

Universidad Nacional de Educación a Distancia

Práctica de Programación III
(Curso 2004 – 2005)

Roberto Hernando Velasco

6 de enero de 2005

Índice

I Prólogo	2
II Memoria de la práctica	5
1. El comercial estresado	6
1.1. Enunciado	6
1.2. Estudio del problema y diseño algorítmico	6
1.2.1. Detección del esquema algorítmico	6
1.2.2. Particularización del algoritmo sobre el problema concreto	9
1.3. Implementación y prueba	12
1.3.1. Implementación del algoritmo	12
1.3.2. Implementación de un programa principal	15
1.3.3. Ejecución y pruebas	18
2. El reponedor responsable	21
2.1. Enunciado	21
2.2. Estudio del problema y diseño algorítmico	22
2.2.1. Detección del esquema algorítmico	22
2.2.2. Particularización del algoritmo sobre el problema completo	23
2.3. Implementación y prueba	26
2.3.1. Implementación del algoritmo	26
2.3.2. Implementación de un programa principal	29
2.4. Ejecución y pruebas	32
3. Maruja la ahorradora	34
3.1. Estudio del problema y diseño algorítmico	34

3.1.1. Detección del esquema algorítmico	34
3.1.2. Particularización del algoritmo sobre el problema concreto	36

Parte I

Prólogo

Introducción

Documentación entregada

1. Hoja de lectura óptica.
2. Memoria de la práctica (éste documento).
3. CD conteniendo:
 - Fichero LEAME.TXT donde se detalla la estructura del CD.
 - Código fuente y ficheros de entrada.
 - Enunciado de la práctica.
 - Memoria de la práctica (éste documento).
 - Programas compilados para Linux x86.
 - Compilador para Linux XDS.

Observaciones

- A la fecha de término de la práctica y de esta documentación el equipo docente todavía no había provisto del juego de pruebas, por lo que los programas se han probado con un conjunto de pruebas propio.
- De los dos primeros bloques de trabajo (obligatorios) se han realizado las dos tareas propuestas – *Estudio del problema y diseño algorítmico e Implementación y prueba*–, mientras que del bloque III (voluntario) sólo se ha realizado la primera tarea propuesta –*Estudio del problema y diseño algorítmico*–.

Compilación y ejecución

Compilación

Si se desea compilar el código fuente de cualquiera de los programas se hará de la forma:

```
xds =m MAIN.mod
```

donde, previamente, se habrán añadido los ejecutables de XDS (por defecto en `/usr/local/xds/bin`) al PATH.

Ejecución

Todos los programas se ejecutan de la forma:

```
./MAIN
```

y cuando pida indicar el nombre del fichero de entrada se escribirá FICHERO.DAT donde FICHERO.DAT será un fichero de texto plano con la estructura adecuada a cada problema.

Parte II

Memoria de la práctica

Bloque 1

El comercial estresado

1.1. Enunciado

Fernando Romero, un comercial de Productos Alimenticios S.L. recorre todos los lunes de cada semana varios hipermercados entrevistándose con sus directores comerciales con el objeto de vender sus productos. La cartera de clientes de que dispone es amplia y se caracteriza porque si el comercial no visita a un cliente éste no realiza su pedido semanal, siendo fija la cantidad de dinero que semanalmente gasta cada uno de ellos. Además, Fernando tiene un problema que consiste en que todos los directores comerciales que éste visita comienzan a trabajar a las 8:00 de la mañana, pero nunca cumplen sus 8 horas de jornada (por algo son jefes), abandonando su puesto de trabajo sin esperar a nuestro comercial. Fernando, que sí que cumple con su jornada laboral completa, sabe que en sus 8 horas de jornada laboral de los lunes (que comienza a las 8:00 de la mañana) debe visitar a aquellos clientes que más dinero se gastan en sus productos, siendo la duración de las reuniones que mantiene con cada uno de ellos de una hora. Con este objetivo, y suponiendo que el tiempo que tarda Fernando en desplazarse de un hipermercado a otro es despreciable, se pide diseñar e implementar un algoritmo que planifique las visitas de Fernando a lo largo de sus 8 horas de jornada laboral de los lunes, maximizando el beneficio obtenido de sus clientes.

El algoritmo que el alumno debe diseñar e implementar deberá mostrar como resultado la secuencia de centros a visitar por el comercial, en el orden en que éstos deben ser visitados, así como el beneficio total (en miles de euros) que obtendría Fernando realizando ese recorrido.

1.2. Estudio del problema y diseño algorítmico

1.2.1. Detección del esquema algorítmico

1. – *¿Por qué el esquema algorítmico más adecuado para solucionar el problema es un esquema voraz? Razone porqué otras aproximaciones no serían válidas.*

El esquema algorítmico a aplicar es el **voraz** ya que se trata de un problema de optimización en el que se distingue un conjunto de candidatos (los centros a visitar) que tenemos que escoger en cierto orden.

La aproximación mediante el esquema **divide y vencerás** no es apropiada ya que el problema no se puede dividir en subproblemas más pequeños del mismo tipo a los que aplicar el mismo procedimiento.

El esquema de **exploración de grafos** no tiene interés en nuestro caso, porque nos interesa examinar los nodos en cierto orden.

2. – *Presente y explique el esquema algorítmico voraz, describiendo la funcionalidad de los distintos elementos del mismo (funciones auxiliares).*

El algoritmo voraz (cuyo esquema se muestra a continuación) consiste en ir seleccionando elementos de un conjunto de candidatos que se van incorporando a la solución. Los candidatos desechados no vuelven a ser considerados, y los incorporados a la solución permanecen en ella hasta el final del algoritmo.

Esquema voraz

función *voraz* (C : conjunto) **devuelve** (S : conjunto)

- 1: $\{C$ es el conjunto de candidatos}
- 2: $S \leftarrow \emptyset$ {Construimos la solución en el conjunto S }
- 3: **mientras** $C \neq \emptyset$ **y no** *solución* (S) **hacer**
- 4: $x \leftarrow$ *seleccionar* (C)
- 5: $C \leftarrow C \setminus \{x\}$
- 6: **si** *factible* ($S \cup \{x\}$) **entonces**
- 7: $S \leftarrow S \cup \{x\}$
- 8: **fin si**
- 9: **fin mientras**
- 10: **si** *solución* (S) **entonces**
- 11: **devolver** S
- 12: **fin si**

fin función *voraz*

El algoritmo voraz busca un conjunto de candidatos que constituya una solución y que optimice el valor de la función objetivo. Inicialmente el conjunto de elementos seleccionados (S) está vacío. En cada paso se añade a este conjunto el mejor candidato. Si el conjunto ampliado de candidatos ya no fuera factible, rechaza el candidato seleccionado, sino lo añade. Cada vez que se amplía el conjunto de candidatos, comprueba si constituye una solución para el problema. La primera solución que se encuentre de esta manera es óptima.

Las funciones auxiliares del algoritmo son *solución*, *seleccionar* y *factible*:

- La función *solución* (líneas 3 y 10) decide si el conjunto S donde se almacenan los candidatos seleccionados es o no una solución.
- La función *seleccionar* (línea 4) selecciona en cada iteración del bucle el mejor candidato de entre los restantes (de C) según la función *objetivo*.
- La función *factible* (línea 6) determina si un conjunto de elementos seleccionados puede llegar a ser una solución o no.

3. – En las páginas 237 y 240 del texto base se presentan dos algoritmos basados en el esquema voraz que resuelven de manera directa el problema propuesto. ¿Con cuál de los dos se quedaría? ¿Por qué?

En este caso particular con el algoritmo de la página 237.

Aunque el algoritmo de la página 240 tiene un coste menor asintóticamente, como el tamaño del problema es acotado (y, de hecho, muy pequeño), prevalece en la elección la facilidad de implementación que tiene el algoritmo de la página 237 sobre el de la página 240.

4. – Presente y explique el algoritmo seleccionado comparando sus diferentes elementos (funciones auxiliares y bloques de código) con los elementos genéricos identificados y explicados en el punto 2 al exponer el esquema voraz.

Algoritmo seleccionado

función *secuencia* ($d[0 \dots n]$: vector) **devuelve** (k : entero, $j[1 \dots k]$: vector)

```
1: vector  $j[0 \dots n]$ 
2:  $d[0] \leftarrow 0$ ;  $j[0] \leftarrow 0$  {centinelas}
3:  $k \leftarrow 1$ ;  $j[1] \leftarrow 1$  {la tarea 1 siempre se selecciona}
4:
5: {Bucle voraz}
6: para  $i = 2$  hasta  $n$  hacer
7:    $r \leftarrow k$ 
8:   mientras  $d[[j[r]]] > \text{máx}(d[i], r)$  hacer
9:      $r \leftarrow r - 1$ 
10:  fin mientras
11:  si  $d[i] > r$  entonces
12:    para  $m = k$  paso  $-1$  hasta  $r + 1$  hacer
13:       $j[m + 1] \leftarrow j[m]$ 
14:    fin para
15:  fin si
16:   $j[r + 1] \leftarrow i$ 
17:   $k \leftarrow k + 1$ 
18: fin para
19:
20: devolver  $k, j[1 \dots k]$ 
```

fin función *secuencia*

Se supone que las tareas están numeradas de forma que $g_1 \geq g_2 \geq \dots \geq g_n$, con $n > 0$. Se supone además que el conjunto de candidatos verifica que $d_i > 0$ para $1 \leq i \leq n$. La solución se construye en el vector j .

También se supone un espacio adicional al principio de d y j para almacenar los centinelas (línea 2 del algoritmo).

Las k tareas de j están por orden creciente de plazo. Cuando se está considerando la tarea i , el algoritmo comprueba si se puede insertar en j en el lugar oportuno sin llevar alguna tarea que ya esté en j más allá de su plazo. Si se puede, i se acepta; en caso contrario i se rechaza.

Los valores exactos de las tareas g_i no son necesarios siempre y cuando las tareas estén numeradas decrecientemente.

Los vectores d y j del algoritmo seleccionado corresponden respectivamente con los conjuntos C (candidatos) y S (solución) del algoritmo voraz.

- La función *solución* del esquema voraz la tenemos en la línea 8 del algoritmo seleccionado. No tenemos la solución mientras $d[[j[r]]] > \text{máx}(d[i], r)$.
- La función *factible* corresponde a la línea 11: $d[i] > r$.
- La función *seleccionar* correspondería a las líneas 16 y 17: $j[r + 1] \leftarrow i, k \leftarrow k + 1$.

El bucle voraz del algoritmo seleccionado (líneas 6 a 18) corresponde a las líneas 3 a 10 del algoritmo voraz.

1.2.2. Particularización del algoritmo sobre el problema concreto

5. – ¿Cuáles son las estructuras de datos más adecuadas para resolver el problema? Exponga y explique las estructuras de datos principales para resolver el problema, así como aquellas estructuras de datos auxiliares necesarias para almacenar la información de cálculos intermedios del algoritmo.

Las estructuras de datos más adecuadas son vectores (arrays). Sin embargo, a priori, no conocemos el tamaño de los vectores; por lo que se utilizará una estructura de datos del tipo lista.

Para los datos principales se utilizará la siguiente estructura:

```

tipo  tipoDato  =  registro
        tamaño   :  entero
        indice   :  vector [1 ... longMax] de enteros
        ganancia :  vector [1 ... longMax] de reales
        horas    :  vector [1 ... longMax] de enteros
    
```

tamaño indica el número de supermercados leídos. *ganancia* y *horas* se utilizan para guardar dicha información de cada supermercado (según *indice*). Donde *longMax* es una constante que nos da el máximo número de supermercados con el que vamos a trabajar (en este problema es 50).

Además, para devolver la solución del problema se utilizará la siguiente estructura:

```

tipo  solucion =  registro
        tamaño   :  entero
        indice   :  vector [1 ... longMax] de enteros
    
```

ya que en la solución sólo nos interesa saber qué supermercados y en qué orden visitar, sin ningún dato más. Las demás estructuras de datos a utilizar son estructuras elementales.

6. – *Adapte el esquema algorítmico seleccionado a su problema concreto. Especifique, diseñe y explique todas aquellas funciones auxiliares necesarias para la resolución del mismo.*

No habría que hacer ninguna modificación en el algoritmo. Simplemente habría que ordenar previamente las tareas g_i decrecientemente; podríamos utilizar, por ejemplo, el algoritmo de ordenación por inserción, ya que tenemos muy pocos elementos a ordenar, y prima la facilidad de implementación sobre el coste asintótico.

7. – *Presente el diseño del algoritmo completo.*

Algoritmo completo

```
función ordena ( $T[0 \dots n]$ : vector)
  {Ordenación decreciente por inserción}
para  $i = 1$  hasta  $n$  hacer
   $x \leftarrow T[i]$ ;  $j \leftarrow i - 1$ 
  mientras  $j > -1$  y  $x > T[j]$  hacer
     $T[j + 1] \leftarrow T[j]$ 
     $j \leftarrow j - 1$ 
  fin mientras
   $T[j + 1] \leftarrow x$ 
fin para
fin función ordena
```

.....

función *secuencia* ($d[0 \dots n]$: vector) **devuelve** (k : entero, $j[1 \dots k]$: vector)

vector $j[0 \dots n]$

$d[0] \leftarrow 0; j[0] \leftarrow 0$ {centinelas}

$k \leftarrow 1; j[1] \leftarrow 1$ {la tarea 1 siempre se selecciona}

{Bucle voraz}

para $i = 2$ hasta n **hacer**

$r \leftarrow k$

mientras $d[[j[r]]] > \text{máx}(d[i], r)$ **hacer**

$r \leftarrow r - 1$

fin mientras

si $d[i] > r$ **entonces**

para $m = k$ paso -1 hasta $r + 1$ **hacer**

$j[m + 1] \leftarrow j[m]$

fin para

$j[r + 1] \leftarrow i$

$k \leftarrow k + 1$

fin si

fin para

devolver $k, j[1 \dots k]$

fin función *secuencia*

.....

función *principal* ($d[0 \dots n]$: vector)

{Función Principal}

ordena ($d[0 \dots n]$)

secuencia ($d[0 \dots n]$)

fin función *ordena*

8. – Realice un estudio teórico de la complejidad del algoritmo diseñado. Para ello deberá calcular y presentar separadamente el coste de todas y cada una de las funciones auxiliares implementadas en el algoritmo y, finalmente, calcular y presentar el coste total del algoritmo completo.

- ¿Podría mejorarse la eficiencia del algoritmo obtenido? En caso afirmativo explicar cómo.

1. Coste de la función *ordena* (algoritmo de ordenación por inserción decreciente):

Para un i fijo sea $x = T[i]$. El caso peor se da cuando $x > T[j], \forall 1 \leq j \leq i - 1$, ya que en este caso se compara x con $T[1], T[2], \dots, T[i - 1]$ antes de salir del bucle **mientras** (cuando $j = 0$); luego el bucle **mientras** se efectúa i veces en el caso peor.

El caso peor se da para todos los valores entre 2 y n cuando el vector está ordenado de forma creciente, en este caso la comprobación se realiza $\sum_{i=2}^n i = n(n + 1)$ veces, lo que está en $\Theta(n^2)$.

Luego la función *ordena* tiene un coste de $\Theta(n^2)$.

2. Coste de la función *secuencia* (algoritmo voraz):

El bucle **para** del algoritmo voraz se recorre exactamente $n - 1$ veces. El caso peor se da cuando el algoritmo vuelve a clasificar las tareas por orden decreciente de plazos, y cuando todas ellas tienen cabida en la planificación. En este caso, cuando se está considerando la tarea i el algoritmo examina las $k = i - 1$ tareas que ya están planificadas, para encontrar un lugar para la nueva tarea, y después desplaza todas un lugar.

Para ello se realizan $\sum_{k=1}^{n-1} k$ pasadas por el bucle **mientras** y $\sum_{m=1}^{n-1} m$ pasadas por el bucle **para** interno. Por tanto el algoritmo requiere un tiempo que está en $\Omega(n^2)$.

Luego el coste total del algoritmo será de $\Omega(n^2)$.

Se puede mejorar la eficiencia del algoritmo de varias formas:

1. Se podría utilizar un algoritmo de ordenación más eficiente, de forma que la ordenación tuviera un coste de $\Theta(n \log n)$.
2. Se podría cambiar la implementación de las estructuras de datos. Si para el conjunto de candidatos en lugar de una lista utilizáramos un montículo, de forma que el supermercado de mayor ganancia siempre estuviera en la raíz, el coste de seleccionar el candidato sería constante. De esta forma se conseguiría un algoritmo de coste $\Omega(n \log n)$.

Combinando ambas opciones tendríamos un algoritmo con un coste total de $\Omega(n \log n)$. Véase que éste es el coste que obtendríamos con el algoritmo de la página 240.

1.3. Implementación y prueba

1.3.1. Implementación del algoritmo

9. – Cree un módulo de nombre *COMERCIAL*, que contenga una función, de nombre *PLANFIJO*, que implemente el algoritmo diseñado y que devuelva un parámetro con la solución al problema.

- Incluya en el módulo *COMERCIAL* todas aquellas funciones auxiliares necesarias para el correcto funcionamiento de la función *PLANFIJO*.
- Se podrá suponer que como máximo la cartera de clientes del comercial va a contener 50 supermercados diferentes.

Módulo de definición

DEFINITION MODULE COMERCIAL;

CONST longMax = 50; (* número máximo de supermercados *)

```

TYPE tipoDato = RECORD
    tamaño    : INTEGER;
    indice    : ARRAY [1..longMax] OF INTEGER;
    ganancia  : ARRAY [1..longMax] OF REAL;
    horas     : ARRAY [0..longMax] OF INTEGER;
END;
(* Estructura de datos principal del programa.
   Servirá para los bloques de datos leídos. *)

TYPE solucion = RECORD
    tamaño    : INTEGER;
    indice    : ARRAY [0..longMax] OF INTEGER;
END;
(* Estructura de datos en la que se devolverá
   la solución del problema. *)

PROCEDURE Maximo (a : INTEGER; b : INTEGER) : INTEGER;
(* Halla el máximo de dos números enteros *)

PROCEDURE Ordena (VAR T: tipoDato);
(* Ordena un vector de forma decreciente por inserción *)

PROCEDURE PLANFIJO (T: tipoDato): solucion;
(* Implementa el algoritmo principal y devuelve la solución *)

END COMERCIAL.

```

Módulo de implementación

```

IMPLEMENTATION MODULE COMERCIAL;

(***** M A X I M O *****)
PROCEDURE Maximo (a : INTEGER; b : INTEGER) : INTEGER;
(* Halla el máximo de dos números enteros *)
BEGIN
    IF a>b THEN
        RETURN a
    ELSE
        RETURN b
    END;
END Maximo;

(***** O R D E N A *****)
PROCEDURE Ordena (VAR T: tipoDato);
(* Ordena una estructura de tipoDato de forma decreciente,
   según las ganancias, por inserción *)

TYPE tipoTemporal = RECORD
    i : INTEGER;
    g : REAL;
    h : INTEGER;

```

```

                                END;

VAR
    x      :  tipoTemporal; (* temporal *)
    i, j   :  INTEGER; (* contadores *)

BEGIN
    FOR i:=2 TO T.tamano DO
        x.i := T.indice[i];
        x.g := T.ganancia[i];
        x.h := T.horas[i];
        j:=i-1;

        WHILE (j>0) AND (x.g > T.ganancia[j]) DO
            T.indice[j+1] := T.indice[j];
            T.ganancia[j+1] := T.ganancia[j];
            T.horas[j+1] := T.horas[j];
            j:=j-1;
        END; (*end while*)

        T.indice[j+1] := x.i;
        T.ganancia[j+1] := x.g;
        T.horas[j+1] := x.h;
    END; (*end for*)
END Ordena;

(***** PLANFIJO *****)
PROCEDURE PLANFIJO (T: tipoDato): solucion;
(* Implementa el algoritmo voraz. En el componente indice de T
   devuelve la secuencia óptima *)

VAR
    j      :  solucion; (* lista de tareas *)
    i,m    :  INTEGER; (* contadores *)
    r      :  INTEGER; (* variable auxiliar *)

BEGIN
    T.horas[0] := 0; j.indice[0] := 0; (*centinelas*)
    j.tamano := 1; (* número de tareas completadas *)
    j.indice[1] := 1; (* la primera tarea (la de mayor ganancia)
                       siempre se selecciona *)

    (* BUCLE VORAZ *)
    FOR i:=2 TO T.tamano DO
        r:=j.tamano;
        WHILE T.horas[j.indice[r]] > Maximo(T.horas[i],r) DO
            r := r-1;
        END; (*while*)
        IF T.horas[i] > r THEN
            FOR m:=j.tamano TO r+1 BY -1 DO
                j.indice[m+1] := j.indice[m]
            END; (*for*)
            j.indice[r+1] := i;
            j.tamano := j.tamano+1;
        END;
    END;
END PLANFIJO;

```

```

        END; (*if*)
    END; (*for*)

    RETURN j; (* devuelve las tareas a completar *)

END PLANFIJO;

END COMERCIAL.

```

1.3.2. Implementación de un programa principal

10. – Implementar un módulo principal MAIN que:

- a) tomará como entrada de datos un fichero de texto plano con nombre *COMERIN.DAT*, con la información de diferentes grupos de hipermercados que debe visitar el comercial;
 - b) mapeará la entrada leída a partir del fichero de datos sobre las estructuras de datos diseñadas;
 - c) hará las correspondientes llamadas a la función *PLANFIJO*, presentando los resultados obtenidos en la pantalla.
- El fichero *COMERIN.DAT* consta de un número indefinido de bloques de información separados por la etiqueta *FDE*. Cada bloque de información representa un conjunto de hipermercados sobre los que el algoritmo deberá hacer la planificación correcta. La primera línea del bloque representa el índice asociado a cada hipermercado, la segunda representa la ganancia obtenida en cada uno de los hipermercados (en miles de euros) y, finalmente, la tercera línea representa el número de horas que el director comercial del hipermercado se encuentra en la oficina (contadas a partir de las 8:00 de la mañana).
 - Se propone el siguiente ejemplo de formato del fichero de entrada *COMERIN.DAT*:

```

1 2 3 4 5 6 7 8 9 10 11 12
0.5 0.1 1 2 3 0.1 0.2 0.5 7 12 0.05 0.1
8 2 6 3 1 8 2 5 6 7 3 3
<FDE>
1 2 3 4 5 6 7 8
0.5 0.1 0.2 0.5 7 12 0.05 0.1
8 8 2 5 6 7 3 3
<FDE>
1 2 3 4 5 6 7 8 9 10
1 2 3 0.1 0.2 0.5 7 12 0.05 0.1
8 2 6 3 1 5 6 7 3 3
<FDE>

```

Módulo principal

```
MODULE MAIN;

    FROM InOut      IMPORT ReadString, WriteString, WriteInt,
                        WriteLn, OpenInput, CloseInput, Done;
    FROM RealInOut  IMPORT WriteReal;
    FROM Strings    IMPORT Equal;
    FROM RealConv   IMPORT ValueReal;

    FROM COMERCIAL  IMPORT longMax, tipoDato, solucion, Ordena, PLANFIJO;

    VAR
        S      : ARRAY[1..(longMax*3+1)] OF ARRAY[1..longMax] OF CHAR;
                (*variable auxiliar para leer el fichero*)
        i,j     : INTEGER; (*contadores*)
        l       : solucion; (*variable auxiliar*)
        gT      : REAL; (*Ganancia Total*)
        dato    : tipoDato;

    (* LEER FICHERO *)
    PROCEDURE LeeFichero();
    BEGIN
        REPEAT
            WriteLn;
            WriteString("Indicar nombre de Archivo de Entrada
                (p.ejem. COMERIN.DAT):");

            WriteLn;
            OpenInput(""); (*abre el fichero*)
        UNTIL Done;
    END LeeFichero;

    (* IMPRIMIR BLOQUE *)
    PROCEDURE ImprimeBloque(dato: tipoDato);
    (* Imprime un bloque de datos *)
    VAR j : INTEGER; (*contador*)
    BEGIN
        FOR j:=1 TO dato.tamano DO
            WriteString("Supermercado ");WriteInt(dato.indice[j],2);
            WriteString(": ");
            WriteString("Ganancia-> ");WriteReal(dato.ganancia[j],6);
            WriteString("| ");
            WriteString("Horas-> ");WriteInt(dato.horas[j],4);WriteLn;
        END;
    END ImprimeBloque;

    (* IMPRIMIR RESULTADO *)
    PROCEDURE ImprimeResultado(l: solucion; gT: REAL);
    VAR
        j : INTEGER; (*contador*)
    BEGIN
        WriteLn; WriteString("Las visitas se harán en el siguiente orden:");
```

```

WriteLn; WriteLn;
FOR j:=1 TO l.tamano DO
(*las tareas están por orden creciente de plazo*)
  gT := gT + dato.ganancia[l.indice[j]];
  WriteString("                Supermercado ");
  WriteInt(dato.indice[l.indice[j]],2);
  WriteString(" --> ganancia: ");
  WriteReal(dato.ganancia[l.indice[j]],6); WriteLn;
END; (*for*)
WriteLn; WriteString("                La ganancia total es de: ");
WriteReal(gT,8);
WriteLn; WriteString(".....");
WriteLn;
END ImprimeResultado;

(* FUNCION PRINCIPAL *)
BEGIN
  LeeFichero();
  i:=1;
  REPEAT
    (* Lee un bloque *)
    ReadString(S[i]);
    IF Equal(S[i], "<FDE>") THEN (*termina un bloque*)
      dato.tamano := i DIV 3;
      FOR j:=1 TO dato.tamano DO
        dato.indice[j] := TRUNC(ValueReal(S[j]));
        dato.ganancia[j] := ValueReal(S[dato.tamano+j]);
        dato.horas[j] := TRUNC(ValueReal(S[2*dato.tamano+j]));
      END;

      ImprimeBloque(dato);

      (* ORDENACIÓN *)
      Ordena(dato);

      (* ALGORITMO VORAZ *)
      l := PLANFIJO(dato);
      gT := 0.0;

      ImprimeResultado(l,gT);

      i:=0;
      END; (*end if*)
      i:=i+1;
  UNTIL NOT Done; (*Lee del fichero hasta que termina*)

  CloseInput; (*cierra el fichero*)

END MAIN.

```

1.3.3. Ejecución y pruebas

A continuación se muestra la salida del programa para varios ficheros de entrada¹:

1. Para el fichero datos.txt:

```
1 2 3 4 5 6
20 15 10 7 5 3
3 1 1 2 1 2
<FDE>
```

se obtiene:

```
Supermercado 1: Ganancia-> 20.000 | Horas-> 3
Supermercado 2: Ganancia-> 15.000 | Horas-> 1
Supermercado 3: Ganancia-> 10.000 | Horas-> 1
Supermercado 4: Ganancia-> 7.000 | Horas-> 2
Supermercado 5: Ganancia-> 5.000 | Horas-> 1
Supermercado 6: Ganancia-> 3.000 | Horas-> 2
```

Las visitas se harán en el siguiente orden:

```
Supermercado 2 --> ganancia: 15.000
Supermercado 4 --> ganancia: 7.000
Supermercado 1 --> ganancia: 20.000
```

La ganancia total es de: 42.000

.....

2. Para el fichero COMERIN.DAT:

```
1 2 3 4 5 6 7 8 9 10 11 12
0.5 0.1 1 2 3 0.1 0.2 0.5 7 12 0.05 0.1
8 2 6 3 1 8 2 5 6 7 3 3
<FDE>
1 2 3 4 5 6 7 8
0.5 0.1 0.2 0.5 7 12 0.05 0.1
8 8 2 5 6 7 3 3
<FDE>
1 2 3 4 5 6 7 8 9 10
1 2 3 0.1 0.2 0.5 7 12 0.05 0.1
8 2 6 3 1 5 6 7 3 3
<FDE>
```

se obtiene:

```
Supermercado 1: Ganancia-> 0.5000 | Horas-> 8
Supermercado 2: Ganancia-> 0.1000 | Horas-> 2
Supermercado 3: Ganancia-> 1.0000 | Horas-> 6
```

¹incluidos en el CD con el nombre dado

Supermercado 4:	Ganancia->	2.000		Horas->	3
Supermercado 5:	Ganancia->	3.000		Horas->	1
Supermercado 6:	Ganancia->	0.1000		Horas->	8
Supermercado 7:	Ganancia->	0.2000		Horas->	2
Supermercado 8:	Ganancia->	0.5000		Horas->	5
Supermercado 9:	Ganancia->	7.000		Horas->	6
Supermercado 10:	Ganancia->	12.000		Horas->	7
Supermercado 11:	Ganancia->	0.0500		Horas->	3
Supermercado 12:	Ganancia->	0.1000		Horas->	3

Las visitas se harán en el siguiente orden:

Supermercado 5	-->	ganancia:	3.000
Supermercado 7	-->	ganancia:	0.2000
Supermercado 4	-->	ganancia:	2.000
Supermercado 8	-->	ganancia:	0.5000
Supermercado 9	-->	ganancia:	7.000
Supermercado 3	-->	ganancia:	1.0000
Supermercado 10	-->	ganancia:	12.000
Supermercado 1	-->	ganancia:	0.5000

La ganancia total es de: 26.200

.....

Supermercado 1:	Ganancia->	0.5000		Horas->	8
Supermercado 2:	Ganancia->	0.1000		Horas->	8
Supermercado 3:	Ganancia->	0.2000		Horas->	2
Supermercado 4:	Ganancia->	0.5000		Horas->	5
Supermercado 5:	Ganancia->	7.000		Horas->	6
Supermercado 6:	Ganancia->	12.000		Horas->	7
Supermercado 7:	Ganancia->	0.0500		Horas->	3
Supermercado 8:	Ganancia->	0.1000		Horas->	3

Las visitas se harán en el siguiente orden:

Supermercado 3	-->	ganancia:	0.2000
Supermercado 8	-->	ganancia:	0.1000
Supermercado 7	-->	ganancia:	0.0500
Supermercado 4	-->	ganancia:	0.5000
Supermercado 5	-->	ganancia:	7.000
Supermercado 6	-->	ganancia:	12.000
Supermercado 1	-->	ganancia:	0.5000
Supermercado 2	-->	ganancia:	0.1000

La ganancia total es de: 20.450

.....

Supermercado 1:	Ganancia->	1.0000		Horas->	8
Supermercado 2:	Ganancia->	2.000		Horas->	2
Supermercado 3:	Ganancia->	3.000		Horas->	6
Supermercado 4:	Ganancia->	0.1000		Horas->	3
Supermercado 5:	Ganancia->	0.2000		Horas->	1
Supermercado 6:	Ganancia->	0.5000		Horas->	5
Supermercado 7:	Ganancia->	7.000		Horas->	6
Supermercado 8:	Ganancia->	12.000		Horas->	7
Supermercado 9:	Ganancia->	0.0500		Horas->	3

Supermercado 10: Ganancia-> 0.1000 | Horas-> 3

Las visitas se harán en el siguiente orden:

Supermercado	5	-->	ganancia:	0.2000
Supermercado	2	-->	ganancia:	2.000
Supermercado	4	-->	ganancia:	0.1000
Supermercado	6	-->	ganancia:	0.5000
Supermercado	7	-->	ganancia:	7.000
Supermercado	3	-->	ganancia:	3.000
Supermercado	8	-->	ganancia:	12.000
Supermercado	1	-->	ganancia:	1.0000

La ganancia total es de: 25.800

.....

Bloque 2

El reponedor responsable

2.1. Enunciado

Luis Manuel López acaba de ser contratado como reponedor del hipermercado CARROFUL. Día tras día intenta demostrar a sus jefes que es un reponedor eficiente capaz de colocar el género en sus correspondientes secciones en un tiempo record, pero se ve muy a menudo obstaculizado por los pesados de sus compañeros. Efectivamente, en casi la totalidad de los pasillos del hipermercado hay compañeros suyos que, aburridos de mirar las estanterías todo el día y aprovechando la novedad del recién contratado, no dudan en abordarle en su camino para entablar una conversación que retrasa su tarea unos minutos. Ya han pasado varios días y, dado que sus jefes le han avisado de que si no quiere perder su empleo debe repartir los productos más rápido, Luis Manuel ha decidido poner fin a esta situación ideando un método para poder reponer los productos en el menor tiempo posible.

Con este objetivo, y suponiendo que el hipermercado se puede representar como un grafo no dirigido, se pide diseñar e implementar un algoritmo que dado un conjunto de productos a reponer calcule los caminos óptimos (se entiende que en función del tiempo empleado en llegar) que debe seguir Luis Manuel para reponerlos. Suponga que los nodos del grafo representan las diferentes secciones del hipermercado (e.g. lácteos, carne, charcutería, imagen y sonido, etc...), y que dispone de un nodo especial (que denominaremos almacén) del que parte el reponedor cargado con los productos de una categoría concreta a reponer. Tenga también en cuenta que el reponedor únicamente tiene permitido reponer productos de una misma categoría en cada viaje, por lo que cada vez que repone éste debe volver al almacén (también por el camino óptimo) y coger un nuevo tipo de producto para reponer. Los enlaces del grafo representarán los pasillos que unen las diferentes secciones del hipermercado, considerando un peso para cada uno de ellos, que puede ser ≥ 0 , y que representa el número de minutos que el reponedor emplearía en charlar con el empleado que se encuentra en ese pasillo.

2.2. Estudio del problema y diseño algorítmico

2.2.1. Detección del esquema algorítmico

11. – *Compare un esquema algorítmico voraz (Dijkstra) con un esquema basado en exploración de grafos y explique cuál de los dos sería el más apropiado para abordar el problema propuesto.*

Depende del número de secciones en las que repartir productos.

Con Dijkstra vamos a obtener el camino mínimo a todas las secciones. En el caso de un algoritmo de exploración obtendremos el camino mínimo a la sección elegida; si queremos obtener el camino a varias secciones habrá que ejecutar el algoritmo varias veces.

12. – *¿Cuál sería la aproximación óptima en el caso de que el reponedor tuviera que repartir productos en todas las secciones del hipermercado? Razone su respuesta.*

En el caso de tener que repartir productos en todas las secciones la aproximación óptima sería el algoritmo de Dijkstra, ya que el esquema voraz nos da todos los caminos mínimos. Si utilizáramos un algoritmo de exploración habría que ejecutarlo tantas veces como secciones tiene el supermercado, y en cada caso volvería a repetir muchas exploraciones anteriores disparándose el coste del algoritmo.

13. – *¿Y en el caso de que el reponedor tuviera que repartir productos en sólo unas pocas secciones del supermercado? Razone su respuesta.*

En este caso el uso de los dos esquemas algorítmicos (voraz o de exploración) sería similar.

14. – *¿Y en el que caso de que sólo tuviera que repartir productos en una única sección? Razone su respuesta.*

En este caso la aproximación óptima es la del algoritmo de exploración de grafos. No sería necesario aplicar un esquema voraz que optimizara todos los caminos cuando sólo necesitamos uno.

15. – *Seleccionando el algoritmo de Dijkstra como candidato a modelar nuestro problema, presente y explique su esquema algorítmico, describiendo la funcionalidad de los distintos elementos del mismo (funciones auxiliares).*

El algoritmo de Dijkstra utiliza dos conjuntos de nodos: S , que contiene aquellos nodos que ya han sido seleccionados; y C , que contiene el resto de nodos. Para los nodos en S se conoce la distancia desde el origen, no así para los nodos en C .

En un primer momento, S sólo contiene el nodo origen; al término del algoritmo S contendrá todos los nodos del grafo. En cada paso se selecciona de C el nodo cuya distancia al origen sea mínima y se añade a S .

Algoritmo de Dijkstra

función *Dijkstra* ($L[1 \dots n, 1 \dots n]$: matriz) **devuelve** ($D[2 \dots n]$, $P[2 \dots n]$: vectores)

```
1: vector  $D[2 \dots n]$  {longitud de los caminos mínimos}
2: vector  $L[2 \dots n]$  {caminos mínimos}
3:
4: {Inicialización}
5:  $C \leftarrow \{2, 3, \dots, n\}$   $\{S = N \setminus C\}$ 
6:  $P \leftarrow \{1, 1, \dots, 1\}$ 
7: para  $i = 2$  hasta  $n$  hacer
8:    $D[i] \leftarrow L[1, i]$ 
9: fin para
10:
11: {Bucle voraz}
12: repetir  $\{n - 2$  veces $\}$ 
13:    $v \leftarrow$  algún elemento de  $C$  que minimice  $D[v]$ 
14:    $C \leftarrow C \setminus \{v\}$ 
15:   para cada  $w \in C$  hacer
16:     si  $D[w] > D[v] + L[v, w]$  entonces
17:        $D[w] \leftarrow D[v] + L[v, w]$ 
18:        $P[2] \leftarrow v$ 
19:     fin si
20:   fin para
21: fin repetir
22:
23: Devolver  $D, P$ 
fin función Dijkstra
```

2.2.2. Particularización del algoritmo sobre el problema completo

16. – ¿Cuáles son las estructuras de datos más adecuadas para resolver el problema? Exponga y explique las estructuras de datos principales para resolver el problema, así como aquellas estructuras de datos auxiliares necesarias para almacenar la información de cálculos intermedios del algoritmo.

La estructura de secciones del supermercado y de los caminos para ir de una sección a otra viene representada por un grafo no orientado con pesos. Para implementar este grafo se utilizará una *matriz de adyacencia*, en la forma

tipo	<i>grafo</i>	=	registro
	<i>numNodos</i>	:	entero
	<i>seccion</i>	:	vector $[1 \dots longMax]$ de palabras
	<i>adyacente</i>	:	matriz $[1 \dots longMax, 1 \dots LongMax]$ de enteros

En *numNodos* se almacena el número de secciones del supermercado. En *seccion* se almacena el nombre de las secciones a tratar. La matriz *adyacente* contiene las distancias entre las secciones (entre aquellas secciones que no haya camino se establecerá una distancia infinita).

Para almacenar la solución se utilizará la siguiente estructura:

```
tipo grafo      = registro
      distancias : vector [1...longMax] de enteros
      punteros   : vector [1...longMax] de enteros
```

Donde en *distancias* estarán las longitudes del camino más corto que va desde el origen hasta cada nodo del grafo. El vector de *punteros* contiene el número de nodo que precede al actual dentro del camino más corto.

Además se utilizará una estructura del tipo conjunto para el conjunto de candidatos:

```
tipo conjunto = conjunto de enteros
```

17. – *Adapte el esquema algorítmico seleccionado a su problema concreto. Especifique, diseñe y explique todas aquellas funciones auxiliares necesarias para la resolución del mismo.*

Lo único que tenemos que especificar para nuestro problema es la línea 13 del algoritmo, donde tenemos que hallar el elemento v de C que minimice $D[v]$.

18. – *Presente el diseño del algoritmo completo.*

Algoritmo completo

```
función MinVector ( $U[2..n]$ : vector) devuelve ( $k$  : entero)
   $val\_min \leftarrow U[2]$  {valor mínimo}
   $pos\_min \leftarrow 2$  {posición donde está el mínimo}
  para  $i = 2$  hasta  $tamaño(U)$  hacer
    si  $U[i] < val\_min$  entonces
       $val\_min \leftarrow U[i]$ 
       $pos\_min \leftarrow i$ 
    fin si
  fin para
  devolver  $pos\_min$ 
fin función MinVector
```

.....

función *Dijkstra* ($L[1 \dots n, 1 \dots n]$: matriz) **devuelve** ($D[2 \dots n]$, $P[2 \dots n]$: vectores)

vector $D[2 \dots n]$ {longitud de los caminos mínimos}

vector $L[2 \dots n]$ {caminos mínimos}

{Inicialización}

$C \leftarrow \{2, 3, \dots, n\}$ $\{S = N \setminus C\}$

$P \leftarrow \{1, 1, \dots, 1\}$

para $i = 2$ hasta n **hacer**

$D[i] \leftarrow L[1, i]$

fin para

{Bucle voraz}

repetir $\{n - 2$ veces}

$v \leftarrow \text{MinVector}(D)$

$C \leftarrow C \setminus \{v\}$

para cada $w \in C$ **hacer**

si $D[w] > D[v] + L[v, w]$ **entonces**

$D[w] \leftarrow D[v] + L[v, w]$

$P[w] \leftarrow v$

fin si

fin para

fin repetir

Devolver D, P

fin función *Dijkstra*

19. – Realice un estudio teórico de la complejidad del algoritmo diseñado. Para ello deberá calcular y presentar separadamente el coste de todas y cada una de las funciones auxiliares implementadas en su algoritmo y, finalmente, calcular y presentar el coste total del algoritmo completo.

1. Coste de la función *MinVector*:

Se entra en el bucle **para** $\text{tamaño}(U) - 3$ veces. Luego el coste es del orden $\mathcal{O}(n)$.

2. El coste del algoritmo de Dijkstra implementando el grafo mediante una matriz de adyacencia es cuadrático ($\mathcal{O}(n^2)$):

Si el grafo tiene n nodos y a aristas la inicialización requiere un tiempo $\mathcal{O}(n)$. La selección de v dentro del bucle **repetir** es una llamada a la función *MinVector*, luego el tiempo está en $\mathcal{O}(n)$.

El bucle **para** interno realiza $n - 2, n - 3, \dots, 1$ iteraciones; dando un tiempo total que está en $\mathcal{O}(n^2)$.

Por tanto, el tiempo requerido está en $\mathcal{O}(n^2)$.

El coste del algoritmo completo es el del algoritmo de Dijkstra (donde ya hemos incluido la llamada a la función auxiliar) que es de $\mathcal{O}(n^2)$.

2.3. Implementación y prueba

2.3.1. Implementación del algoritmo

20. – Cree un módulo, de nombre *REPONEDOR*, que contenga una función, de nombre *RECORRIDO*, que implemente el algoritmo diseñado y que devuelva un parámetro con la solución al problema.

- Incluya en el módulo *REPONEDOR* todas aquellas funciones auxiliares necesarias para el correcto funcionamiento de la función *RECORRIDO*.
- El alumno podrá suponer que el hipermercado dispone de un máximo de 50 secciones diferentes

Módulo de definición

```
DEFINITION MODULE REPONEDOR;

CONST longMax = 50;
TYPE grafo = RECORD
  (* Estructura de datos principal del programa.
  Servirá para los bloques de datos leídos. *)
  numNodos : INTEGER;
  seccion : ARRAY [1..longMax] OF ARRAY [1..200] OF CHAR;
  adyacente : ARRAY [1..longMax], [1..longMax] OF INTEGER;
END;

TYPE solucion = RECORD
  (* Estructura para devolver la solución de recorrido *)
  distancias : ARRAY [1..longMax] OF INTEGER;
  punteros : ARRAY [1..longMax] OF INTEGER;
END;

TYPE conjunto = SET OF [1..longMax];

PROCEDURE MinVector (v: ARRAY OF INTEGER): INTEGER;
(* Devuelve la posición donde se encuentra un elemento mínimo
del vector dado *)

PROCEDURE BuscaSeccion (g : grafo; seccion : ARRAY OF CHAR) : INTEGER;
(* Busca en qué índice está determinada sección en el grafo *)

PROCEDURE RECORRIDO (g : grafo) : solucion;
(* Aplica el algoritmo de Dijkstra al grafo dado, devolviendo los
camino mínimos *)

END REPONEDOR.
```

Módulo de implementación

```
IMPLEMENTATION MODULE REPONEDOR;
```

```
FROM Strings IMPORT Equal;
```

```
PROCEDURE MinVector (v: ARRAY OF INTEGER): INTEGER;  
(* Devuelve la posición donde se encuentra un elemento mínimo  
del vector dado *)
```

```
(* se supone que v [2..tamano] *)
```

```
VAR
```

```
  i          : INTEGER; (*contador*)  
  tamano     : INTEGER; (*tamaño del vector*)  
  val_min    : INTEGER; (*valor mínimo*)  
  pos_min    : INTEGER; (*posición donde está el mínimo*)
```

```
BEGIN
```

```
  tamano := HIGH(v);  
  val_min := v[2];  
  pos_min := 2;
```

```
  FOR i:=3 TO tamano DO  
    IF v[i] < val_min THEN  
      val_min := v[i];  
      pos_min := i;  
    END;  
  END;
```

```
  RETURN pos_min;
```

```
END MinVector;
```

```
PROCEDURE BuscaSeccion (g : grafo; seccion : ARRAY OF CHAR) : INTEGER;  
(* Busca en qué índice está determinada sección en el grafo *)
```

```
VAR
```

```
  i : INTEGER; (*contador*)
```

```
BEGIN
```

```
  i:=1;  
  LOOP  
    IF Equal(g.seccion[i], seccion) THEN  
      RETURN i;  
    END;  
    i:=i+1;  
  END; (*loop*)
```

```
END BuscaSeccion;
```

```
PROCEDURE RECORRIDO (g: grafo) : solucion;
```

```
(* Implementa el algoritmo de Dijkstra para el grafo dado *)
```

```
VAR
```

```
  sol : solucion;  
  C   : conjunto;
```

```

        i      : INTEGER; (*contador*)
        v,w    : INTEGER; (*auxiliares*)

BEGIN

(* Inicialización *)
FOR i:=2 TO g.numNodos DO
    INCL(C,i);
    sol.punteros[i] := 1;
    sol.distancias[i] := g.adyacente[1,i];
END;

(* Bucle voraz *)
i:=0;
REPEAT
    (* Calculo el v de C que minimice sol.distancias[v] *)
    v:=2;
    LOOP (*calculo el primer v en C*)
        IF v IN C THEN
            EXIT;
        END;(*if*)
    v:=v+1;
    END;(*loop*)
    FOR w:=3 TO g.numNodos DO
        IF (w IN C) AND (sol.distancias[w]<sol.distancias[v]) THEN
            v:=w;
        END;(*if*)
    END;(*for*)
    EXCL(C,v); (*quita el elemento v de C*)
    FOR w:=2 TO g.numNodos DO (*recorro el vector*)
        IF w IN C THEN
            IF g.adyacente[v,w]<MAX(INTEGER) THEN
                (*Se hace esta comprobación previa para evitar overflow*)
                IF sol.distancias[w]>sol.distancias[v]+g.adyacente[v,w] THEN
                    sol.distancias[w] := sol.distancias[v] + g.adyacente[v,w];
                    sol.punteros[w] := v;
                END; (*if*)
            END; (*if*)
        END; (*if*)
    END; (*for*)
    i:=i+1;
UNTIL i=g.numNodos-2;

RETURN sol;
END RECORRIDO;
END REPONEDOR.

```

2.3.2. Implementación de un programa principal

21. – El alumno deberá implementar un módulo principal MAIN que:

- a) tomará como entrada de datos un fichero de texto plano con nombre HIPER.DAT, con la información del grafo correspondiente a un hipermercado y, a continuación, una lista de productos que el reponedor deberá colocar en las estanterías,
 - b) mapeará la entrada leída a partir del fichero de datos sobre las estructuras de datos diseñadas, y
 - c) hará la correspondiente llamada a la función RECORRIDO, presentando los resultados obtenidos en la pantalla.
- El fichero HIPER.DAT consta de una única definición de hipermercado, donde se especifican las distintas secciones del mismo en forma de nodos, así como los caminos existentes para pasar de una sección a otra. Cada línea del fichero representa un nodo origen, un nodo destino y el coste (en tiempo en minutos invertido) en pasar de uno a otro. Nótese como al tratarse de un grafo no dirigido, no es necesario definir enlaces de ida y de vuelta entre los dos nodos. El fin de la definición del grafo del hipermercado se indica en el fichero mediante la etiqueta REPONER, que a su vez indica el comienzo de la lista de productos que deben reponerse. El final del fichero vendrá indicado por la etiqueta FDE.
 - Se propone el siguiente ejemplo de formato del fichero de entrada HIPER.DAT:

```
almacen charcuteria 0  
carniceria congelados 0  
carniceria bebida 15  
lacteos bebida 5  
congelados bebida 40  
congelados imagen-sonido 0  
bebida ropa 35  
juguetes zapateria 25  
juguetes discos 12  
<REPONER>  
lacteos  
congelados  
juguetes  
discos  
<FDE>
```
 - Como salida, el programa deberá mostrar por pantalla la secuencia de caminos para cada uno de los productos a reponer, así como el tiempo empleado en llevar a cabo la operación. Finalmente, deberá mostrar por pantalla el tiempo total invertido en reponer todos los productos.

```

MODULE MAIN;

    FROM InOut      IMPORT ReadString, WriteString, WriteInt,
                          WriteLn, OpenInput, CloseInput, Done;
    FROM Strings    IMPORT Equal, Assign;
    FROM RealConv   IMPORT ValueReal;

    FROM REPONEDOR IMPORT longMax, grafo, solucion, BuscaSeccion, RECORRIDO;

    VAR
        S          : ARRAY[1..3] OF ARRAY[1..200] OF CHAR;
                    (* Variable auxiliar para leer los datos del fichero
                       hay que pensar qué valor dar al 200??
                       Los datos se leen de 3 en 3*)
        reponer     : ARRAY[1..longMax] OF INTEGER; (*secciones a reponer*)
        long_reponer: INTEGER;
        i,j,k       : INTEGER; (*contador*)
        ex          : BOOLEAN;
        g           : grafo;
        seccion1, seccion2 : INTEGER; (*auxiliares*)
        sol         : solucion;

    (* LEER FICHERO *)
    PROCEDURE LeeFichero();
    BEGIN
        REPEAT
            WriteLn;
            WriteString("Indicar nombre de Archivo de Entrada
                          (p.ejem. HIPER.DAT):");
            WriteLn;
            OpenInput(""); (*abre el fichero*)
        UNTIL Done;
    END LeeFichero;

    (* FUNCION PRINCIPAL *)
    BEGIN
        (* Inicializo el grafo *)

        g.numNodos := 0;
        FOR i:=1 TO longMax DO
            FOR j:=1 TO longMax DO
                g.adyacente[i,j] := MAX(INTEGER);
                (*pongo todas las distancias a infinito*)
            END;
        END;

        LeeFichero();

        REPEAT

```

```

(* Lee las secciones *)
(* el formato es sección1-sección2-distancia *)
ReadString(S[1]);
IF NOT Equal(S[1], "<REPONER>") THEN
  ReadString(S[2]); ReadString(S[3]);
  (* compruebo si la sección ya existe *)
  FOR i:=1 TO 2 DO
    (* Inicializo la variable ex (sirve para comprobar si
      ya existe la sección dada *)
    ex := FALSE;
    IF g.numNodos>0 THEN
      FOR j:=1 TO g.numNodos DO
        IF Equal(g.seccion[j], S[i]) THEN
          (*Comprueba si se había leído anteriormente*)
          ex := TRUE;
        END;(*if*)
      END; (*for*)
    END; (*if*)
    IF NOT ex THEN
      Assign(S[i], g.seccion[g.numNodos+1]);
      g.numNodos := g.numNodos + 1 ;
    END; (*if*)
  END; (*for*)

  (*ahora trato las distancias*)
  seccion1 := BuscaSeccion(g,S[1]);
  seccion2 := BuscaSeccion(g,S[2]);
  g.adyacente[seccion1,seccion2]:= TRUNC(ValueReal(S[3]));
  g.adyacente[seccion2,seccion1]:= TRUNC(ValueReal(S[3]));

  END;(*if*)
UNTIL Equal(S[1], "<REPONER>");
long_reponer:=1;
REPEAT
(*lee las secciones a reponer*)
  ReadString(S[1]);
  IF NOT Equal(S[1], "<FDE>") THEN
    reponer[long_reponer]:=BuscaSeccion(g,S[1]);
    long_reponer:=long_reponer+1;
  END;
UNTIL Equal(S[1], "<FDE>"); (*Lee del fichero hasta que termina*)

CloseInput; (*cierra el fichero*)

(*Aplica el algoritmo voraz*)
sol:=RECORRIDO(g);

(*Muestra la solución*)
WriteLn;
FOR i:=1 TO long_reponer-1 DO
  WriteString(g.seccion[reponer[i]]); WriteString(" :");WriteLn;WriteLn;
  WriteString(" - Tarda ");WriteInt(2*sol.distancias[reponer[i]],3);

```

```

WriteString(" minutos (ida");
WriteString(" =");WriteInt(sol.distancias[reponer[i]],2);
WriteString(" y vuelta");
WriteString(" =");WriteInt(sol.distancias[reponer[i]],2);
WriteString(")");
WriteLn;
WriteString(" - Realizando el siguiente recorrido: ");
WriteString(g.seccion[reponer[i]]; WriteString(" <--> ");
k:=sol.punteros[reponer[i]];
REPEAT
WriteString(g.seccion[k]); WriteString(" <--> ");
k:=sol.punteros[k];
UNTIL k=1;
WriteString(g.seccion[1]);
WriteLn;WriteLn;
END;
END MAIN.

```

2.4. Ejecución y pruebas

A continuación se muestra la salida del programa para varios ficheros de entrada¹:

1. Para el fichero HIPER.DAT:

```

almacen charcuteria 0
almacen carniceria 20
almacen videojuegos 10
carniceria congelados 0
carniceria bebida 15
carniceria lacteos 30
charcuteria congelados 60
lacteos bebida 5
congelados bebida 40
congelados imagen-sonido 0
bebida ropa 35
bebida juguetes 75
imagen-sonido juguetes 10
imagen-sonido ordenadores 55
ropa zapateria 30
juguetes zapateria 25
juguetes discos 12
ordenadores discos 5
ordenadores videojuegos 20
zapateria discos 0
<REPONER>
lacteos

```

¹incluidos en el CD con el nombre dado

congelados
juguetes
discos
zapateria
<FDE>

la salida es:

lacteos :

- Tarda 80 minutos (ida =40 y vuelta =40)
- Realizando el siguiente recorrido:
lacteos <--> bebida <--> carniceria <--> almacen

congelados :

- Tarda 40 minutos (ida =20 y vuelta =20)
- Realizando el siguiente recorrido:
congelados <--> carniceria <--> almacen

juguetes :

- Tarda 60 minutos (ida =30 y vuelta =30)
- Realizando el siguiente recorrido:
juguetes <--> imagen-sonido <--> congelados <--> carniceria <--> almacen

discos :

- Tarda 70 minutos (ida =35 y vuelta =35)
- Realizando el siguiente recorrido:
discos <--> ordenadores <--> videojuegos <--> almacen

zapateria :

- Tarda 70 minutos (ida =35 y vuelta =35)
- Realizando el siguiente recorrido:
zapateria <--> discos <--> ordenadores <--> videojuegos <--> almacen

Y el tiempo total invertido en reponer todos los productos es 320 minutos

Bloque 3

Maruja la ahorradora

3.1. Estudio del problema y diseño algorítmico

3.1.1. Detección del esquema algorítmico

22. – *¿Por qué el esquema algorítmico más adecuado para solucionar el problema es un esquema basado en exploración de grafos? Razonar por qué otras aproximaciones no serían válidas.*

Aunque se trata de un problema de optimización, no existe ningún criterio para elegir el producto a comprar en cada supermercado sin deshacer decisiones. Por tanto, en lugar de utilizar un esquema voraz habrá que elegir un esquema de exploración ciega de un árbol de búsqueda.

23. – *¿Qué tipo de estrategia de exploración de grafos (vuelta atrás o ramificación y poda) considera más adecuada para resolver su problema concreto? Razone su respuesta y explique las diferencias entre ambas.*

En este caso, el esquema más útil es el de ramificación y poda.

Los dos esquemas algorítmicos se utilizan para explorar un grafo dirigido implícito. El esquema de vuelta atrás se utiliza cuando queremos encontrar soluciones a un problema. El de ramificación y poda se utiliza cuando lo que se busca es la *solución óptima* de un problema.

24. – *¿Tendría cabida como solución de este problema la aplicación de un esquema de exploración de grafos tipo recorrido en anchura o recorrido en profundidad? Razone su respuesta.*

Al igual que en el caso de vuelta atrás los esquemas de recorrido en anchura o profundidad se utilizan para buscar soluciones, pero no son aplicables directamente para problemas de optimización.

En lugar de explorar el árbol en anchura o profundidad en el algoritmo de ramificación y poda, se examinará en cada momento el nodo más prometedor, es decir, aquel cuya cota inferior sea mínima.

25. – Presentar y explicar el esquema algorítmico de exploración de grafos finalmente seleccionado, describiendo la funcionalidad de los distintos elementos del mismo (funciones auxiliares).

El esquema de ramificación y poda es una variante de búsqueda ciega en árbol, en la que se busca una solución que sea óptima de acuerdo a un criterio dado. Se dispone de una cota superior global, que coincide en cada momento con la mejor solución encontrada hasta entonces; y de una forma de asignar cotas inferiores a cada nodo (ninguna solución obtenida a partir de ese nodo será mejor que su cota inferior). El algoritmo se distingue porque aquellas ramas cuya cota inferior sea mayor que la cota superior global pueden abandonarse sin ser exploradas. Además, se puede recorrer el árbol expandiendo en cada momento la rama más prometedora, lo que reduce el número de nodos a visitar.

El esquema algorítmico de ramificación y poda es el siguiente:

Algoritmo de Ramificación y Poda

función *ramificación-y-poda* (C : ensayo) **devuelve** (S : ensayo)

- 1: {Inicialización}
- 2: $m \leftarrow$ montículo-vacío
- 3: $cota\text{-superior} \leftarrow$ $cota\text{-superior-inicial}$
- 4: $S \leftarrow \emptyset$
- 5:
- 6: *añadir-elemento* (C, m)
- 7: **mientras no vacío**(m) **hacer**
- 8: nodo \leftarrow *extraer-raíz* (m)
- 9: **si válido** (nodo) **entonces**
- 10: **si** $nodo < cota\text{-superior}$ **entonces**
- 11: $S \leftarrow$ nodo
- 12: $cota\text{-superior} \leftarrow$ *coste* (nodo)
- 13: **fin si**
- 14: **sino**
- 15: **si** $cota\text{-inferior}$ (nodo) $\geq cota\text{-superior}$ **entonces**
- 16: **devolver** S
- 17: **sino**
- 18: **para cada hijo en** *compleciones* **hacer**
- 19: **si** *condiciones-de-poda* (hijo) **y** $cota\text{-inferior}$ (hijo) $< cota\text{-superior}$ **entonces**
- 20: *añadir-nodo* (hijo, m)
- 21: **fin si**
- 22: **fin para**
- 23: **fin si**
- 24: **fin si**
- 25: **fin mientras**

fin función *ramificación-y-poda*

donde la estructura de datos **ensayo** se detalla más adelante.

3.1.2. Particularización del algoritmo sobre el problema concreto

26. – *¿Cuáles son las estructuras de datos más adecuadas para resolver el problema? Exponga y explique las estructuras de datos principales para resolver el problema, así como aquellas estructuras de datos auxiliares necesarias para almacenar la información de cálculos intermedios del algoritmo.*

La estructura más adecuada es un montículo de mínimos, de forma que en la raíz siempre tengamos el nodo más prometedor.

Tendría la siguiente definición:

tipo	<i>ensayo</i>	=	registro
	<i>asignaciones</i>	:	vector [1 . . . <i>longMax</i>] de enteros
	<i>tareas-no-asignadas</i>	:	lista [1 . . . <i>longMax</i>] de enteros
	<i>ultimo-producto-asignado</i>	:	entero
	<i>coste</i>	:	entero