PROGRAMACIÓN Y ESTRUCTURAS DE DATOS AVANZADAS

PRUEBA DE EVALUACIÓN CONTINUA - 2

Andrea Garea González
48117937M
agarea10@alumno.uned.es
andreagareagonzalez@gmail.com

Contenido

Presentación y breve explicación de la prueba

En esta práctica se ha utilizado el **algoritmo de ramificación y poda (Branch and bound)** para la resolución del **problema de asignación de pasteleros**.

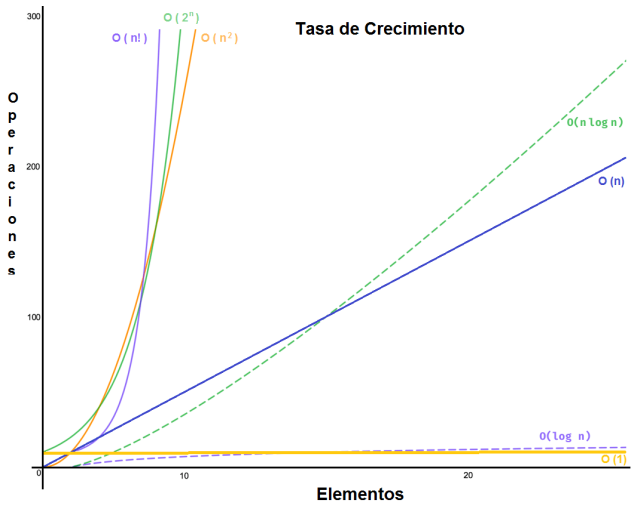
El algoritmo se utiliza para asignar pasteleros a de forma óptima para obtener el menor coste total posible.

Funciona de la siguiente manera:

- *Se inicializa un montículo de mínimos* para almacenar cada nodo y obtener en la cima el de menor coste.
- *Se agrega el nodo inicial al montículo* con una estimación óptima y una estimación pésima iniciales.
- *Mientras el montículo no esté vacío o el coste del nodo superior sea inferior a la cota*, se extrae el nodo con la estimación óptima mínima y se procesa.
- *Si el nodo es una solución completa* (es decir, todos los pedidos han sido asignados a pasteleros), *se actualiza la solución óptima* actual si el costo de la solución es menor que el costo de la solución óptima actual.
- *Si el nodo no es una solución completa*, se generan los nodos hijos *expandiendo* la búsqueda a todas las posibles asignaciones de pasteleros para el pedido siguiente. Se calculan las estimaciones óptima y pésima para cada nodo hijo y se agrega el nodo hijo al montículo si su estimación pésima es menor que el costo de la solución óptima actual (poda).
- La ejecución termina cuando el montículo está vacío y se devuelve la solución óptima encontrada.

Cuestiones teóricas sobre la prueba

Indica y razona sobre el coste temporal y espacial del algoritmo

1. 

2. $$O(g(x)) = \left\{ \begin{array}{c} f(x) : \text{existen } x_0, c > 0 \text{ tales que} \\ \forall x \geq x_0 > 0 : 0 \leq |f(x)| \leq c|g(x)| \end{array} \right\}$$

Para calcular el coste del algoritmo se ha usado la **cota superior asintótica** (O grande). La característica principal de este análisis es que se calcularán los costes en relación al peor caso.

Se determina que el coste temporal para este algoritmo es **$O(n^a)$**. N representaría la longitud de la asignación entre pasteleros y pedido mientras que A serían todas las posibles ramas que se podrían generar. Esta asignación delata que el algoritmo es bastante ineficiente y puede necesitar mucha memoria en comparación a otros como el voraz.

El **coste espacial** del algoritmo de ramificación y poda, o sea, el espacio de memoria ocupado, *dependerá del número de ramas que cree*. El número de ramas es dependiente de la longitud de asignación del pedido, por lo que cuantas más asignaciones de pasteleros haya mas elevado será el coste espacial.

Explica qué otros esquemas pueden resolver el problema y razona sobre su idoneidad

En concreto hay un esquema que podría servir como sustituto, el algoritmo voraz. Este algoritmo podría reducir los costes temporales y espaciales de este problema, ya que este algoritmo consiste en seleccionar la primera mínima asignación en cuanto a los pasteleros. El problema de este algoritmo es que no tiene en cuenta los largos plazos; la solución mas optima siempre la proporcionará el algoritmo de ramificación y poda mientras que el voraz dará la más rápida (que no significa que sea la mejor).

Por otra parte, este algoritmo no puede ser sustituido por divide y vencerás ya que el ejercicio no tiene el enfoque que requiere el algoritmo, ya que, si pudiésemos dividir el problema en subproblemas, podría afectar a la asignación de pasteleros a los pedidos restantes.

Ejemplo de ejecución con distintos tamaños del problema

NOTA: **Las ejecuciones para demostrar la optimalidad de la prueba se han generado mediante otro pequeño script a mayores.**

```
import java.io.*;

public class GeneratorClass {

	public static BufferedReader reader = new BufferedReader(new
InputStreamReader(System.in));

	public static void main(String[] args) throws NumberFormatException,
IOException {
		int pointer = 0, pointer2 = 0, pointer3 = 0;
		BufferedWriter bw = new BufferedWriter(new
OutputStreamWriter(new FileOutputStream("autogeneratedData.txt"), "utf-
8"));

		System.out.println("Numero de pasteleros:");
		int totalPasteleros = Integer.parseInt(reader.readLine());
		bw.write(totalPasteleros + "\n");
		System.out.println("Numero pasteles:");
		int totalPasteles = Integer.parseInt(reader.readLine());
		bw.write(totalPasteles + "\n");
		System.out.println("Longitud pedido:");
		int longPedido = Integer.parseInt(reader.readLine());

		int[] pedido = new int[longPedido];
		while (pointer <  longPedido) {
			int thisPedido = (int)
Math.floor(Math.random()*(totalPasteles-2+1)+1);
			pedido[pointer] = thisPedido;
			pointer ++;
		}
		bw.write(String.join("-",
Arrays.stream(pedido).mapToObj(String::valueOf).toArray(String[]::new))+
"\n");

		while (pointer2 <  totalPasteleros) {
			int[] localCost = new int[totalPasteles];
			while(pointer3 < totalPasteles) {
				int thisCost = (int)
Math.floor(Math.random()*(totalPasteles-1+1)+1);
				localCost[pointer3] = thisCost;
				pointer3 ++;
			}
			bw.write(String.join(" ",
Arrays.stream(localCost).mapToObj(String::valueOf).toArray(String[]::new))+
"\n");
			pointer3=0;
			pointer2++;
		}

		bw.close();
	}
}
```

He tenido diversos problemas a la hora de realizar esta práctica y tengo que lamentar no tenerla del todo completa. Sin embargo, la implementación que he logrado es capaz de recrear cómo de costoso temporal y espacial es este enunciado mediante ramificación y poda.

Para el correcto funcionamiento de la prueba, he tenido que ajustar los ficheros de entrada para que línea del pedido comience a contar en 0 (es decir, si tenemos 3 diferentes tipos de pasteles, serán el 0, el 1 y el 2).

PRUEBA 1: 9ms

Entrada:

```
5
3
0-0-2-1-0
2 5 3
5 3 2
6 4 9
6 3 8
7 5 8
```

```
>java -jar testPEC2preda.jar prueba1.txt
============================
```

Salida:

```
==========================
** Coste total: 20

** Asignacion de pasteleros

==========================

** [Pedido 1] | Pastelero: 1

** [Pedido 2] | Pastelero: 1

** [Pedido 3] | Pastelero: 2

** [Pedido 4] | Pastelero: 2

** [Pedido 5] | Pastelero: 1

==========================
```

PRUEBA 2: (63ms)

Para estas pruebas, dejo adjuntos los resultados en el propio .rar

Entrada: autogeneratedData10.txt

```
 -jar testPEC2preda.jar autogeneratedData10.txt ficheroSalida10.txt
** Resultados en el archivo indicado.
Tiempo de ejecucion: 63 ms
```

Salida: ficheroSalida10.txt

PRUEBA 3: Excesivo

Entrada: autogeneratedData100.txt

```
java -jar testPEC2preda.jar autogeneratedData100.txt ficheroSalida100.txt
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
        at base.Pasteleria.AsignaPasteleros(Pasteleria.java:99)
        at base.Main.main(Main.java:242)
```

Fuera de memoria

PRUEBA 4: No introducir ficheros de entrada y salida

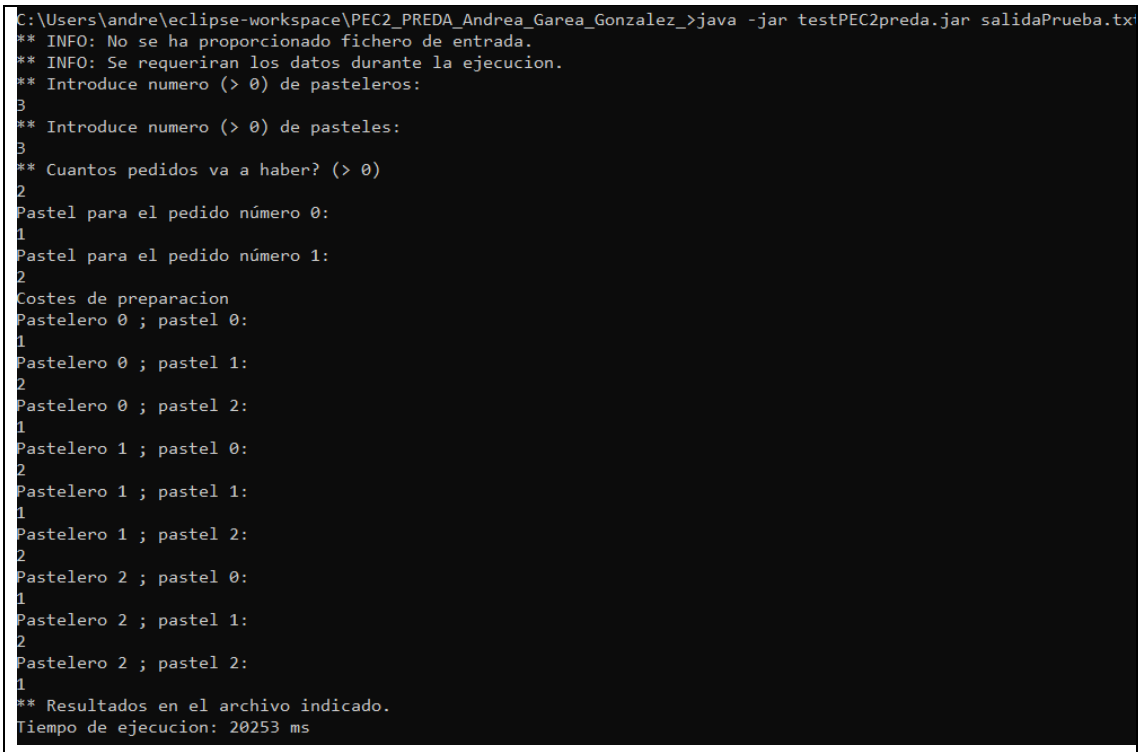
Resultado: no ejecutable

```
C:\Users\andre\eclipse-workspace\PEC2_PREDA_Andrea_Garea_Gonzalez_>java -jar testPEC2preda.jar

C:\Users\andre\eclipse-workspace\PEC2_PREDA_Andrea_Garea_Gonzalez_>
```

PRUEBA 5: No introducir fichero de entrada, pero si de salida

Resultado: Necesitamos incluir los datos por consola



PRUEBA 6: Incluir traza (-t) y ayuda (-h) sin fichero de salida

Entrada (enunciado de la práctica):

```
3
2
0-0-0
1 1
2 2
1 1
```

Resultado: muestra la ayuda, lo que va haciendo y el resultado por pantalla.
C:\Users\andre\eclipse-workspace\PEC2_PREDA_Andrea_Garea_Gonzalez_>java -jar testPEC2preda.jar autogeneratedData3.txt -t -h

// MAIN //
SINTAXIS: pasteleria [fichero entrada] [fichero salida] [-t][-h]
-t Traza el algoritmo
-h Muestra esta ayuda
[fichero entrada] Nombre del fichero de entrada
[fichero salida] Nombre del fichero de salida
// Leyendo datos de entrada...
// Preparando fichero de salida...
** INFO: Fichero de salida no proporcionado, el resultado se proporcionara por pantalla.
// Se obtiene numero de pasteleros: 3
// Se obtiene el numero diferentes pasteles: 2
// Leemos pedido: 0 y lo añadimos al array de pedido
// Leemos pedido: 0 y lo añadimos al array de pedido
// Leemos pedido: 0 y lo añadimos al array de pedido

```
    // Nuestro pedido es: [0, 0, 0]
    // Añadimos coste: 1 al pastelero 0 para el pastel 0
    // Añadimos coste: 1 al pastelero 0 para el pastel 1
    // Añadimos coste: 2 al pastelero 1 para el pastel 0
    // Añadimos coste: 2 al pastelero 1 para el pastel 1
    // Añadimos coste: 1 al pastelero 2 para el pastel 0
    // Añadimos coste: 1 al pastelero 2 para el pastel 1
INICIO METODO ASIGNAPASTELEROS
// Se crea un monticulo de minimos y se asigna coste 0
// Se crea un array de booleanos con los pasteleros asignados y se establece todo su
contenido a false
// Se crea el primer nodo
// BUSQUEDA ESTIMACION OPTIMA
    // Menor coste pedido 1 (partida) pastelero 0: 1
    // Encontrado menor coste pedido 1 (FINAL) pastelero 0: 1
    // Menor coste pedido 2 (partida) pastelero 0: 1
    // Encontrado menor coste pedido 2 (FINAL) pastelero 0: 1
    // Estimacion optima de este nodo: 2.0
// Se calcula la estimacion optima del nodo incial
// Se agrega el nodo al monticulo
// BUSQUEDA ESTIMACION PESIMA
    // Encontrado menor coste pedido 1 (FINAL): 2
    // Encontrado menor coste pedido 2 (FINAL): 2
    // Estimacion pesima de este nodo: 4.0
// Se obtiene la cota (coste pesimista) y se crea el primer nodo hijo con ella
// Mientras que la cola no este vacio o el siguiente elemento tenga una estimacion optima
inferior a la cota
// Obtiene el nodo de la cima del monticulo (el minimo)
// Clona el nodo inicial
// El pastelero no esta asignado, se asigna el pastelero al pedido actual, se asigna y se
actualiza el coste total del pastelero i en para el pastel de la posicion del pedido k
// Todavia no se han asignado todos los pasteleros, se crea un nuevo nodo y se actualizan
estimaciones
// BUSQUEDA ESTIMACION OPTIMA
    // Menor coste pedido 2 (partida) pastelero 0: 1
    // Encontrado menor coste pedido 2 (FINAL) pastelero 0: 1
    // Estimacion optima de este nodo: 2.0
// BUSQUEDA ESTIMACION PESIMA
    // Encontrado menor coste pedido 2 (FINAL): 2
    // Estimacion pesima de este nodo: 3.0
// Se desmarca el pastelero
// El pastelero no esta asignado, se asigna el pastelero al pedido actual, se asigna y se
actualiza el coste total del pastelero i en para el pastel de la posicion del pedido k
// Todavia no se han asignado todos los pasteleros, se crea un nuevo nodo y se actualizan
estimaciones
// BUSQUEDA ESTIMACION OPTIMA
    // Menor coste pedido 2 (partida) pastelero 0: 1
    // Encontrado menor coste pedido 2 (FINAL) pastelero 0: 1
    // Estimacion optima de este nodo: 3.0
// BUSQUEDA ESTIMACION PESIMA
    // Encontrado menor coste pedido 2 (FINAL): 2
    // Estimacion pesima de este nodo: 4.0
```

```
// Se desmarca el pastelero
// El pastelero no esta asignado, se asigna el pastelero al pedido actual, se asigna y se actualiza el coste total del pastelero i en para el pastel de la posicion del pedido k
// Todavia no se han asignado todos los pasteleros, se crea un nuevo nodo y se actualizan estimaciones
// BUSQUEDA ESTIMACION OPTIMA
    // Menor coste pedido 2 (partida) pastelero 0: 1
    // Encontrado menor coste pedido 2 (FINAL) pastelero 0: 1
    // Estimacion optima de este nodo: 2.0
// BUSQUEDA ESTIMACION PESIMA
    // Encontrado menor coste pedido 2 (FINAL): 2
    // Estimacion pesima de este nodo: 3.0
// Se desmarca el pastelero
// Obtiene el nodo de la cima del monticulo (el minimo)
// Clona el nodo inicial
// El pastelero no esta asignado, se asigna el pastelero al pedido actual, se asigna y se actualiza el coste total del pastelero i en para el pastel de la posicion del pedido k
// Ya se asignaron todos los pasteleros
// El coste total es menor o igual a la cota de pesimismo, se actualiza el coste total y la cota
// Se desmarca el pastelero
// El pastelero no esta asignado, se asigna el pastelero al pedido actual, se asigna y se actualiza el coste total del pastelero i en para el pastel de la posicion del pedido k
// Ya se asignaron todos los pasteleros
// Se desmarca el pastelero
// El pastelero no esta asignado, se asigna el pastelero al pedido actual, se asigna y se actualiza el coste total del pastelero i en para el pastel de la posicion del pedido k
// Ya se asignaron todos los pasteleros
// El coste total es menor o igual a la cota de pesimismo, se actualiza el coste total y la cota
// Se desmarca el pastelero
// Obtiene el nodo de la cima del monticulo (el minimo)
// Clona el nodo inicial
// El pastelero no esta asignado, se asigna el pastelero al pedido actual, se asigna y se actualiza el coste total del pastelero i en para el pastel de la posicion del pedido k
// Ya se asignaron todos los pasteleros
// El coste total es menor o igual a la cota de pesimismo, se actualiza el coste total y la cota
// Se desmarca el pastelero
// El pastelero no esta asignado, se asigna el pastelero al pedido actual, se asigna y se actualiza el coste total del pastelero i en para el pastel de la posicion del pedido k
// Ya se asignaron todos los pasteleros
// Se desmarca el pastelero
// El pastelero no esta asignado, se asigna el pastelero al pedido actual, se asigna y se actualiza el coste total del pastelero i en para el pastel de la posicion del pedido k
// Ya se asignaron todos los pasteleros
// El coste total es menor o igual a la cota de pesimismo, se actualiza el coste total y la cota
// Se desmarca el pastelero
// Obtiene el nodo de la cima del monticulo (el minimo)
// Clona el nodo inicial
// El pastelero no esta asignado, se asigna el pastelero al pedido actual, se asigna y se actualiza el coste total del pastelero i en para el pastel de la posicion del pedido k
// Ya se asignaron todos los pasteleros
// El coste total es menor o igual a la cota de pesimismo, se actualiza el coste total y la cota
// Se desmarca el pastelero
```

```
// El pastelero no esta asignado, se asigna el pastelero al pedido actual, se asigna y se actualiza el coste total del pastelero i en para el pastel de la posicion del pedido k
// Ya se asignaron todos los pasteleros
// Se desmarca el pastelero
// El pastelero no esta asignado, se asigna el pastelero al pedido actual, se asigna y se actualiza el coste total del pastelero i en para el pastel de la posicion del pedido k
// Ya se asignaron todos los pasteleros
// El coste total es menor o igual a la cota de pesimismo, se actualiza el coste total y la cota
// Se desmarca el pastelero
==========
** Coste total: 2

** Asignacion de pasteleros

==========

** [Pedido 1] | Pastelero: 1

** [Pedido 2] | Pastelero: 1

** [Pedido 3] | Pastelero: 1

==========
Tiempo de ejecucion: 41 ms
```

Listado del código fuente completo

Clase Main

```
package base;


import java.io.BufferedReader;

import java.io.BufferedWriter;

import java.io.File;

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

import java.io.InputStreamReader;

import java.io.OutputStreamWriter;

import java.util.Arrays;

import java.util.List;



public class Main {


	public static BufferedReader reader = new BufferedReader(new
InputStreamReader(System.in));


	/*
	 * Comprueba la validez del fichero de entrada
	 * @param file
	 * @return boolean
	 */
	public static Boolean checkInput(String file) throws IOException {
		File f = new File(file);
		if(!f.exists() || !f.isFile() || !f.canRead()) {
			System.out.println("** INFO: No se ha proporcionado fichero de
entrada.");
```

```
                System.out.println("** INFO: Se requeriran los datos durante la
ejecucion.");
                return false;
            } else {
                try {
                    BufferedReader br = new BufferedReader(new
InputStreamReader(new FileInputStream(file), "utf-8"));
                    int controllInput = Integer.parseInt(br.readLine());
                } catch (Exception e) {
                    System.out.println("** INFO: No se ha proporcionado fichero
de entrada.");
                    System.out.println("** INFO: Se requeriran los datos durante la
ejecucion.");
                    return false;
                }
                return true;
            }
        }

        /*
         * Comprueba la validez del fichero de salida
         * @param file
         * @return boolean
         */
        public static Boolean checkOutput(String file) {
            if(!file.contains(".txt") && !file.contains(".dat")) {
                System.out.println("** INFO: Fichero de salida no proporcionado, el
resultado se proporcionara por pantalla.");
                return false;
            }
            File f = new File(file);
            f=f.getAbsoluteFile();
```

```
                if(!f.getParentFile().exists() || !f.getParentFile().canWrite()) {
                        System.out.println("** ERROR: No se puede manipular el fichero de
salida.");
                        return false;
                }
                if(f.exists()) {
                        if(!f.isFile() || !f.canWrite()) {
                                System.out.println("** ERROR: El archivo de salida no es un
fichero o no puede sobreescribirse.");
                                return false;
                        }
                }
                return true;
        }

        /*
         * Menu de ayuda
         */
        public static void getHelp() {
                System.out.println("SINTAXIS: pasteleria [fichero entrada] [fichero salida] [-t][-
h] ");
                System.out.println("-t\t\tTraza el algoritmo");
                System.out.println("-h\t\tMuestra esta ayuda");
                System.out.println("[fichero entrada]\t\tNombre del fichero de entrada");
                System.out.println("[fichero salida]\t\tNombre del fichero de salida");
        }


        /*
         * Imprime los datos finales en el fichero parametrizado
         * @params int[] pasteleros
         * @params bw fichero sobre el que escribir
```

```
	 */
	public static void interpretaData(int[] pasteleros, BufferedWriter bw) throws
IOException {
		bw.write("** Coste total: " +Pasteleria.costeFinal + "\n");
		bw.write("** Asignacion de pasteleros\n");
		bw.write("==========================" + "\n");
		for (int b = 0; b < pasteleros.length; b++) {
			bw.write("** [Pedido " + (b+1) +"] | Pastelero: " +
(Pasteleria.pastelerosFinal[b] + 1) + "\n");
		}
		bw.write("==========================");
	}

	/*
	 * Muestra los datos finales por consola
	 * @params int[] pasteleros
	 */
	public static void interpretaData(int[] pasteleros) {
		System.out.println("==========================");
		System.out.println("** Coste total: " +Pasteleria.costeFinal + "\n");
		System.out.println("** Asignacion de pasteleros\n");
		System.out.println("==========================" + "\n");
		for (int b = 0; b < pasteleros.length; b++) {
			System.out.println("** [Pedido " + (b+1) +"] | Pastelero: " +
(Pasteleria.pastelerosFinal[b] + 1) + "\n");
		}
		System.out.println("==========================");
	}

	public static void main(String[] args) throws IOException, NumberFormatException  {
		long startTime = System.currentTimeMillis();
		//////
```

```
		if(args.length != 0) {
			List<String> transformedData = Arrays.asList(args);
			Boolean needHelp = transformedData.contains("-h"); // Muestra la
ayuda
			Boolean isTraza = transformedData.contains("-t"); // Muestra la traza
de lo que el script hace
			BufferedWriter bw = null;
			BufferedReader br = null;
			//////

			if(isTraza) {
				System.out.println("");
				System.out.println("// MAIN //");
			}

			// MARK: GET HELP
			if(needHelp)
				getHelp();

			// MARK: PREPARA FICHEROS O SALIDAS
			if(isTraza)
				System.out.println("	// Leyendo datos de entrada...");
			Boolean hasInputFile = checkInput(args[0]);
			if(hasInputFile)
				br = new BufferedReader(new InputStreamReader(new
FileInputStream(args[0]), "utf-8")); // Lectura del fichero de entrada

			if(isTraza)
				System.out.println("	// Preparando fichero de salida...");
			if((hasInputFile && args.length > 1 && checkOutput(args[1])) ||
(!hasInputFile && checkOutput(args[0]))) { // Si ha aportado fichero de salida, el resultado se
imprimira en el mismo en caso de ser valido
				if(hasInputFile)
```

```
                                        bw = new BufferedWriter(new
OutputStreamWriter(new FileOutputStream(args[1]), "utf-8"));
                                else
                                        bw = new BufferedWriter(new
OutputStreamWriter(new FileOutputStream(args[0]), "utf-8"));
                        }

                        // MARK: OBTIENE PASTELEROS
                        int numPasteleros = 0;
                        if(hasInputFile) {
                                numPasteleros = Integer.parseInt(br.readLine()); // La primera
linea siempre sera el total de pasteleros
                        } else {
                                do {
                                        System.out.println("** Introduce numero (> 0) de
pasteleros:");
                                        numPasteleros = Integer.parseInt(reader.readLine());
                                } while (numPasteleros < 0);
                        }
                        if(isTraza)
                                System.out.println("     // Se obtiene numero de pasteleros: " +
numPasteleros);


                        // MARK: OBTIENE PASTELES
                        int numPasteles = 0;
                        if(hasInputFile) {
                                numPasteles = Integer.parseInt(br.readLine()); // Distintos
pasteles
                        } else {
                                do {
                                        System.out.println("** Introduce numero (> 0) de
pasteles:");
                                        numPasteles = Integer.parseInt(reader.readLine());
```

```
                } while (numPasteles < 0);
            }
            if(isTraza)
                System.out.println("      // Se obtiene el numero diferentes pasteles: " + numPasteles);

            // MARK: OBTIENE PEDIDO
            int pedido [];
            if(hasInputFile) {
                String[] pastelesSt = br.readLine().split("-");
                pedido = new int[pastelesSt.length];
                for (int i = 0; i < pastelesSt.length; i++) {
                    if(isTraza)
                        System.out.println("      // Leemos pedido: " + Integer.parseInt(pastelesSt[i]) + " y lo añadimos al array de pedido");
                    pedido[i] = Integer.parseInt(pastelesSt[i]); // RESTAMOS 1 PORQUE LOS NUMEROS EN JAVA EMPIEZAN DESDE EL 0 Y NATURALMENTE DESDE EL 1 (p.ej 1 = 0)
                }
            } else {
                int longitudPedido = 0;
                do {
                    System.out.println("** Cuantos pedidos va a haber? (> 0)");
                    longitudPedido = Integer.parseInt(reader.readLine());
                } while (longitudPedido < 0);
                pedido = new int[longitudPedido];
                for (int i = 0; i < longitudPedido; i++) {
                    int pedidoActual = 0;
                    do {
                        System.out.println("Pastel para el pedido número " + i + ":");
```

```
                                        pedidoActual =
Integer.parseInt(reader.readLine()); // RESTAMOS 1 PORQUE LOS NUMEROS EN JAVA
EMPIEZAN DESDE EL 0 Y NATURALMENTE DESDE EL 1 (p.ej 1 = 0)
                                    } while (pedidoActual < 0 || pedidoActual >
numPasteles);
                                    if(isTraza)
                                        System.out.println("     // Añadimos pastel: " +
pedidoActual + " al array de pedido en " + i +"º posicion");
                                    pedido[i] = pedidoActual;
                                }
                            }
                            if(isTraza)
                                System.out.println("     // Nuestro pedido es: " +
Arrays.toString(pedido));


                            // MARK: OBTIENE COSTES
                            int[][] costes = new int [numPasteleros][numPasteles];
                            if(hasInputFile) {
                                for (int i = 0; i < numPasteleros; i++) {
                                    String[] costesRead = br.readLine().split(" ");
                                    for (int m = 0; m < costesRead.length; m++) {
                                        if(isTraza)
                                            System.out.println("     // Añadimos
coste: " + Integer.parseInt(costesRead[m]) + " al pastelero " + i +" para el pastel "+ m);
                                        costes[i][m] = Integer.parseInt(costesRead[m]);


                                    }
                                }
                            } else {
                                System.out.println("Costes de preparacion");
                                for (int i = 0; i < numPasteleros; i++) {
                                    for (int j = 0; j < numPasteles; j++) {
                                        int costeActual = 0;
```

```
                                        do {
                                                System.out.println("Pastelero "+i+" ;
pastel "+j+":");
                                                costeActual =
Integer.parseInt(reader.readLine());
                                        } while (costeActual < 0);
                                        costes[i][j] = costeActual;
                                        if(isTraza)
                                                System.out.println("     // Añadimos
coste: " + costes[i][j] + " al pastelero " + i +" para el pastel "+ j);
                                    }
                                }
                        }



                        //MARK: Ejecucion
                        int[] pasteleros = new int[costes.length];
                        Pasteleria.pastelerosFinal = pasteleros.clone();
                        for (int i = 0; i < pasteleros.length; i++) {
                                Pasteleria.pastelerosFinal[i] = 0;
                        }
                        int costeT = 0;
                        pasteleros = Pasteleria.AsignaPasteleros(costes, pedido, pasteleros,
costeT,isTraza);
                        // Impresion asignacion final de pasteleros y el coste total
                        if(bw != null) { // en caso de haberse proporcionado fichero de salida
se mostraran los datos a traves de el
                                interpretaData(pasteleros, bw);
                                System.out.println("** Resultados en el archivo indicado.");
                                bw.close();
                        }else { // Sino se escribe por consola
                                interpretaData(pasteleros);
```

```
                }
                if(br!=null)
                        br.close();

                long endTime = System.currentTimeMillis() - startTime;
                System.out.println("Tiempo de ejecucion: " + endTime + " ms");
            }
        }
}
```

Clase montículo

```
package base;


import java.util.Comparator;

import java.util.PriorityQueue;


public class Monticulo {

	public static class Nodo {

		/* VARIABLES */

		int[] pasteleros;			// Vector de pasteleros asignados a cada
pedido

		boolean[] asignados;	// Vector que indica si un pastelero ya ha sido asignado

		int k;					// Pedido actual

		int costeT;				// Coste total hasta el momento

		double estOpt;			// Estimacion optima de la rama

		public Nodo(int[] pasteleros, boolean[] asignados, int k, int costeT, double
estOpt) {

			this.pasteleros = pasteleros;

			this.asignados = asignados;

			this.k = k;

			this.costeT = costeT;

			this.estOpt = estOpt;

		}

	}


	public static class MonticuloMinimos {

		// Se inicia via colas de prioridad (priorityqueue)

		PriorityQueue<Nodo> colaPrioridad = new PriorityQueue<>(new
ComparadorNodo());


		public void agregarNodo(Nodo nodo) { // añadir

			colaPrioridad.add(nodo);
```

```
        }

        public Nodo obtenerNodoMinimo() { // obtener el nodo mas pequeño (primer elemento)
            return colaPrioridad.poll();
        }
    }

    public static class ComparadorNodo implements Comparator<Nodo> {
        @Override
        public int compare(Nodo nodo1, Nodo nodo2) { // Override del metodo compare, devuelve la estimacion optima mas pequeña
            return Double.compare(nodo1.estOpt, nodo2.estOpt);
        }
    }
}
```

Clase Pasteleria

```
package base;

import java.util.*;

import base.Monticulo.MonticuloMinimos;
import base.Monticulo.Nodo;

public class Pasteleria {
	static int costeFinal = 0;
	static int[] pastelerosFinal;

	// ESTIMACION OPTIMA DE LA RAMA
	public static double estimacionOpt(int[][] costes, int[] pedido, int
k, int costeT, boolean isTraza) {
		if(isTraza)
			System.out.println("// BUSQUEDA ESTIMACION OPTIMA");
		double estimacion = costeT;
		int past = 0;
		for (int i = k + 1; i < pedido.length; i++) {
			// Buscamos el pastelero con menor coste para el pedido i
			int menorC = costes[0][pedido[i]];
			if(isTraza)
				System.out.println("	// Menor coste pedido
"+i+" (partida) pastelero 0: "+menorC);
			for (int j = 1; j < pedido.length; j++) {
				if (menorC > costes[j][pedido[i]]) {
					menorC = costes[j][pedido[i]];
					past = j;
					if(isTraza)
						System.out.println("	//
Encontrado menor coste pedido "+i+" (partida) pastelero "+past+": "+menorC);
				}
			}
			pastelerosFinal[i] =past;
			estimacion += menorC;
			if(isTraza)
				System.out.println("	// Encontrado menor coste
pedido "+i+" (FINAL) pastelero "+past+": "+menorC);
		}
		if(isTraza)
			System.out.println("	// Estimacion optima de este
nodo: "+estimacion);
		return estimacion;
	}

	// ESTIMACION PESIMA DE LA RAMA
	public static double estimacionPes(int[][] costes, int[] pedido, int
k, int costeT, boolean isTraza) {
		if(isTraza)
			System.out.println("// BUSQUEDA ESTIMACION PESIMA");
		double estimacion = costeT;
		for (int i = k +1; i < pedido.length; i++) {
			// Buscamos el pastelero con mayor coste para el pedido i
			int mayorC = costes[0][pedido[i]];
			for (int j = 1; j < pedido.length; j++) {
				if (mayorC < costes[j][pedido[i]]) {
					mayorC = costes[j][pedido[i]];
				}
```

```
                }
                estimacion += mayorC;
                if(isTraza)
                        System.out.println("      // Encontrado menor coste
pedido "+i+" (FINAL): "+mayorC);
            }
            if(isTraza)
                System.out.println("      // Estimacion pesima de este
nodo: "+estimacion);
            return estimacion;
      }

      public static int[] AsignaPasteleros(int[][] costes, int[] pedido,
int[] pasteleros, int costeT, boolean isTraza){
            if(isTraza)
                System.out.println("INICIO METODO ASIGNAPASTELEROS");
            MonticuloMinimos monticulo = new MonticuloMinimos(); //Crear un
monticulo de minimos
            costeT = 0; //Inicializar el coste total en 0
            if(isTraza)
                System.out.println("// Se crea un monticulo de minimos y
se asigna coste 0");

            boolean[] asignados = new boolean[pedido.length];
            Arrays.fill(asignados,false);
            if(isTraza)
                System.out.println("// Se crea un array de booleanos con
los pasteleros asignados y se establece todo su contenido a false");

            Nodo nodo = new Nodo(pasteleros, asignados, 0, costeT, 0);
//Crear el primer nodo
            if(isTraza)
                System.out.println("// Se crea el primer nodo");

            nodo.estOpt = estimacionOpt(costes, pedido, nodo.k, nodo.costeT,
isTraza); //Calcular la estimacion optima del nodo
            if(isTraza)
                System.out.println("// Se calcula la estimacion optima del
nodo incial");

            monticulo.agregarNodo(nodo); //Añdir el nodo al monticulo
            if(isTraza)
                System.out.println("// Se agrega el nodo al monticulo");

            double cota = estimacionPes(costes, pedido, nodo.k,
nodo.costeT,isTraza); //Calcular la cota de pesimismo
            if(isTraza)
                System.out.println("// Se obtiene la cota (coste
pesimista) y se crea el primer nodo hijo con ella");
            Nodo hijo = new Nodo(pasteleros, asignados, costeT, costeT,
cota);

            if(isTraza)
                System.out.println("// Mientras que la cola no este vacio
o el siguiente elemento tenga una estimacion optima inferior a la cota");
            while(!monticulo.colaPrioridad.isEmpty() &&
monticulo.colaPrioridad.peek().estOpt <= cota){
                nodo = monticulo.obtenerNodoMinimo(); //Obtener el nodo en
la cima del monticulo
```

```
                    if(isTraza)
                          System.out.println("// Obtiene el nodo de la cima
del monticulo (el minimo)");
                    hijo = new Nodo(nodo.pasteleros.clone(),
nodo.asignados.clone(), nodo.k+1, nodo.costeT, 0); //Clonar nuestro nodo
                    if(isTraza)
                          System.out.println("// Clona el nodo inicial");
                    for(int i=0; i<pedido.length; i++){
                          if(!hijo.asignados[i]){ //Si el pastelero no esta
asignado
                                if(isTraza)
                                      System.out.println("// El pastelero no
esta asignado, se asigna el pastelero al pedido actual, se asigna y se
actualiza el coste total del pastelero i en para el pastel de la posicion del
pedido k");
                                hijo.pasteleros[hijo.k] = i; //Asignar el
pastelero actual
                                hijo.asignados[i] = true; //Marcar el
pastelero como asignado
                                hijo.costeT = nodo.costeT +
costes[i][pedido[hijo.k]]; //Actualizar el coste total

                                if(hijo.k == pedido.length-1){ //Si ya se
asignaron todos los pasteleros
                                      if(isTraza)
                                            System.out.println("// Ya se
asignaron todos los pasteleros");
                                      if(cota >= hijo.costeT){ //Si el coste
total es menor que la cota de pesimismo
                                            if(isTraza)
                                                  System.out.println("// El
coste total es menor o igual a la cota de pesimismo, se actualiza el coste
total y la cota");
                                            pasteleros = hijo.pasteleros;
                                            costeT =
hijo.costeT;//Actualizar el coste
                                            cota = costeT; //Actualizar la
cota de pesimismo
                                      }
                                }
                                else{ //Si la solucion no esta completa
                                      if(isTraza)
                                            System.out.println("// Todavia
no se han asignado todos los pasteleros, se crea un nuevo nodo y se
actualizan estimaciones");
                                      hijo.estOpt = estimacionOpt(costes,
pedido, hijo.k, hijo.costeT,isTraza); //Calcular la estimacion optima del
nodo hijo
                                      monticulo.agregarNodo(hijo); //Añadir
el nodo hijo al monticulo
                                      double estPes = estimacionPes(costes,
pedido, hijo.k, hijo.costeT,isTraza); //Calcular la estimacion de pesimismo
del nodo hijo
                                      if(cota > estPes){ //Si la cota de
pesimismo es mayor que la estimacion de pesimismo del nodo hijo
                                            cota = estPes; //Actualizar la
cota de pesimismo
                                      }
                                }
```

```
                                    if(isTraza)
                                          System.out.println("// Se desmarca el
pastelero");
                                    hijo.asignados[i] = false; //Desmarcar el
pastelero
                              }
                        }
                  }
                  costeFinal = hijo.costeT;
                  return pasteleros;
            }
}
```