



PROGRAMACIÓN Y ESTRUCTURAS DE DATOS AVANZADAS

PRIMERA PRUEBA
DE
EVALUACIÓN A DISTANCIA
(PED1)

CURSO 2015/2016

David Pastor Sanz

Índice

| | |
|---|-----------|
| 1. Esquema algorítmico | 2 |
| 2. Demostración de optimilidad | 4 |
| 3. Coste computacional del algoritmo | 5 |
| 4. Alternativas al esquema utilizado | 6 |
| 5. Datos de prueba | 7 |
| 6. Código fuente | 11 |
| 6.1. Paquete de Montículos: heap | 11 |
| 6.1.1. Interfaz de la clase montículo: HeapIN | 11 |
| 6.1.2. Montículo de mínimos: HeapMin | 12 |
| 6.2. Paquete servidor: server | 15 |
| 6.2.1. Clase de lectura de archivo de entrada: Reader | 15 |
| 6.2.2. Clase servidor (contiene el algoritmo voraz): Server | 17 |
| 6.2.3. Clase cliente: Client | 18 |
| 6.2.4. Clase controladora, contiene el main: Controller | 19 |

MINIMIZACIÓN DEL TIEMPO EN EL SISTEMA

1. Esquema algorítmico

Se ha utilizado un esquema algorítmico voraz, tal como pide el enunciado de la práctica. Este esquema consiste en encontrar un conjunto de candidatos que constituya una solución y que optimice una función objetivo. Dicho algoritmo se ha realizado según el esquema descrito en el siguiente pseudocódigo de alto nivel:

```
1. fun Voraz( c: conjuntoCandidatos ): conjuntoCandidatos
2.   sol ← ∅
3.   mientras c ≠ ∅ ∧ ¬ solucion( sol ) hacer
4.     x ← seleccionar( c )
5.     c ← c \ {x}
6.     si factible( sol ∪ {x} ) entonces
7.       sol ← sol ∪ {x}
8.     fsi
9.   fmientras
10.  si solucion( sol ) entonces devolver sol
11.  sino imprimir( 'no hay solucion' )
12. ffun
```

La función de este algoritmo, empezando como solución el conjunto vacío, es que mientras la solución no sea factible y el conjunto de candidatos no esté vacío, irá seleccionando el candidato más prometedor, que cumpla que haga a la solución que sea subóptima. Este proceso se repetirá hasta que se cumpla la condición de que el conjunto de candidatos esté vacío o se haya encontrado la solución.

Para solucionar el problema de una manera optima primero se van a diferenciar, a partir del esquema, las variables y funciones necesarias:

- **Conjunto inicial de candidatos, *c*:** Serán los clientes. La posición del cliente y su tiempo de estancia en el sistema.
- **Conjunto de candidatos que ya han sido considerados y seleccionados, *sol*:** Empezará como conjunto vacío y cada vez que se seleccione uno que cumpla la solución optima se añadirá a este.
- **Función, *solucion()*:** Determina si la solución es óptima. En este caso cada vez que se añada un candidato a la solución será una solución subóptima, cuando ya no queden candidatos se puede decir que el conjunto solución es una solución óptima.
- **Función factible, *factible()*:** Determina si añadiendo nuevos candidatos la solución será factible o no.
- **Función de selección, *seleccionar()*:** Escoge al candidato más prometedor de los que todavía no se han considerado.

En cada problema concreto habrá que particularizar cada uno de estos conjuntos y funciones. En el caso del problema del enunciado de calcular el *Tiempo de Minimización de Estancia en el Sistema*, se

ha realizado el siguiente algoritmo, en lenguaje de programación *Java*, para lograr el mínimo tiempo de estancia de cada cliente y en general del tiempo del servicio (se han eliminado las sentencias que crean la traza para que sea más visible el algoritmo):

```
1 public static String [] timeMinimalize( ArrayList<Client> clients )
2 {
3     String [] sol = new String [2];
4     HeapIN<Client> h = new HeapMin<Client>( clients );
5     String orderClients = new String ();
6     int totalTime = 0;
7
8     while( !h.isEmpty() )
9     {
10        Client c = h.top();
11        orderClients += c.getPosition() + " ";
12        totalTime += totalTime + c.getWaitingTime();
13    }
14
15    sol[0] = totalTime + " ";
16    sol[1] = orderClients; incluye la traza
17    return sol;
18 }
```

Para conseguir la solución óptima a este problema, minimizar el tiempo de servicio, se han de ir seleccionando los candidatos, en orden de menor a mayor, que necesiten tiempo de uso del servicio. Para conseguir esto siguiendo el esquema voraz, conseguir la solución óptima sin necesidad de reconsiderar decisiones ya tomadas anteriormente, se ordenarán los candidatos en una estructura de menor a mayor tiempo que necesiten en el servicio. Se ha decidido ordenar mediante el uso de montículos.

En este caso en particular cada una de los conjuntos y funciones nombrados anteriormente vienen dados por:

1. El conjunto inicial de candidatos será el *ArrayList* de clientes (objetos que contienen el número de cliente y su tiempo de uso del sistema), *clients*, que primeramente antes de empezar a seleccionar candidatos de él, se volcaran sus elementos (clientes) en una estructura de datos de tipo *montículo de mínimos*, *h*.
2. La función de selección será la que escoge el cliente con menos tiempo de uso del sistema que haya en el conjunto de candidatos. Al haber metido todos los candidatos en un montículo de mínimos, esta función simplemente será sacar la cima del montículo y restaurar la propiedad del montículo mediante la sentencia de la línea 11, la función *top()* del montículo.
3. El conjunto de candidatos ahora será menor, ya que se ha eliminado el candidato obtenido en la función de selección.
4. El añadir un candidato a la solución siguiendo este criterio siempre será factible con lo cual se añadirá el candidato a la solución. La variable *totalTime* irá acumulando todos los tiempos de cada candidato que haya sido considerado además del tiempo del servicio en el sistema hasta el momento. Y la variable *orderClients* que almacenará en un *String* el orden de los clientes que se ha seguido para conseguir la solución óptima.
5. Hasta que el conjunto de candidatos no esté vacío se repetirán todos los pasos anteriores, esta será la función solución. Una vez el conjunto de candidatos está vacío el algoritmo podrá devolver la solución, que en este caso será un *array* de *String* en el que se encuentra el tiempo total de servicio y el orden de los clientes que se ha utilizado para conseguir la solución más óptima, el menor tiempo de servicio del sistema.

2. Demostración de optimilidad

Tomando a $P = (p_1, p_2, \dots, p_n)$ como una permutación cualquiera de un conjunto de números naturales $\{1, 2, \dots, n\}$ que representan el tiempo de estancia de cada cliente, t_n . Si se atiende a dichos clientes según la secuencia P , el tiempo total que los clientes estarán en el sistema vendrá dado por la suma de todos los tiempos de cada cliente más el tiempo de la suma de los anteriores. Se puede describir de manera matemática como:

$$\begin{aligned} T(P) &= t_1 + (t_1 + t_2) + \dots + (t_n + (t_1 + t_2 + \dots + t_{n-1})) = nt_1 + (n-1)t_2 + \dots + t_n \\ &= \sum_{k=1}^n (n-k+1)t_k \end{aligned}$$

Se supone que la permutación P no organiza a los candidatos (clientes) en orden creciente de tiempos de estancia en el servicio. Con lo cual se tendría dos enteros naturales x e y que cumplirían que $x < y$ y que $t_x > t_y$. Si se intercambian ambas posiciones en la secuencia P , se obtendrá una nueva secuencia de servicio P' y su tiempo total en el sistema sería:

$$T(P') = (n-x+1)t_y + (n-y+1)t_x + \sum_{\substack{k=1 \\ k \neq x, y}}^n (n-k+1)t_k$$

Si se restan ambas permutaciones quedaría:

$$\begin{aligned} T(P) - T(P') &= (n-x+1)(t_x - t_y) + (n-y+1)(t_y - t_x) = (n-x+1 - n+y-1)(t_x - t_y) \\ &= (y-x)(t_x - t_y) > 0 \end{aligned}$$

Al tener que $x < y$ y que $t_x > t_y$ se sabe que la permutación P' necesita menor tiempo que la P . Como conclusión se puede decir que cualquier permutación que no esté ordenada en orden de tiempos creciente se podría mejorar hasta conseguir la solución óptima buscada en el problema.

En el caso que los tiempos de x e y sean iguales ($t_x = t_y$), su orden será indiferente. Será lo mismo que estén colocados como $x > y$ que como $y > x$.

Como conclusión se puede decir que el algoritmo pedido consiste simplemente en ordenar las tareas, el tiempo de servicio de cada cliente, según el orden establecido en la estrategia, en un orden creciente de dichos tiempos de estancia de cada cliente.

Por tanto, la secuencia óptima, que no podrá mejorarse, será aquella que tenga a todos sus clientes ordenados por su tiempo de estancia en el sistema en orden creciente.

3. Coste computacional del algoritmo

Para calcular el coste computacional del algoritmo realizado, mostrado en el apartado 1 de este documento (*timeMinimalize()*), se va a proceder a observar y comentar línea a línea para llegar a una conclusión final.

- **Tamaño del problema:** El tamaño del problema serán el número de clientes que entran en el servicio. Vendrá dado por la longitud del *ArrayList* de clientes *clients* introducido como parámetro.
- **Líneas 3, 5, 6, 15 y 16:** En estas sentencias se crean, inicializan o se asignan valores a variables cuyo coste asintótico temporal será constante, $T(n) \in \mathcal{O}(1)$.
- **Línea 4:** Aquí se crea un montículo de mínimos a partir del *ArrayList* introducido como parámetro, usando el constructor de crear un montículo lineal de dicha clase. Para calcular el coste de esta sentencia se va a observar el método (la clase montículo ha sido implementada por el alumno):

```

1 public HeapMin( ArrayList<T> v )
2 {
3     if( v != null )
4     {
5         this.MAX = v.size() ;
6         this.vec = v;
7         this.numElem = v.size() ;
8         for( int i = v.size()/2; i > -1; i-- )
9             sink( i, this.vec );
10    }
11 }

```

Siempre que el *ArrayList*, introducido como parámetro, no sea un objeto nulo se inicializarán las variables de clase (tamaño máximo del montículo, número de elementos que contendrá y el campo *vec*, que será un puntero al *ArrayList* parámetro). Todas estas sentencias tendrán un coste asintótico temporal de $T(n) = \mathcal{O}(1)$. Las siguiente sentencia, el método hundir, *sink()*, se repetirá la mitad de n veces lo que hará al coste de este bucle ser n veces el coste de *sink()*. Este método se encarga de hundir un elemento en el montículo siempre que, en este caso de montículo de mínimos, el elemento a hundir sea mayor que sus dos hijos. Al empezar en la altura $(k-1)$ del árbol se evitará tener que hundir 2^k nodos (las hojas del árbol). El número de iteraciones totales por el bucle será:

$$\sum_{i=1}^{k-1} i2^{k-i-1} = 2^k - k - 1 = n - \log n - 1$$

Dicha expresión $n - \log n - 1 \in \mathcal{O}(n)$, con lo cual queda que el coste asintótico temporal de dicha creación del montículo lineal será de $T(n) \in \mathcal{O}(n)$.

- **Bucle while:** Este bucle se repetirá n veces. En el cual se realizarán las asignaciones con coste constante de las líneas 11 y 12. La línea 10 realiza la función de obtención de la cima del montículo, borrándola de esta y restaurando la propiedad de la estructura (las dos primeras tienen un coste constante, la tercera, mediante el uso de la función hundir) tendrá un coste logarítmico. Por tanto el coste asintótico temporal de esta función será de $T(n) \in \mathcal{O}(\log n)$. Como repetimos n veces estas sentencias se puede llegar a la conclusión que el coste del bucle será tal que: $n(1+\log n) = n+n \log n$.

Sumando todas estos costes de cada sentencia o grupo de sentencias del método quedaría tal que $(5 \cdot 1 + n + n \log n) \in \mathcal{O}(n \log n)$ por lo tanto el coste asintótico temporal del algoritmo realizado es de $T(n) \in \mathcal{O}(n \log n)$.

4. Alternativas al esquema utilizado

Una alternativa posible sería utilizando otro algoritmo *voraz*, esta vez de forma “*pura*”. Este debería ir recorriendo el vector de tiempos de clientes e ir seleccionando cada vez el menor de todos. En este caso la función de *selección*, cada vez que se la llame recorrerá todos los elementos del conjunto de candidatos y escogerá al que menor tiempo tenga y no haya sido seleccionado todavía (habría que indicarlo mediante un vector de *booleano*). Esta función tendría un coste asintótico temporal lineal ($T(n) \in \mathcal{O}(n)$) ya que tendría que recorrer el vector de tiempos cada vez que se le llamase. *Selección* se llama n veces, desde el bucle del método *voraz*, con lo cual el coste asintótico temporal en el caso peor sería cuadrático, $T(n) \in \mathcal{O}(n^2)$.

Comparado este esquema con la solución dada, se ve que el coste es mayor usando este y que es mejor solución ordenar previamente el vector de tiempos en orden creciente y después simplemente ir cogiendo candidato a candidato.

Otra posible solución sería utilizar un algoritmo que use la *fuerza bruta*. Consiste en enumerar todas las posibles soluciones del problema y después escoger la que lo optimice. El número posible de soluciones sería de $n!$, todas las permutaciones posibles entre todos los candidatos, y habría que elegir la más optima de todas ellas. Para verlo mejor se va a poner un ejemplo:

Suponiendo que se tienen tres clientes y sus tiempos de servicio $t_1 = 2$, $t_2 = 8$ y $t_3 = 4$. Tendríamos 6 posibilidades. En la siguiente tabla se muestran dichas posibilidades y el resultado de cada una:

| Orden de Servicio | $T = \sum_{i=1}^n$ tiempo en el sistema del cliente i |
|-------------------|---|
| 1 2 3 | $2+(2+8)+(2+8+4)=26$ |
| 1 3 2 | $2+(2+4)+(2+4+8)=\mathbf{22}$ |
| 2 1 3 | $8+(8+2)+(8+2+4)=32$ |
| 2 3 1 | $8+(8+4)+(8+4+2)=34$ |
| 3 1 2 | $4+(4+2)+(4+2+8)=24$ |
| 3 2 1 | $4+(4+8)+(4+8+2)=30$ |

Este algoritmo debería ir calculando línea a línea y almacenar la combinación que menos tiempo necesite en una variable, será la solución óptima que devuelva, en este ejemplo la secuencia *1 3 2* cuyo tiempo total en el servicio será de 22 u.t. (unidades de tiempo). Como el número de veces que tiene que ir calculando es el número de permutaciones posibles de sus candidatos, en este caso $n! = 3! = 6$. Se tendría entonces que se harían $n!$ calculos distintos donde en cada cálculo se irá sumando cada tiempo correspondiente al orden de su permutación. Entonces se podría decir que su coste asintótico temporal sería de $T(n) \in \mathcal{O}(n!)$. Esto podría aplicarse a algoritmos de *vuelta atrás*.

Como conclusión se tiene que el algoritmo de fuerza bruta no mejoraría la solución propuesta al problema. Es más, la empeora bastante. No sería práctico utilizar una solución con coste factorial.

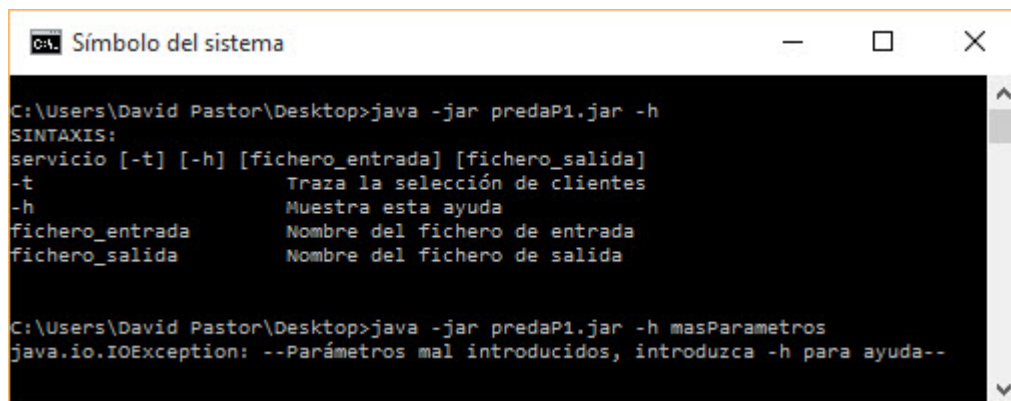
Por tanto la mejor solución dada es la propuesta como un algoritmo *voraz* al que se le da un vector ordenado con todos los candidatos posibles. De esta manera, tal como se ha expuesto en el apartado anterior, se consigue una solución de coste $n \log n$. Que es mejor, tardará menos en ejecutarse, que la propuesta como la del algoritmo *voraz* “*puro*”, con un coste cuadrático; y mucho menos tiempo que el propuesto como *fuerza bruta*, con coste factorial.

También podría solucionarse el problema con un esquema de *ramificación y poda*, pero al poderse solucionar con un esquema *voraz* siempre será menos costoso.

5. Datos de prueba

Se van a mostrar diferentes datos introducidos para probar el programa, entre ellos se probarán con datos que hagan fallar el programa para probar su robustez.

- I*) Para el funcionamiento del comando `-h` se mostrará por pantalla la ayuda (). Si se introducen parámetros de más junto a dicho comando se mostrará un mensaje de error. Se puede observar en la *Figura 1*.

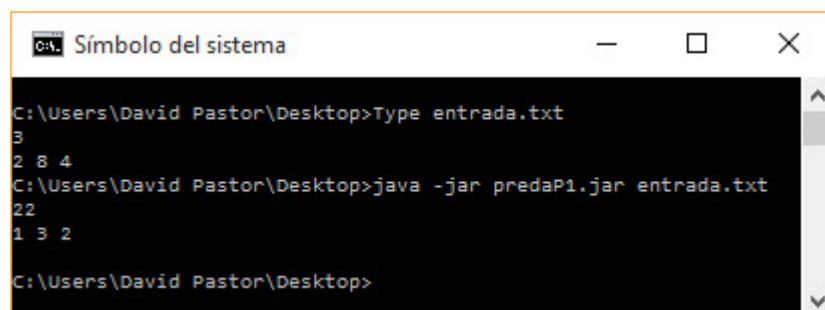


```
C:\Users\David Pastor\Desktop>java -jar predaP1.jar -h
SINTAXIS:
servicio [-t] [-h] [fichero_entrada] [fichero_salida]
-t                      Traza la selección de clientes
-h                      Muestra esta ayuda
fichero_entrada         Nombre del fichero de entrada
fichero_salida          Nombre del fichero de salida

C:\Users\David Pastor\Desktop>java -jar predaP1.jar -h masParametros
java.io.IOException: --Parámetros mal introducidos, introduzca -h para ayuda--
```

Figura 1: Muestra del comando `-h` y un error al meter más parámetros junto a él

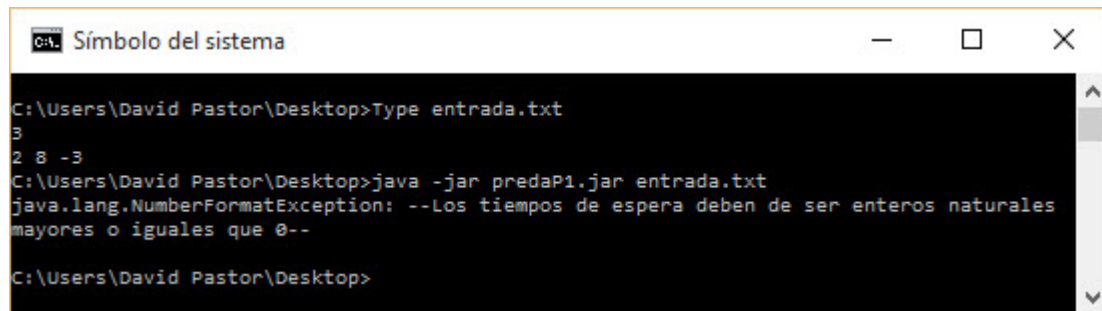
- II*) A continuación se mostrará la salida dada metiendo el fichero de entrada. Para mostrar el archivo en *cmd* se utiliza el comando `Type archivo.txt`. En la *Figura 2* se puede observar el correcto funcionamiento para la entrada mostrada en la imagen.



```
C:\Users\David Pastor\Desktop>Type entrada.txt
3
2 8 4
C:\Users\David Pastor\Desktop>java -jar predaP1.jar entrada.txt
22
1 3 2
C:\Users\David Pastor\Desktop>
```

Figura 2: Fichero de entrada y su salida correspondiente por pantalla de comando

Si los números de la entrada no son enteros naturales (números menores que 0 u otros símbolos) el programa devolverá un mensaje de error. Lo mismo ocurrirá si el número de clientes es menor o mayor que el número de tiempos. También en el caso de que se introduzcan datos de más. En la *Figura 3* se muestra un ejemplo de este tipo de errores.



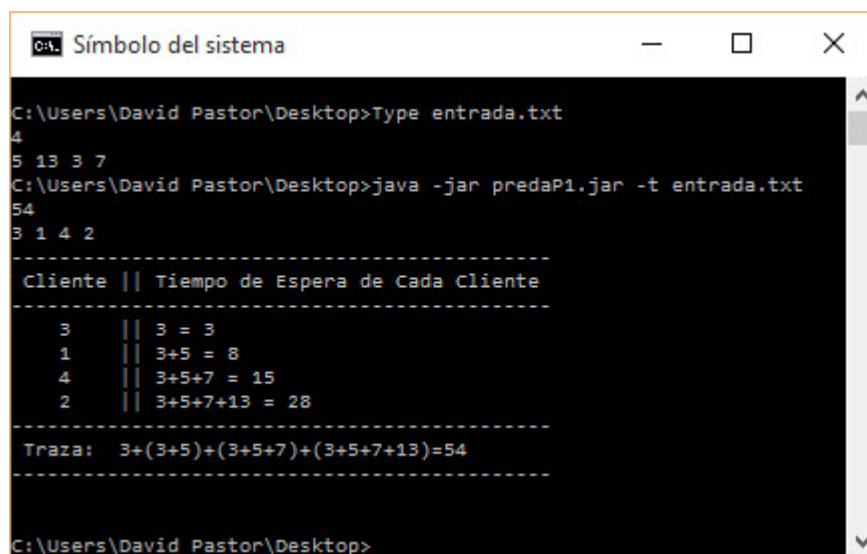
```

C:\Users\David Pastor\Desktop>Type entrada.txt
3
2 8 -3
C:\Users\David Pastor\Desktop>java -jar predaP1.jar entrada.txt
java.lang.NumberFormatException: --Los tiempos de espera deben de ser enteros naturales
mayores o iguales que 0--
C:\Users\David Pastor\Desktop>

```

Figura 3: Fichero de entrada con errores y su salida

III) Ahora se probará la salida de la traza por pantalla usando dos parámetros de entrada, el comando `-t` y el archivo de entrada. En la *Figura 4* se puede ver cual sería la ejecución del programa para un cierto archivo de entrada, mostrado también en esta, y su correcta salida impresa.



```

C:\Users\David Pastor\Desktop>Type entrada.txt
4
5 13 3 7
C:\Users\David Pastor\Desktop>java -jar predaP1.jar -t entrada.txt
54
3 1 4 2

-----
Cliente || Tiempo de Espera de Cada Cliente
-----
3 || 3 = 3
1 || 3+5 = 8
4 || 3+5+7 = 15
2 || 3+5+7+13 = 28
-----

Traza: 3+(3+5)+(3+5+7)+(3+5+7+13)=54
-----
C:\Users\David Pastor\Desktop>

```

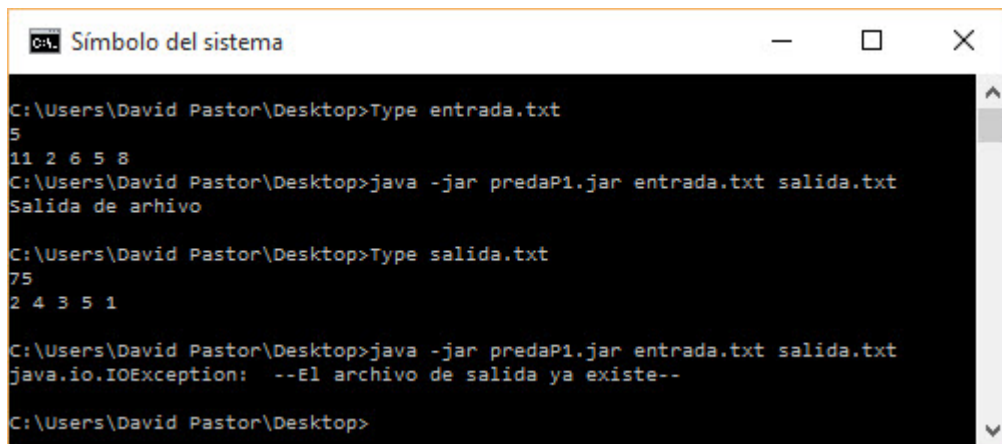
Figura 4: Fichero de entrada con su resultado y traza mostrada por pantalla

Los posibles errores que puede haber en este caso son los mismos que para el anterior. Si el archivo de entrada no existe o contiene datos que no son válidos no se ejecutará el algoritmo y simplemente mostrará un error por pantalla.

Falta matizar que en cada línea que se muestra en la traza se muestra el orden de cada cliente y su tiempo de espera para cada cliente, el cual es su tiempo de servicio más el tiempo de los anteriores.

IV) De nuevo se demostrará el funcionamiento del programa con dos parámetros de entrada, esta vez metiendo el archivo de entrada y el archivo de salida. Respecto al archivo de salida si ya existe, se mostrará por pantalla un mensaje de error indicando que dicho archivo ya existe y el algoritmo no se ejecutará. Si existe algún error en el algoritmo, por datos de entrada erróneos, se mostrará el mensaje de error correspondiente dentro del archivo de salida creado, siempre que no exista ya. Cuando se crea un archivo de salida se mostrará el mensaje de **salida de archivo** por la pantalla de comando.

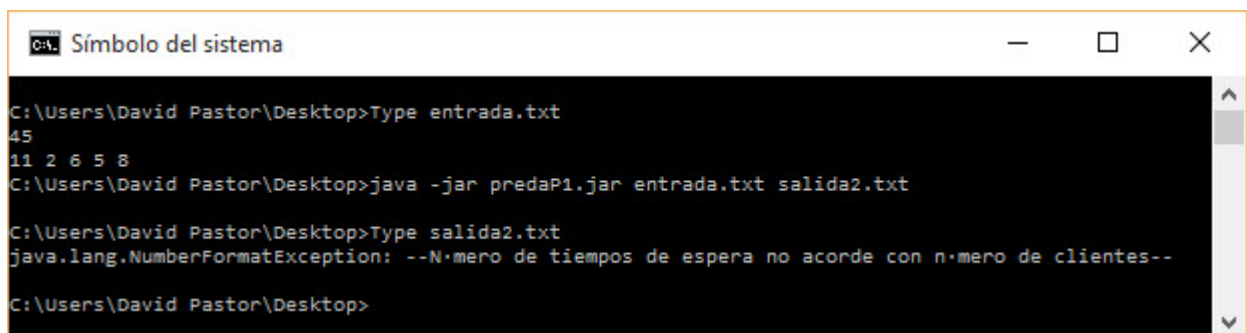
En la siguiente imagen (*Figura 5*) se muestra una posible ejecución del programa, mostrando también el fichero de salida que genera. Además a continuación se muestra el error que generaría al ejecutar el programa con los mismos parámetros, una vez que ya se ha creado el archivo de salida.



```
C:\Users\David Pastor\Desktop>Type entrada.txt
5
11 2 6 5 8
C:\Users\David Pastor\Desktop>java -jar predaP1.jar entrada.txt salida.txt
Salida de arhivo
C:\Users\David Pastor\Desktop>Type salida.txt
75
2 4 3 5 1
C:\Users\David Pastor\Desktop>java -jar predaP1.jar entrada.txt salida.txt
java.io.IOException: --El archivo de salida ya existe--
C:\Users\David Pastor\Desktop>
```

Figura 5: Fichero de entrada con su resultado mostrado en un archivo de salida

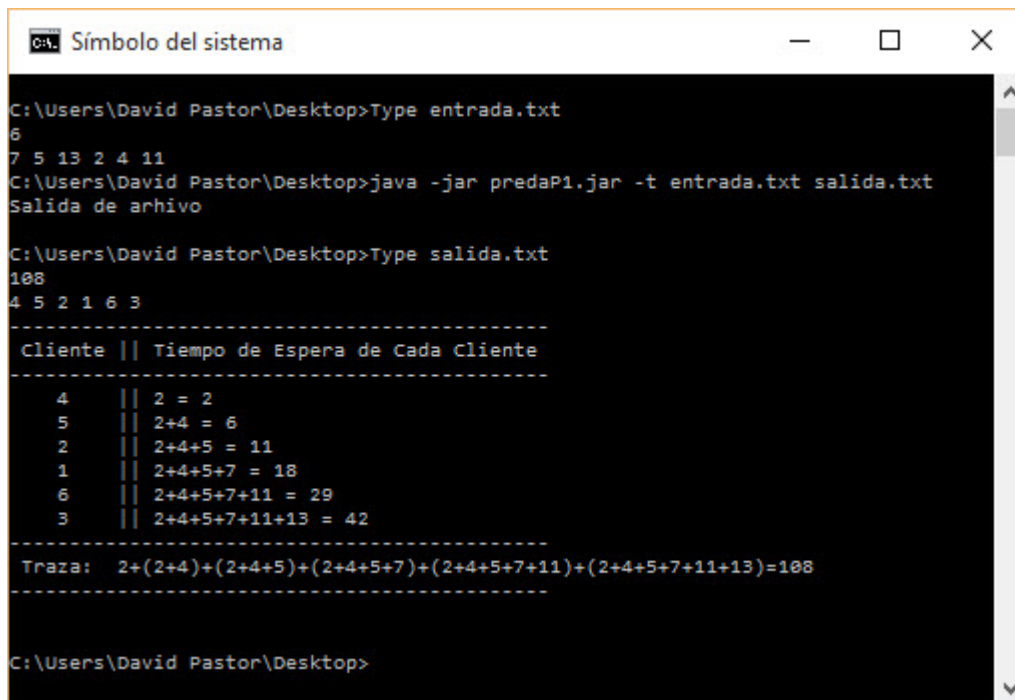
En la *Figura 6* se muestra la ejecución del programa con los mismos parámetros pero con un error en el archivo de entrada. Se puede observar que el error se muestra en el archivo de salida.



```
C:\Users\David Pastor\Desktop>Type entrada.txt
45
11 2 6 5 8
C:\Users\David Pastor\Desktop>java -jar predaP1.jar entrada.txt salida2.txt
C:\Users\David Pastor\Desktop>Type salida2.txt
java.lang.NumberFormatException: --N·mero de tiempos de espera no acorde con n·mero de clientes--
C:\Users\David Pastor\Desktop>
```

Figura 6: Fichero de entrada con error mostrado en un archivo de salida

- ∪) Por último se va a proceder a probar el programa introduciendo los tres argumentos posibles: `-t`, que indicará que se muestre la traza; `archivo_entrada`; y `archivo_salida`, que mostrará la solución al archivo de entrada junto a la traza que realiza, en caso de error y que el archivo de salida no exista, se mostrará el error en el archivo de salida. Se puede observar como se ejecuta en la *Figura 7* que se muestra a continuación.



```
C:\Users\David Pastor\Desktop>Type entrada.txt
6
7 5 13 2 4 11
C:\Users\David Pastor\Desktop>java -jar predaP1.jar -t entrada.txt salida.txt
Salida de archivo

C:\Users\David Pastor\Desktop>Type salida.txt
108
4 5 2 1 6 3
-----
Cliente || Tiempo de Espera de Cada Cliente
-----
4      || 2 = 2
5      || 2+4 = 6
2      || 2+4+5 = 11
1      || 2+4+5+7 = 18
6      || 2+4+5+7+11 = 29
3      || 2+4+5+7+11+13 = 42
-----
Traza:  2+(2+4)+(2+4+5)+(2+4+5+7)+(2+4+5+7+11)+(2+4+5+7+11+13)=108
-----

C:\Users\David Pastor\Desktop>
```

Figura 7: Fichero de entrada con su resultado mostrado en un archivo de salida junto a la traza

6. Código fuente

En esta sección se muestra el código fuente utilizado para la realización del programa. El código se ha separado en dos paquetes, en uno se crea la estructura utilizada para la ordenación (en este caso montículos implementados por el alumno) y en el otro las demás clases que interfieren en el servicio.

6.1. Paquete de Montículos: heap

6.1.1. Interfaz de la clase montículo: HeapIN

```
1 package heap;
2
3 public interface HeapIN<T extends Comparable<T>>
4 {
5     /**
6      * Montículo vacío
7      * @return true si el montículo está vacío
8      */
9     public boolean isEmpty();
10
11     /**
12      * Primero
13      * Devuelve el elemento mayor/menor
14      * @return El elemento mayor/menor
15      */
16     public T getTop();
17
18     /**
19      * ObtenerCima
20      * Devuelve la cima y la borra,
21      * restaura las propiedades de montículo
22      * @return La cima del montículo
23      */
24     public T top();
25
26     /**
27      * Inserta un elemento en el montículo
28      * se inserta al fondo y se flota hasta su posición
29      * @param e El elemento a insertar
30      * @return el montículo con el nuevo elemento
31      */
32     public HeapIN<T> insert( T e );
33
34     /**
35      * Devuelve el número de elementos que contiene
36      * el montículo
37      * @return número de elementos del montículo
38      */
39     public int getNumElem();
40 }
```

6.1.2. Montículo de mínimos: HeapMin

```

1 package heap;
2
3 import java.util.ArrayList;
4
5
6 /**
7  * Implementa la clase de montículo de mínimos
8  *
9  * @author David Pastor
10 */
11 public class HeapMin<T extends Comparable<T>> implements HeapIN<T>
12 {
13     private ArrayList<T> vec;
14     private int numElem;
15     private int MAX;
16
17     public HeapMin( int max )
18     {
19         this.vec = new ArrayList<T>( max );
20         this.numElem = 0;
21         this.MAX = max;
22     }
23
24     /**
25     * Crear montículo lineal
26     * Crea un montículo a partir de los elementos de
27     * un vecto mediante el procedimiento hundir
28     * @param v El vector mediante el cual se crea el montículo
29     */
30     public HeapMin( ArrayList<T> v )
31     {
32         if( v != null )
33         {
34             this.MAX = v.size();
35             this.vec = v;
36             this.numElem = v.size();
37             for( int i = v.size()/2; i > -1; i-- )
38                 sink( i, this.vec );
39         }
40     }
41
42     /**
43     * Montículo vacío
44     * @return true si el montículo está vacío
45     */
46     public boolean isEmpty()
47     {
48         boolean empty = false;
49         if( this.numElem == 0 ) empty = true;
50         return empty;
51     }
52
53     /**
54     * Hundir
55     * Reubica el elemento n-ésimo, en caso de que sea
56     * mayor/menor que alguno de sus hijos
57     * @param n El elemento del vector a hundir
58     */
59     private void sink( int n, ArrayList<T> v )
60     {
61         int leftChild, rightChild, root;
62         if( this.numElem > 1 )

```

```

63     {
64         do
65         {
66             leftChild = n*2+1;
67             rightChild = n*2+2;
68             root = n;
69
70             if( rightChild < this.numElem && v.get( rightChild ).compareTo( v.get( n ) ) < 0
71                 )
72                 n = rightChild;
73             if( leftChild < this.numElem && v.get( leftChild ).compareTo( v.get( n ) ) < 0 )
74                 n = leftChild;
75
76             change( root, n, v );
77         }
78         while( root != n );
79     }
80 }
81
82 /**
83  * Flotar
84  * Reubica el elemento n-ésimo, en caso de que sea
85  * mayor/menor que alguno de sus hijos
86  * Se utiliza para introducir elementos
87  * @param n El elemento del vector a flotar
88  */
89 private void hover( int n, ArrayList<T> v )
90 {
91     while( n > 0 && v.get( n/2 ).compareTo( v.get( n ) ) < 0 )
92     {
93         change( n, n/2, v );
94         n /= 2;
95     }
96 }
97
98 private void change( int son, int father, ArrayList<T> v )
99 {
100     T auxF = v.get( father );
101     v.set( father, v.get( son ) );
102     v.set( son, auxF );
103 }
104
105 /**
106  * Primero
107  * Devuelve el elemento mayor/menor
108  * @return El elemento mayor/menor
109  */
110 public T getTop()
111 {
112     T e = null;
113     if( this.numElem != 0 )
114         e = vec.get( 0 );
115     return e;
116 }
117
118 /**
119  * ObtenerCima
120  * Devuelve la cima y la borra,
121  * restaura las propiedades de montículo
122  * @return La cima del montículo
123  */
124 public T top()
125 {
126     T top = null;

```

```

126     if( this.numElem != 0 )
127     {
128         top = this.vec.get( 0 );
129         this.vec.set( 0, this.vec.get( this.numElem-1 ) );
130         this.vec.remove( this.numElem-1 );
131         this.numElem -= 1;
132         sink( 0, this.vec );
133     }
134     return top;
135 }
136
137
138 /**
139  * Inserta un elemento en el montículo
140  * se inserta al fondo y se flota hasta su posición
141  * @param e El elemento a insertar
142  * @return el montÁculo con el nuevo elemento
143  */
144 public HeapIN<T> insert( T e )
145 {
146     if( e != null )
147     {
148         if( this.isEmpty() )
149         {
150             this.numElem += 1;
151             vec.add( 0, e );
152         }
153         else if( this.MAX > this.numElem )
154         {
155             this.numElem += 1;
156             this.vec.add(numElem-1, e );
157             hover( numElem-1, this.vec );
158         }
159     }
160     return this;
161 }
162
163 /**
164  * Devuelve el número de elementos que contiene el montículo
165  * @return el número de elementos del montículo
166  */
167 public int getNumElem()
168 {
169     return numElem;
170 }
171
172 @Override
173 public String toString()
174 {
175     StringBuffer sb = new StringBuffer();
176     sb.append( "HeapMin [ " );
177     int it = this.numElem;
178     for( int i = 0; i < numElem; i++ )
179     {
180         sb.append( vec.get( i ) );
181         if( i+1 != it ) sb.append( ", " );
182     }
183     sb.append( " ]" );
184     return sb.toString();
185 }
186 }

```


6.2. Paquete servidor: server

6.2.1. Clase de lectura de archivo de entrada: Reader

```
1 package server;
2
3 import java.io.File;
4 import java.io.FileReader;
5 import java.io.BufferedReader;
6 import java.io.IOException;
7 import java.io.FileNotFoundException;
8 import java.lang.NumberFormatException;
9 import java.util.ArrayList;
10
11 /**
12  * Se encarga de leer un archivo y verificar que la entrada es correcta
13  *
14  * @author David Pastor Sanz
15  */
16 public abstract class Reader
17 {
18     /**
19      * Devuelve un ArrayList de clientes a partir de un archivo de
20      * entrada, si este es correcto
21      * @return El array de clientes
22      */
23     public static ArrayList<Client> getClientList( String file ) throws
24         FileNotFoundException, IOException, IllegalStateException, NumberFormatException
25     {
26         File rFile = new File( file );
27         ArrayList<Client> clientList = new ArrayList<Client>();
28         try
29         {
30             if( rFile.exists() )
31             {
32                 try
33                 {
34                     FileReader fileReader = new FileReader( rFile );
35                     BufferedReader bufferedReader = new BufferedReader( fileReader );
36                     String firstLine = bufferedReader.readLine();
37                     int numOfClients = checkFirstLine( firstLine );
38                     String waitingTime = bufferedReader.readLine();
39                     clientList = checkSecondLine( waitingTime, numOfClients );
40                     moreLines( bufferedReader );
41                     bufferedReader.close();
42                 }
43                 catch( IOException e )
44                 {
45                     throw( e );
46                 }
47             }
48             else throw new FileNotFoundException();
49         }
50         catch( FileNotFoundException ef )
51         {
52             throw new FileNotFoundException( "—Fichero no encontrado—" );
53         }
54         return clientList;
55     }
56
57     /**
58      * Comprueba si la primera linea del archivo contiene un entero
59      * que indica el número de clientes a la espera y lo devuelve si
```

```

59  * es correcto
60  * @param numOfClients El String leído con el buffer del archivo
61  * @return el número de clientes
62  * @throws IOException Indica que o no se ha detectado nada o que no se
63  *       ha introducido un entero
64  */
65  private static int checkFirstLine( String numOfClients ) throws IOException,
        NumberFormatException
66  {
67      if( numOfClients == null || numOfClients.isEmpty() ) throw new
        NumberFormatException( "—No se han encontrado datos de lectura—" );
68      int numOfClient = 0;
69      try
70      {
71          numOfClient = Integer.parseInt( numOfClients );
72      }
73      catch( NumberFormatException e )
74      {
75          throw new NumberFormatException( "—El número de clientes debe ser un entero
        natural—" );
76      }
77      return numOfClient;
78  }
79
80  /**
81  * Comprueba que la segunda linea del archivo contiene los tiempos de espera de los
82  * clientes acorde al número de clientes, si es correcto devuelve un ArrayList
83  * de tipo cliente con la posición del cliente y su tiempo de espera
84  * @param waitingTime String leído con el buffer del archivo
85  * @param numClients número de clientes
86  * @return la lista de los clientes
87  * @throws IOException Indica o que no se ha detectado nada o que los tiempos de
88  *       espera no son acordes con el número de clientes
89  */
90  private static ArrayList<Client> checkSecondLine( String waitingTime, int numClients
        ) throws IOException, NumberFormatException
91  {
92      ArrayList<Client> server = new ArrayList<Client>();
93      try
94      {
95          if( waitingTime == null || waitingTime.equals( "" ) ) throw new
        IllegalStateException( "—No se dispone de ningún dato para el tiempo de
        espera de clientes—" );
96          String[] aux = waitingTime.split( " " );
97          if( numClients != aux.length ) throw new NumberFormatException( "—Número de
        tiempos de espera no acorde con número de clientes—" );
98
99          for( int i = 0; i < aux.length; i++ )
100          {
101              try
102              {
103                  int time = Integer.parseInt( aux[i] );
104                  if( time <= 0 ) throw new NumberFormatException();
105                  server.add( new Client( i+1, time ) );
106              }
107              catch( NumberFormatException e )
108              {
109                  throw new NumberFormatException( "—Los tiempos de espera deben de ser
        enteros naturales mayores o iguales que 0—" );
110              }
111          }
112      }
113      catch( NumberFormatException | IllegalStateException ie )
114      {

```

```

115         throw ( ie );
116     }
117
118     return server;
119 }
120
121 /**
122  * Comprueba que no se encuentre nada más en el archivo
123  * @param br el Buffer crado a partir del archivo
124  * @throws IOException Indica que se han introducido datos de más
125  */
126 private static void moreLines( BufferedReader br ) throws IOException
127 {
128     if( br.readLine() != null )
129         throw new IllegalStateException( "—Se han introducido datos de más en el archivo
130         —" );
131 }

```

6.2.2. Clase servidor (contiene el algoritmo voraz): Server

```

1 package server;
2
3 import java.util.ArrayList;
4 import heap.HeapIN;
5 import heap.HeapMin;
6
7 /**
8  * Esta clase se encarga de realizar el esquema voraz
9  * Encuentra la solución más optima de entre todos los candidatos
10  *
11  * @author David Pastor
12  *
13  */
14 public abstract class Server
15 {
16     /**
17      * De un conjunto de clientes , con sus respectivos tiempos de servicios ,
18      * se obtiene el menor tiempo de estancia en el servicio para todos.
19      * Primero se introducen los candidatos en un montículo de mínimos , después mientras
20      * el montículo siga teniendo candidatos se realizaá un ordenamiento por montículos
21      * y se irán obteniendo los candidatos en orden de menor a mayor y se irán
22      * introduciendo en la solución
23      * @param clients el conjunto de clientes
24      * @return Un array con el tiempo total de servicio , orden de los clientes y
25      *         la traza.
26      */
27     public static String [] timeMinimalize( ArrayList<Client> clients )
28     {
29         String orderClients = new String();
30         String [] sol = new String [3];
31         HeapIN<Client> h = new HeapMin<Client>( clients );
32         int time = 0;
33         int totalTime = 0;
34         String aux = "";
35         String aux2 = "";
36         String trace = "";
37
38         while( !h.isEmpty() )
39         {
40             Client c = h.top();

```

```

41     orderClients += c.getPosition() + " ";
42     time += c.getWaitingTime();
43
44     if( aux == "" )
45     {
46         aux += c.getWaitingTime();
47         aux2 += aux;
48     }
49     else
50     {
51         aux += "+" + c.getWaitingTime();
52         aux2 += "(" + aux + ")";
53     }
54     trace = drawTrace( trace, aux, time, c.getPosition(), true );
55     totalTime += time;
56 }
57 trace = drawTrace( trace, aux2, totalTime, 0, false );
58 sol[0] = totalTime + "";
59 sol[1] = orderClients;
60 sol[2] = trace;
61 return sol;
62 }
63
64 private static String drawTrace( String t, String n, int time, int client, boolean
65     cont )
66 {
67     if( t.isEmpty() )
68     {
69         t += "_____\\r\\n";
70         t += " Cliente || Tiempo de Espera de Cada Cliente \\r\\n";
71         t += "_____\\r\\n";
72         t += String.format( " %4d      || %s = %d\\r\\n", client, n, time );
73     }
74     else if( cont )
75         t += String.format( " %4d      || %s = %d\\r\\n", client, n, time );
76     else if( !cont )
77         t += "_____\\r\\n" + " Traza: "
78             + n + "=" + time + "\\r\\n_____\\r\\n";
79     return t;
80 }
81 }

```

6.2.3. Clase cliente: Client

```

1 package server;
2
3 /**
4  * Clase cliente que almacena la posición del cliente en la lista de espera
5  * y el tiempo que necesitará ser atendido
6  * Se compone de getter y setters además de extender la clase Comparable
7  * e implementar su método compareTo
8  *
9  * @author David Pastor
10  *
11  */
12 public class Client implements Comparable<Client>
13 {
14     private int position;
15     private int waitingTime;
16

```

```
17 public Client( int position , int waitingTime )
18 {
19     this.position = position;
20     this.waitingTime = waitingTime;
21 }
22
23 public int getPosition()
24 {
25     return position;
26 }
27
28 public void setPosition( int position )
29 {
30     this.position = position;
31 }
32
33 public int getWaitingTime()
34 {
35     return waitingTime;
36 }
37
38 public void setWaitingTime( int waitingTime )
39 {
40     this.waitingTime = waitingTime;
41 }
42
43 @Override
44 public String toString()
45 {
46     return "( " + position + ", " + waitingTime + " )";
47 }
48
49 /**
50  * Comparador de la clase cliente.
51  * Compara dos objetos de la clase cliente por su tiempo
52  * necesario en servicio
53  * @param el cliente a comparar con this
54  * @return devuelve 1 si this tiene mayor tiempo que c, -1 si tiene
55  *         menor tiempo y 0 si son iguales
56  */
57 public int compareTo( Client c )
58 {
59     if( this.waitingTime > c.waitingTime ) return 1;
60     else if( this.waitingTime < c.waitingTime ) return -1;
61     else return 0;
62 }
63 }
```

6.2.4. Clase controladora, contiene el main: Controller

```
1 package server;
2
3 import java.util.ArrayList;
4 import java.io.IOException;
5 import java.io.FileNotFoundException;
6 import java.io.File;
7 import java.io.PrintStream;
8 import java.io.FileOutputStream;
9 import java.io.FileDescriptor;
10
11 /**
```

```

12  * Controla las demás clases y contiene el método main del programa
13  * @author David Pastor
14  *
15  */
16  public class Controller
17  {
18      public static void main( String[] args )
19      {
20          try
21          {
22              getArguments( args );
23          }
24          catch( IOException | IllegalStateException | NumberFormatException e )
25          {
26              System.out.println( e );
27          }
28      }
29
30      private static void getArguments( String[] args ) throws IOException
31      {
32          try
33          {
34              ArrayList<Client> cl = new ArrayList<Client>();
35              if( args.length == 0 )
36                  System.out.println( "No se han detectado parámetros" );
37              if( args.length == 1 )
38              {
39                  if ( args[0].equals( "-h" ) ) System.out.println( help() );
40                  else System.out.println( resultWithOutTrace( cl, args[0] ) );
41              }
42              else if( args.length == 2 )
43              {
44                  if( args[0].equals( "-t" ) ) System.out.println( resultWithTrace( cl, args[1] ) );
45                  else if( args[0].equals( "-h" ) ) throw new IOException( "—Parámetros mal
46                      introducidos, introduzca -h para ayuda—" );
47                  else
48                  {
49                      printInFile( args[1] );
50                      System.out.println( resultWithOutTrace( cl, args[0] ) );
51                      System.setOut( new PrintStream( new FileOutputStream( FileDescriptor.out ) ) );
52                      System.out.println( "Salida de archivo" );
53                  }
54              }
55              else if( args.length == 3 )
56              {
57                  if( args[0].equals( "-t" ) )
58                  {
59                      printInFile( args[2] );
60                      System.out.println( resultWithTrace( cl, args[1] ) );
61                      System.setOut( new PrintStream( new FileOutputStream( FileDescriptor.out ) ) );
62                      System.out.println( "Salida de archivo" );
63                  }
64                  else throw new IOException( "—Parámetros mal introducidos, introduzca -h para
65                      ayuda—" );
66              }
67          }
68          catch( IOException | IllegalStateException | NumberFormatException e )
69          {
70              System.out.println( e );
71          }
72      }
73  }

```

```

71
72 private static String resultWithTrace( ArrayList<Client> cl, String arg ) throws
    IOException
73 {
74     cl = Reader.getClientList( arg );
75     String[] sol = Server.timeMinimalize( cl );
76     return sol[0] + "\r\n" + sol[1] + "\r\n" + sol[2];
77 }
78
79 private static String resultWithoutTrace( ArrayList<Client> cl, String arg ) throws
    IOException
80 {
81     cl = Reader.getClientList( arg );
82     String[] sol = Server.timeMinimalize( cl );
83     return sol[0] + "\n" + sol[1];
84 }
85
86 private static String help()
87 {
88     StringBuffer sb = new StringBuffer();
89
90     sb.append( "SINTAXIS:\n" );
91     sb.append( "servicio [-t] [-h] [fichero_entrada] [fichero_salida]\n" );
92     sb.append( "-t                                Traza la selección de clientes\n" );
93     sb.append( "-h                                Muestra esta ayuda\n" );
94     sb.append( "fichero_entrada          Nombre del fichero de entrada\n" );
95     sb.append( "fichero_salida           Nombre del fichero de salida\n" );
96
97     return sb.toString();
98 }
99
100
101 private static void printInFile( String fileName ) throws IOException
102 {
103     File file = new File( fileName );
104     if( file.exists() && file.isFile() ) throw new IOException( " —El archivo de salida
        ya existe—" );
105     PrintStream out = null;
106     try
107     {
108         out = new PrintStream( new FileOutputStream( fileName ) );
109     }
110     catch ( FileNotFoundException e )
111     {
112         System.out.println( e + " —No se ha encontrado la ruta del archivo—" );
113     }
114     System.setOut( out );
115 }
116 }

```

Para la realización de esta práctica el alumno se ha basado en la bibliografía básica de la asignatura.

El programa ha sido escrito en el lenguaje de programación Java mediante el IDE *eclipse*.

Este documento ha sido realizado en \LaTeX con el editor de textos *TexMaker*.