_{12} + d_{25}) = 12$
$d_{16} = \min(d_{16}, d_{12} + d_{26}) = \infty$
$d_{34} = \min(d_{34}, d_{32} + d_{24}) = 6$
$d_{35} = \min(d_{35}, d_{32} + d_{25}) = 7$
$d_{36} = \min(d_{36}, d_{32} + d_{26}) = 1$
$d_{45} = \min(d_{45}, d_{42} + d_{25}) = 13$
$d_{46} = \min(d_{46}, d_{42} + d_{26}) = 4$
$d_{56} = \min(d_{56}, d_{52} + d_{26}) = \infty$

La matriz de distancia después de esta iteración es:

$$W_2 = \begin{bmatrix} 0 & 3 & 5 & 1 & 12 & \infty \\ 3 & 0 & 8 & 4 & 9 & \infty \\ 5 & 8 & 0 & 6 & 7 & 1 \\ 1 & 4 & 6 & 0 & 13 & 4 \\ 12 & 9 & 7 & 13 & 0 & \infty \\ \infty & \infty & 1 & 4 & \infty & 0 \end{bmatrix}$$

3ª Iteración:

$d_{12} = \min(d_{12}, d_{13} + d_{32}) = 3$
$d_{14} = \min(d_{14}, d_{13} + d_{34}) = 1$
$d_{15} = \min(d_{15}, d_{13} + d_{35}) = 12$
$d_{16} = \min(d_{16}, d_{13} + d_{36}) = 6$
$d_{24} = \min(d_{24}, d_{23} + d_{34}) = 4$
$d_{25} = \min(d_{25}, d_{23} + d_{35}) = 9$
$d_{26} = \min(d_{26}, d_{23} + d_{36}) = 9$
$d_{45} = \min(d_{45}, d_{43} + d_{35}) = 13$
$d_{46} = \min(d_{46}, d_{43} + d_{36}) = 4$
$d_{56} = \min(d_{56}, d_{53} + d_{36}) = 8$

La matriz de distancia después de esta iteración es:

$$W_3 = \begin{bmatrix} 0 & 3 & 5 & 1 & 12 & 6 \\ 3 & 0 & 8 & 4 & 9 & 9 \\ 5 & 8 & 0 & 6 & 7 & 1 \\ 1 & 4 & 6 & 0 & 13 & 4 \\ 12 & 9 & 7 & 13 & 0 & 8 \\ 6 & 9 & 1 & 4 & 8 & 0 \end{bmatrix}$$

4ª Iteración:

$d_{12} = \min(d_{12}, d_{14} + d_{42}) = 3$
$d_{13} = \min(d_{13}, d_{14} + d_{43}) = 5$
$d_{15} = \min(d_{15}, d_{14} + d_{45}) = 12$
$d_{16} = \min(d_{16}, d_{14} + d_{46}) = 5$
$d_{23} = \min(d_{23}, d_{24} + d_{43}) = 8$
$d_{25} = \min(d_{25}, d_{24} + d_{45}) = 9$
$d_{26} = \min(d_{26}, d_{24} + d_{46}) = 8$
$d_{35} = \min(d_{35}, d_{34} + d_{45}) = 7$
$d_{36} = \min(d_{36}, d_{34} + d_{46}) = 1$
$d_{56} = \min(d_{56}, d_{54} + d_{46}) = 8$

La matriz de distancia después de esta iteración es:

$$W_4 = \begin{bmatrix} 0 & 3 & 5 & 1 & 12 & 5 \\ 3 & 0 & 8 & 4 & 9 & 8 \\ 5 & 8 & 0 & 6 & 7 & 1 \\ 1 & 4 & 6 & 0 & 13 & 4 \\ 12 & 9 & 7 & 13 & 0 & 8 \\ 5 & 8 & 1 & 4 & 8 & 0 \end{bmatrix}$$

5ª Iteración:

$d_{12} = \min(d_{12}, d_{15} + d_{52}) = 3$
$d_{13} = \min(d_{13}, d_{15} + d_{53}) = 5$
$d_{14} = \min(d_{14}, d_{15} + d_{54}) = 1$
$d_{16} = \min(d_{16}, d_{15} + d_{56}) = 5$
$d_{23} = \min(d_{23}, d_{25} + d_{53}) = 8$
$d_{24} = \min(d_{24}, d_{25} + d_{54}) = 4$
$d_{26} = \min(d_{26}, d_{25} + d_{56}) = 8$
$d_{34} = \min(d_{34}, d_{35} + d_{54}) = 6$
$d_{36} = \min(d_{36}, d_{35} + d_{56}) = 1$
$d_{46} = \min(d_{46}, d_{45} + d_{56}) = 4$

La matriz de distancia después de esta iteración es:

$$W_5 = W_4 \begin{bmatrix} 0 & 3 & 5 & 1 & 12 & 5 \\ 3 & 0 & 8 & 4 & 9 & 8 \\ 5 & 8 & 0 & 6 & 7 & 1 \\ 1 & 4 & 6 & 0 & 13 & 4 \\ 12 & 9 & 7 & 13 & 0 & 8 \\ 5 & 8 & 1 & 4 & 8 & 0 \end{bmatrix}$$

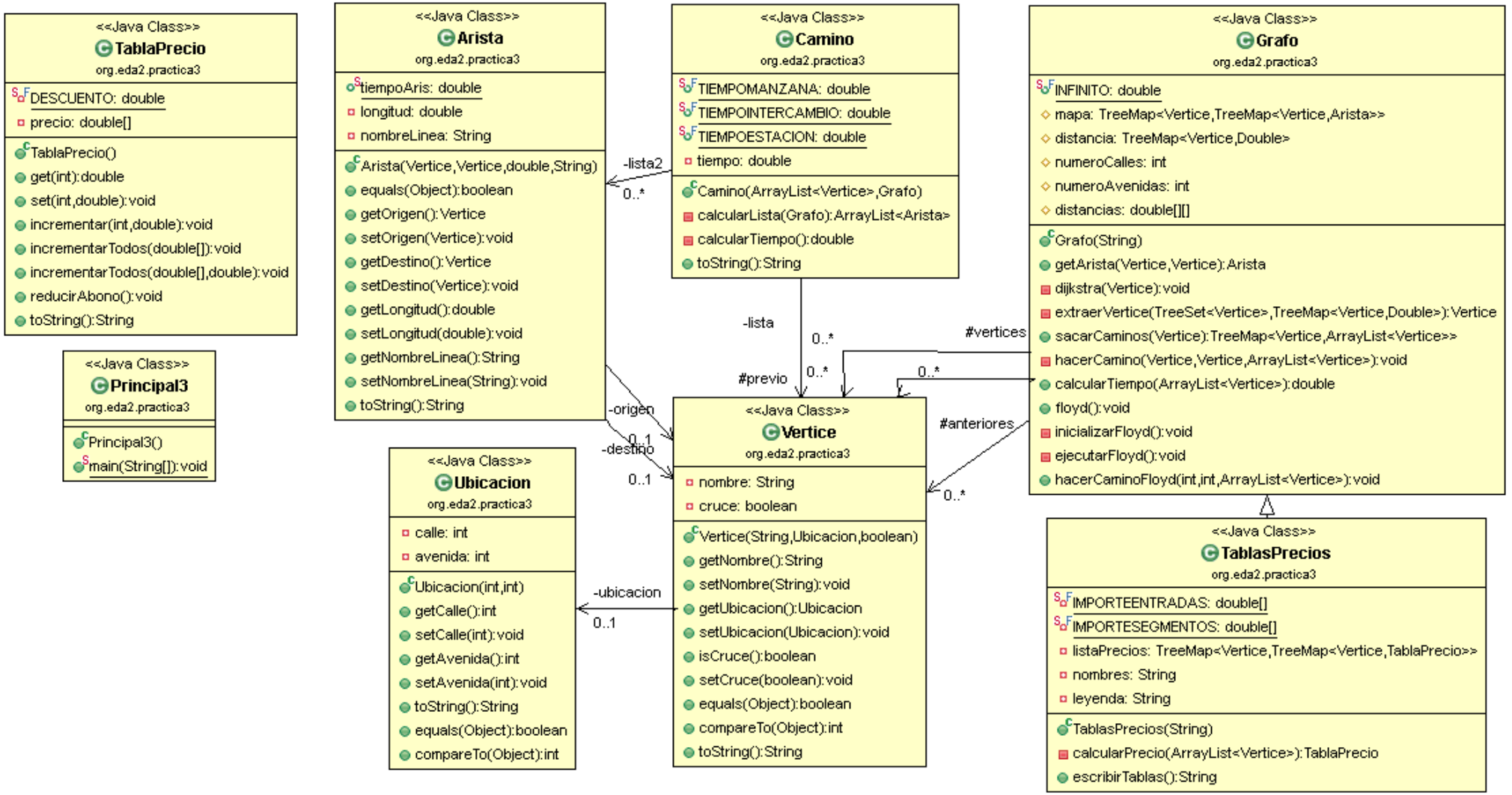
6ª Iteración:

$d_{12} = \min(d_{12}, d_{16} + d_{62}) = 3$
$d_{13} = \min(d_{13}, d_{16} + d_{63}) = 5$
$d_{14} = \min(d_{14}, d_{16} + d_{64}) = 1$
$d_{15} = \min(d_{15}, d_{16} + d_{65}) = 12$
$d_{23} = \min(d_{23}, d_{26} + d_{63}) = 8$
$d_{24} = \min(d_{24}, d_{26} + d_{64}) = 4$
$d_{25} = \min(d_{25}, d_{26} + d_{65}) = 9$
$d_{34} = \min(d_{34}, d_{36} + d_{64}) = 5$
$d_{35} = \min(d_{35}, d_{36} + d_{65}) = 7$
$d_{45} = \min(d_{45}, d_{46} + d_{65}) = 12$

La matriz de distancia después de esta iteración es:

$$W_6 = \begin{bmatrix} 0 & 3 & 5 & 1 & 12 & 5 \\ 3 & 0 & 8 & 4 & 9 & 8 \\ 5 & 8 & 0 & 5 & 7 & 1 \\ 1 & 4 & 5 & 0 & 12 & 4 \\ 12 & 9 & 7 & 12 & 0 & 8 \\ 5 & 8 & 1 & 4 & 8 & 0 \end{bmatrix}$$

DIAGRAMA DE CLASES



ANÁLISIS TEÓRICO DEL TIEMPO DEL ALGORITMO DE PD

El tiempo de ejecución dependerá de las características concretas del problema que se vaya a resolver. En general, será de la forma:

Tamaño de la tabla x Tiempo de rellenar cada elemento de la tabla

Un aspecto importante, ya que la programación dinámica se basa en el uso de tablas, donde se almacenan los resultados parciales, es que hay que tener en cuenta la memoria que puede llegar a ocupar la tabla.

Como podemos observar a continuación el orden de complejidad se debe al triple bucle anidado, en cuyo interior hay tan solo operaciones constantes.

```
private void ejecutarFloyd() {
    for (int k = 0; k < distancias.length; k++) {
        for (int i = 0; i < distancias.length; i++) {
            for (int j = 0; j < distancias.length; j++) {
                if (distancias[i][k] + distancias[k][j] < distancias[i][j]) {
                    distancias[i][j] = distancias[i][k] + distancias[k][j];
                    anteriores[i][j] = vertices.get(k);
                }
            }
        }
    }
}
```

Por lo que tenemos:

$$t(n) \in \theta(n^3)$$

En cuanto al espacio podríamos destacar que Floyd exige $\theta(n^2)$ mientras que Dijkstra precisa de $\theta(n)$.

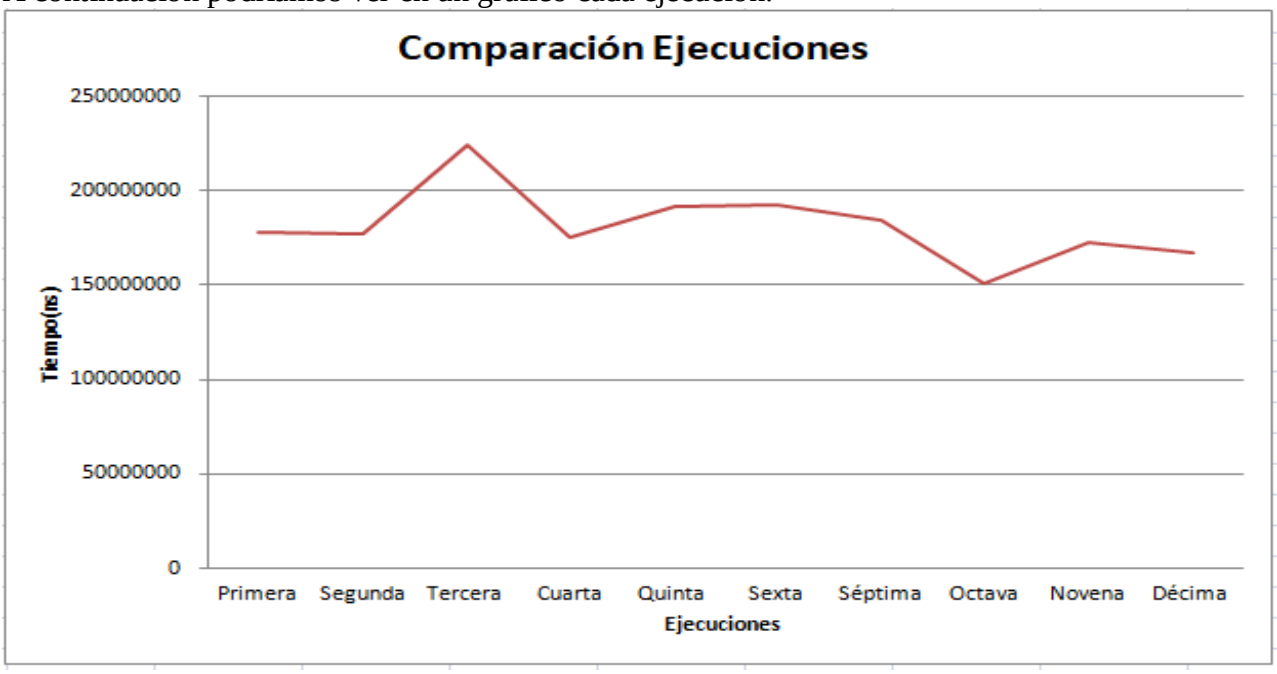
ANÁLISIS DE TIEMPOS DE EJECUCIÓN DEL ALGORITMO PD

A continuación ejecutamos el algoritmo de cálculo de las tablas unas diez veces guardando el tiempo que tarda cada ejecución y realizando una media de las diez ejecuciones:

Ejecución	Tiempo(ns)
Primera	177.685.035
Segunda	177.072.112
Tercera	224.006.023
Cuarta	174.828.562
Quinta	191.333.945
Sexta	192.284.323
Séptima	184.591.391
Octava	150.423.339
Novena	172.947.101
Décima	167.360.190
Media	181.253.202

Como podemos observar el tiempo que tarda el algoritmo en calcular la tabla de precios ronda entre los 0'15 segundos (mínimo valor) a los 0'22 segundos (máximo valor) siendo la media de las diez ejecuciones de 0'18 segundos.

A continuación podríamos ver en un gráfico cada ejecución:



Como podemos observar el tiempo en cada ejecución es semejante, con pequeñas diferencias de tiempos entre una y otra.

PRECONDICIONES Y POSTCONDICIONES

En cuanto a las precondiciones y postcondiciones, muchas de ellas coinciden con la práctica anterior ya que partimos de ella para continuar con el siguiente problema. Debemos fijarnos principalmente en aquellas clases que hemos añadido como "TablaPrecio" o "TablaPrecios" o en aquellas clases que hemos modificado como "Grafo" a la cual le añadimos el algoritmo de Floyd.

- Precondiciones:

  - *Grafo*:
    - Hacemos uso de tres nuevas estructuras de datos, un ArrayList<Vertice> vertices, un double[][] distancias y Vertice[][] anteriores. Todas ellas estructuras necesarias en el algoritmo de Floyd.
    - Asignamos un tiempo de 4 minutos para recorrer una manzana, de 7 minutos para los cambios de línea y de 5 minutos para el tiempo en llegar a una estación, importante a la hora de calcular el tiempo en ir de una estación (origen) a otra (destino).
    - Por comodidad y convenio común dividimos el algoritmo de Floyd en dos partes: *"inicializarFloyd"* y *"ejecutarFloyd"*.
    - Importante definir un valor máximo que se usa para rellenar las matriz de distancias en caso de que no exista arista entre vértices ya sea un *public static final double INFINITO = 1e20* o *doublé.MAX_VALUE*.

  - *TablaPrecio:*
    - Establecemos el *"DESCUENTO"* a una constante en este caso de un 0.1 (10%).
    - Usamos un *"double [] precio"* para almacenar los precios de cada una de las horas y de cada tarjeta. Como tenemos tres horas diferentes y dos tarjetas posibles tendríamos un total de seis precios distintos.
    - Debemos tener implementado un método toString() para mostrar los precios con un cierto formato.

```
public String toString () {
    String cadena = "";
    for (int i=0; i<precio.length; i++) {
        cadena = cadena + "\t\t" + String.format("%.2f", precio[i]);
        if (i < precio.length-1)
            cadena = cadena +" € ";
    }
    return cadena + " €";
}
```

- *TablaPrecios:*
  - Definimos una constante *"double[] IMPORTEENTRADAS"* con el importe de las entradas en este caso sería de la siguiente forma: *IMPORTEENTRADAS = { 0.10, 0.08, 0.05, 0.10, 0.08, 0.05 };*
    La tres primeras posiciones almacenarían los importes de la tarjeta simple en las distintas horas (Punta, Plana y Valle) y las tres últimas posiciones los importes de la tarjeta con abono en sus respectivas horas también.
  - De la misma forma definimos otra constante *"double[] IMPORTESEGMENTOS"* para el importe de cada segmento.

➢ Postcondiciones:

- *Grafo*:
  - La función "calcularTiempo" deberá devolver un doublé.
  - Se debe inicializarFloyd siguiendo la estructura del algoritmo visto en las prácticas. Es decir si es el mismo vértice la diagonal de la matriz de distancias debe ser 0 y la de anteriores null.

```
if (i == j) {
    distancias[i][j] = 0;
    anteriores[i][j] = null;
```

Si no es el mismo vértice obtenemos la arista de uno a otro si esta es nula, la matriz de distancias es INFINITO y la de anteriores null también.

```
Arista a = getArista(vertices.get(i), vertices.get(j));
if (a == null) {
    distancias[i][j] = INFINITO;
    anteriores[i][j] = null;
```

Y si no la matriz de distancias debe ser igual a la longitud de dicha arista y la matriz de anteriores guarda uno de los vértices que tiene la arista

```
distancias[i][j] = a.getLongitud();
anteriores[i][j] = vertices.get(i);
```

- *TablaPrecio:*
  - La función "reducirAbono" debe reducir el precio de las tarjetas del tipo abono en un 10%. Es decir debe establecer las tres últimas posiciones del array de precios a un doublé siendo este el 0'9 del valor total.
  - La función "incrementarTodos" debe aumentar el precio total añadiéndole el precio de la entrada.
  - Por otro lado de existir otra función "incrementarTodos" que aumente el precio total por el precio del segmento*nº de segmentos.

- *TablaPrecios:*
  - Se debe imprimir la tabla de precios siguiendo un formato deseado simulando una aplicación real:

TABLA PRECIOS

LEYENDA

Tarjeta simple en hora punta: TS. Hora Punta
Tarjeta simple en hora plana: TS. Hora Plana
Tarjeta simple en hora valle: TS. Hora Valle
Tarjeta abono en hora simple: TA. Hora Simple
Tarjeta abono en hora plana: TA. Hora Plana
Tarjeta abono en hora valle: TA. Hora Valle

Recorrido		TSH Punta	TSH Plana	TSH Valle	TAH Punta	TAH Plana	TAH Valle
1,2 -> 1,3	Importes:	0,22 €	0,17 €	0,11 €	0,20 €	0,15 €	0,10 €
1,2 -> 1,4	Importes:	0,34 €	0,26 €	0,17 €	0,31 €	0,23 €	0,15 €
1,2 -> 1,6	Importes:	0,58 €	0,44 €	0,29 €	0,52 €	0,40 €	0,26 €
1,2 -> 1,8	Importes:	0,82 €	0,62 €	0,41 €	0,74 €	0,56 €	0,37 €
1,2 -> 1,9	Importes:	0,94 €	0,71 €	0,47 €	0,85 €	0,64 €	0,42 €
1,2 -> 1,10	Importes:	1,06 €	0,80 €	0,53 €	0,95 €	0,72 €	0,48 €
1,2 -> 2,1	Importes:	0,34 €	0,26 €	0,17 €	0,31 €	0,23 €	0,15 €
1,2 -> 2,6	Importes:	0,70 €	0,53 €	0,35 €	0,63 €	0,48 €	0,32 €
1,2 -> 2,11	Importes:	1,30 €	0,98 €	0,65 €	1,17 €	0,88 €	0,59 €
1,2 -> 3,3	Importes:	0,46 €	0,35 €	0,23 €	0,41 €	0,32 €	0,21 €
1,2 -> 3,9	Importes:	1,18 €	0,89 €	0,59 €	1,06 €	0,80 €	0,53 €
1,2 -> 4,1	Importes:	0,58 €	0,44 €	0,29 €	0,52 €	0,40 €	0,26 €
1,2 -> 4,3	Importes:	0,58 €	0,44 €	0,29 €	0,52 €	0,40 €	0,26 €
1,2 -> 4,5	Importes:	0,82 €	0,62 €	0,41 €	0,74 €	0,56 €	0,37 €
1,2 -> 4,6	Importes:	0,94 €	0,71 €	0,47 €	0,85 €	0,64 €	0,42 €
1,2 -> 4,7	Importes:	1,06 €	0,80 €	0,53 €	0,95 €	0,72 €	0,48 €
1,2 -> 4,9	Importes:	1,30 €	0,98 €	0,65 €	1,17 €	0,88 €	0,59 €
1,2 -> 4,11	Importes:	1,54 €	1,16 €	0,77 €	1,39 €	1,04 €	0,69 €
1,3 -> 1,2	Importes:	0,22 €	0,17 €	0,11 €	0,20 €	0,15 €	0,10 €
1,3 -> 1,4	Importes:	0,22 €	0,17 €	0,11 €	0,20 €	0,15 €	0,10 €
1,3 -> 1,6	Importes:	0,46 €	0,35 €	0,23 €	0,41 €	0,32 €	0,21 €
1,3 -> 1,8	Importes:	0,70 €	0,53 €	0,35 €	0,63 €	0,48 €	0,32 €
1,3 -> 1,9	Importes:	0,82 €	0,62 €	0,41 €	0,74 €	0,56 €	0,37 €
1,3 -> 1,10	Importes:	0,94 €	0,71 €	0,47 €	0,85 €	0,64 €	0,42 €
1,3 -> 2,1	Importes:	0,46 €	0,35 €	0,23 €	0,41 €	0,32 €	0,21 €

  - Se debe almacenar dicha tabla creada en un archivo de texto denominado en este caso *"tablaPrecios.txt"*.