Capítulo 1

LA COMPLEJIDAD DE LOS ALGORITMOS

1.1 INTRODUCCIÓN

En un sentido amplio, dado un problema y un dispositivo donde resolverlo, es necesario proporcionar un método preciso que lo resuelva, adecuado al dispositivo. A tal método lo denominamos *algoritmo*.

En el presente texto nos vamos a centrar en dos aspectos muy importantes de los algoritmos, como son su diseño y el estudio de su eficiencia.

El primero se refiere a la búsqueda de métodos o procedimientos, secuencias finitas de instrucciones adecuadas al dispositivo que disponemos, que permitan resolver el problema. Por otra parte, el segundo nos permite medir de alguna forma el coste (en tiempo y recursos) que consume un algoritmo para encontrar la solución y nos ofrece la posibilidad de comparar distintos algoritmos que resuelven un mismo problema.

Este capítulo está dedicado al segundo de estos aspectos: la eficiencia. En cuanto a las técnicas de diseño, que corresponden a los patrones fundamentales sobre los que se construyen los algoritmos que resuelven un gran número de problemas, se estudiarán en los siguientes capítulos.

1.2 EFICIENCIA Y COMPLEJIDAD

Una vez dispongamos de un algoritmo que funciona correctamente, es necesario definir criterios para medir su rendimiento o comportamiento. Estos criterios se centran principalmente en su simplicidad y en el uso eficiente de los recursos.

A menudo se piensa que un algoritmo sencillo no es muy eficiente. Sin embargo, la sencillez es una característica muy interesante a la hora de diseñar un algoritmo, pues facilita su verificación, el estudio de su eficiencia y su mantenimiento. De ahí que muchas veces prime la simplicidad y legibilidad del código frente a alternativas más crípticas y eficientes del algoritmo. Este hecho se pondrá de manifiesto en varios de los ejemplos mostrados a lo largo de este libro, en donde profundizaremos más en este compromiso.

Respecto al uso eficiente de los recursos, éste suele medirse en función de dos parámetros: el *espacio*, es decir, memoria que utiliza, y el *tiempo*, lo que tarda en ejecutarse. Ambos representan los costes que supone encontrar la solución al problema planteado mediante un algoritmo. Dichos parámetros van a servir además para comparar algoritmos entre sí, permitiendo determinar el más adecuado de

entre varios que solucionan un mismo problema. En este capítulo nos centraremos solamente en la eficiencia temporal.

El tiempo de ejecución de un algoritmo va a depender de diversos factores como son: los datos de entrada que le suministremos, la calidad del código generado por el compilador para crear el programa objeto, la naturaleza y rapidez de las instrucciones máquina del procesador concreto que ejecute el programa, y la complejidad intrínseca del algoritmo. Hay dos estudios posibles sobre el tiempo:

1. Uno que proporciona una medida *teórica* (a priori), que consiste en obtener una función que acote (por arriba o por abajo) el tiempo de ejecución del algoritmo para unos valores de entrada dados.
2. Y otro que ofrece una medida *real* (a posteriori), consistente en medir el tiempo de ejecución del algoritmo para unos valores de entrada dados y en un ordenador concreto.

Ambas medidas son importantes puesto que, si bien la primera nos ofrece estimaciones del comportamiento de los algoritmos de forma independiente del ordenador en donde serán implementados y sin necesidad de ejecutarlos, la segunda representa las medidas reales del comportamiento del algoritmo. Estas medidas son *funciones temporales* de los datos de entrada.

Entendemos por *tamaño de la entrada* el número de componentes sobre los que se va a ejecutar el algoritmo. Por ejemplo, la dimensión del vector a ordenar o el tamaño de las matrices a multiplicar.

La unidad de tiempo a la que debe hacer referencia estas medidas de eficiencia no puede ser expresada en segundos o en otra unidad de tiempo concreta, pues no existe un ordenador estándar al que puedan hacer referencia todas las medidas. Denotaremos por $T(n)$ el tiempo de ejecución de un algoritmo para una entrada de tamaño n.

Teóricamente $T(n)$ debe indicar el número de instrucciones ejecutadas por un ordenador idealizado. Debemos buscar por tanto medidas simples y abstractas, independientes del ordenador a utilizar. Para ello es necesario acotar de alguna forma la diferencia que se puede producir entre distintas implementaciones de un mismo algoritmo, ya sea del mismo código ejecutado por dos máquinas de distinta velocidad, como de dos códigos que implementen el mismo método. Esta diferencia es la que acota el siguiente principio:

Principio de Invarianza

Dado un algoritmo y dos implementaciones suyas I_1 e I_2, que tardan $T_1(n)$ y $T_2(n)$ segundos respectivamente, el *Principio de Invarianza* afirma que existe una constante real $c > 0$ y un número natural n_0 tales que para todo $n \geq n_0$ se verifica que $T_1(n) \leq cT_2(n)$.

Es decir, el tiempo de ejecución de dos implementaciones distintas de un algoritmo dado no va a diferir más que en una constante multiplicativa.

Con esto podemos definir sin problemas que un algoritmo tarda un tiempo *del orden de* $T(n)$ si existen una constante real $c > 0$ y una implementación I del algoritmo que tarda menos que $cT(n)$, para todo n tamaño de la entrada.

Dos factores a tener muy en cuenta son la constante multiplicativa y el n_0 para los que se verifican las condiciones, pues si bien a priori un algoritmo de orden cuadrático es mejor que uno de orden cúbico, en el caso de tener dos algoritmos cuyos tiempos de ejecución son 10^6n^2 y $5n^3$ el primero sólo será mejor que el segundo para tamaños de la entrada superiores a 200.000.

También es importante hacer notar que el comportamiento de un algoritmo puede cambiar notablemente para diferentes entradas (por ejemplo, lo ordenados que se encuentren ya los datos a ordenar). De hecho, para muchos programas el tiempo de ejecución es en realidad una función de la entrada específica, y no sólo del tamaño de ésta. Así suelen estudiarse tres casos para un mismo algoritmo: *caso peor*, *caso mejor* y *caso medio*.

El caso mejor corresponde a la traza (secuencia de sentencias) del algoritmo que realiza menos instrucciones. Análogamente, el caso peor corresponde a la traza del algoritmo que realiza más instrucciones. Respecto al caso medio, corresponde a la traza del algoritmo que realiza un número de instrucciones igual a la esperanza matemática de la variable aleatoria definida por todas las posibles trazas del algoritmo para un tamaño de la entrada dado, con las probabilidades de que éstas ocurran para esa entrada.

Es muy importante destacar que esos casos corresponden a un tamaño de la entrada dado, puesto que es un error común confundir el caso mejor con el que menos instrucciones realiza en cualquier caso, y por lo tanto contabilizar las instrucciones que hace para $n = 1$.

A la hora de medir el tiempo, siempre lo haremos en función del *número de operaciones elementales* que realiza dicho algoritmo, entendiendo por operaciones elementales (en adelante OE) aquellas que el ordenador realiza en tiempo acotado por una constante. Así, consideraremos OE las operaciones aritméticas básicas, asignaciones a variables de tipo predefinido por el compilador, los saltos (llamadas a funciones y procedimientos, retorno desde ellos, etc.), las comparaciones lógicas y el acceso a estructuras indexadas básicas, como son los vectores y matrices. Cada una de ellas contabilizará como 1 OE.

Resumiendo, el tiempo de ejecución de un algoritmo va a ser una función que mide el número de operaciones elementales que realiza el algoritmo para un tamaño de entrada dado.

En general, es posible realizar el estudio de la complejidad de un algoritmo sólo en base a un conjunto reducido de sentencias, aquellas que caracterizan que el algoritmo sea lento o rápido en el sentido que nos interesa. También es posible distinguir entre los tiempos de ejecución de las diferentes operaciones elementales, lo cual es necesario a veces por las características específicas del ordenador (por ejemplo, se podría considerar que las operaciones + y ÷ presentan complejidades diferentes debido a su implementación). Sin embargo, en este texto tendremos en cuenta, a menos que se indique lo contrario, todas las operaciones elementales del lenguaje, y supondremos que sus tiempos de ejecución son todos iguales.

Para hacer un estudio del tiempo de ejecución de un algoritmo para los tres casos citados comenzaremos con un ejemplo concreto. Supongamos entonces que disponemos de la definición de los siguientes tipos y constantes:

```
CONST n =...; (* num. maximo de elementos de un vector *);
TYPE vector = ARRAY [1..n] OF INTEGER;
```

y de un algoritmo cuya implementación en Modula-2 es:

```
PROCEDURE Buscar(VAR a:vector;c:INTEGER):CARDINAL;
 VAR j:CARDINAL;
BEGIN
 j:=1;                              (* 1 *)
 WHILE (a[j]<c) AND (j<n) DO        (* 2 *)
    j:=j+1                          (* 3 *)
 END;                               (* 4 *)
 IF a[j]=c THEN                     (* 5 *)
    RETURN j                        (* 6 *)
 ELSE RETURN 0                      (* 7 *)
 END                                (* 8 *)
END Buscar;
```

Para determinar el tiempo de ejecución, calcularemos primero el número de operaciones elementales (OE) que se realizan:

- En la línea (1) se ejecuta 1 OE (una asignación).
- En la línea (2) se efectúa la condición del bucle, con un total de 4 OE (dos comparaciones, un acceso al vector, y un *AND*).
- La línea (3) está compuesta por un incremento y una asignación (2 OE).
- La línea (5) está formada por una condición y un acceso al vector (2 OE).
- La línea (6) contiene un *RETURN* (1 OE) si la condición se cumple.
- La línea (7) contiene un *RETURN* (1 OE), cuando la condición del *IF* anterior es falsa.

Obsérvese cómo no se contabiliza la copia del vector a la pila de ejecución del programa, pues se pasa por referencia y no por valor (está declarado como un argumento *VAR*, aunque no se modifique dentro de la función). En caso de pasarlo por valor, necesitaríamos tener en cuenta el coste que esto supone (un incremento de n OE). Con esto:

- En el *caso mejor* para el algoritmo, se efectuará la línea (1) y de la línea (2) sólo la primera mitad de la condición, que supone 2 OE (suponemos que las expresiones se evalúan de izquierda a derecha, y con "cortocircuito", es decir, una expresión lógica deja de ser evaluada en el momento que se conoce su valor, aunque no hayan sido evaluados todos sus términos). Tras ellas la función acaba ejecutando las líneas (5) a (7). En consecuencia, $T(n)=1+2+3=6$.
- En el *caso peor*, se efectúa la línea (1), el bucle se repite $n-1$ veces hasta que se cumple la segunda condición, después se efectúa la condición de la línea (5) y la función acaba al ejecutarse la línea (7). Cada iteración del bucle está compuesta por las líneas (2) y (3), junto con una ejecución adicional de la línea (2) que es la que ocasiona la salida del bucle. Por tanto

$$T(n) = 1 + \left(\left(\sum_{i=1}^{n-1} (4+2) \right) + 4 \right) + 2 + 1 = 6n + 2.$$

- En el *caso medio*, el bucle se ejecutará un número de veces entre 0 y n–1, y vamos a suponer que cada una de ellas tiene la misma probabilidad de suceder. Como existen n posibilidades (puede que el número buscado no esté) suponemos a priori que son equiprobables y por tanto cada una tendrá una probabilidad asociada de $1/n$. Con esto, el número medio de veces que se efectuará el bucle es de

$$\sum_{i=0}^{n-1} i \frac{1}{n} = \frac{n-1}{2}.$$

  Tenemos pues que

$$T(n) = 1 + \left(\left(\sum_{i=1}^{(n-1)/2} (4+2) \right) + 2 \right) + 2 + 1 = 3n + 3.$$

Es importante observar que no es necesario conocer el propósito del algoritmo para analizar su tiempo de ejecución y determinar sus casos mejor, peor y medio, sino que basta con estudiar su código. Suele ser un error muy frecuente el determinar tales casos basándose sólo en la funcionalidad para la que el algoritmo fue concebido, olvidando que es el código implementado el que los determina.

En este caso, un examen más detallado de la función (¡y no de su nombre!) nos muestra que tras su ejecución, la función devuelve la posición de un entero dado c dentro de un vector ordenado de enteros, devolviendo 0 si el elemento no está en el vector. Lo que acabamos de probar es que su caso mejor se da cuando el elemento está en la primera posición del vector. El caso peor se produce cuando el elemento no está en el vector, y el caso medio ocurre cuando consideramos equiprobables cada una de las posiciones en las que puede encontrarse el elemento dentro del vector (incluyendo la posición especial 0, que indica que el elemento a buscar no se encuentra en el vector).

1.2.1 Reglas generales para el cálculo del número de OE

La siguiente lista presenta un conjunto de reglas generales para el cálculo del número de OE, siempre considerando el peor caso. Estas reglas definen el número de OE de cada estructura básica del lenguaje, por lo que el número de OE de un algoritmo puede hacerse por inducción sobre ellas.

- Vamos a considerar que el tiempo de una OE es, por definición, de orden 1. La constante c que menciona el Principio de Invarianza dependerá de la implementación particular, pero nosotros supondremos que vale 1.
- El tiempo de ejecución de una secuencia consecutiva de instrucciones se calcula sumando los tiempos de ejecución de cada una de las instrucciones.

- El tiempo de ejecución de la sentencia "`CASE C OF v1:S1|v2:S2|...|vn:Sn END;`" es $T = T(C) + \max\{T(S_1),T(S_2),...,T(S_n)\}$. Obsérvese que $T(C)$ incluye el tiempo de comparación con v_1, v_2 ,..., v_n.
- El tiempo de ejecución de la sentencia "`IF C THEN S1 ELSE S2 END;`" es $T = T(C) + \max\{T(S_1),T(S_2)\}$.
- El tiempo de ejecución de un bucle de sentencias "`WHILE C DO S END;`" es $T = T(C) + (\text{nº iteraciones})*(T(S) + T(C))$. Obsérvese que tanto $T(C)$ como $T(S)$ pueden variar en cada iteración, y por tanto habrá que tenerlo en cuenta para su cálculo.
- Para calcular el tiempo de ejecución del resto de sentencias iterativas (*FOR*, *REPEAT*, *LOOP*) basta expresarlas como un bucle *WHILE*. A modo de ejemplo, el tiempo de ejecución del bucle:

```
FOR i:=1 TO n DO
   S
END;
```

  puede ser calculado a partir del bucle equivalente:

```
i:=1;
WHILE i<=n DO
   S; INC(i)
END;
```

- El tiempo de ejecución de una llamada a un procedimiento o función $F(P_1, P_2,..., P_n)$ es 1 (por la llamada), más el tiempo de evaluación de los parámetros P_1, P_2,..., P_n, más el tiempo que tarde en ejecutarse F, esto es, $T = 1 + T(P_1) + T(P_2) + ... + T(P_n) + T(F)$. No contabilizamos la copia de los argumentos a la pila de ejecución, salvo que se trate de estructuras complejas (registros o vectores) que se pasan por valor. En este caso contabilizaremos tantas OE como valores simples contenga la estructura. El paso de parámetros por referencia, por tratarse simplemente de punteros, no contabiliza tampoco.
- El tiempo de ejecución de las llamadas a procedimientos recursivos va a dar lugar a ecuaciones en recurrencia, que veremos posteriormente.
- También es necesario tener en cuenta, cuando el compilador las incorpore, las optimizaciones del código y la forma de evaluación de las expresiones, que pueden ocasionar "cortocircuitos" o realizarse de forma "perezosa" (*lazy*). En el presente trabajo supondremos que no se realizan optimizaciones, que existe el cortocircuito y que no existe evaluación perezosa.

1.3 COTAS DE COMPLEJIDAD. MEDIDAS ASINTÓTICAS

Una vez vista la forma de calcular el tiempo de ejecución T de un algoritmo, nuestro propósito es intentar clasificar dichas funciones de forma que podamos compararlas. Para ello, vamos a definir clases de equivalencia, correspondientes a las funciones que "crecen de la misma forma".

En las siguientes definiciones **N** denotará el conjunto de los números naturales y **R** el de los reales.

1.3.1 Cota Superior. Notación O

Dada una función *f*, queremos estudiar aquellas funciones *g* que a lo sumo crecen tan deprisa como *f*. Al conjunto de tales funciones se le llama cota superior de *f* y lo denominamos O(*f*). Conociendo la cota superior de un algoritmo podemos asegurar que, en ningún caso, el tiempo empleado será de un orden superior al de la cota.

Definición 1.1
Sea f: $\mathbf{N}\rightarrow[0,\infty)$. Se define el conjunto de funciones de orden O (Omicron) de f como:

$$O(f) = \{g: \mathbf{N}\rightarrow[0,\infty) \mid \exists c\in \mathbf{R},\ c>0,\ \exists n_0\in \mathbf{N} \bullet g(n) \le cf(n)\ \ \forall n \ge n_0\}.$$

Diremos que una función t: $\mathbf{N} \rightarrow[0,\infty)$ es de orden O de f si $t \in O(f)$.

Intuitivamente, $t \in O(f)$ indica que t está acotada superiormente por algún múltiplo de f. Normalmente estaremos interesados en la menor función f tal que t pertenezca a O(*f*).

En el ejemplo del algoritmo *Buscar* analizado anteriormente obtenemos que su tiempo de ejecución en el mejor caso es O(1), mientras que sus tiempos de ejecución para los casos peor y medio son O(*n*).

Propiedades de O

Veamos las propiedades de la cota superior. La demostración de todas ellas se obtiene aplicando la definición 1.1.

1. Para cualquier función f se tiene que $f \in O(f)$.
2. $f \in O(g) \Rightarrow\ O(f) \subset O(g)$.
3. $O(f) = O(g) \Leftrightarrow f \in O(g)$ y $g \in O(f)$.
4. Si $f \in O(g)$ y $g \in O(h) \Rightarrow f \in O(h)$.
5. Si $f \in O(g)$ y $f \in O(h) \Rightarrow f \in O(\min(g,h))$.
6. Regla de la suma: Si $f_1 \in O(g)$ y $f_2 \in O(h) \Rightarrow f_1 + f_2 \in O(\max(g,h))$.
7. Regla del producto: Si $f_1 \in O(g)$ y $f_2 \in O(h) \Rightarrow f_1 \cdot f_2 \in O(g \cdot h)$.
8. Si existe $\lim_{n\to\infty} \frac{f(n)}{g(n)} = k$, dependiendo de los valores que tome k obtenemos:
   a) Si $k \neq 0$ y $k < \infty$ entonces $O(f) = O(g)$.
   b) Si $k = 0$ entonces $f \in O(g)$, es decir, $O(f) \subset O(g)$, pero sin embargo se verifica que $g \notin O(f)$.

Obsérvese la importancia que tiene el que exista tal límite, pues si no existiese (o fuera infinito) no podría realizarse tal afirmación, como veremos en la resolución de los problemas de este capítulo.

De las propiedades anteriores se deduce que la relación $\sim_O$, definida por $f \sim_O g$ si y sólo si $O(f) = O(g)$, es una relación de equivalencia. Siempre escogeremos el representante más sencillo para cada clase; así los órdenes de complejidad constante serán expresados por $O(1)$, los lineales por $O(n)$, etc.

1.3.2 Cota Inferior. Notación Ω

Dada una función f, queremos estudiar aquellas funciones g que a lo sumo crecen tan lentamente como f. Al conjunto de tales funciones se le llama cota inferior de f y lo denominamos $\Omega(f)$. Conociendo la cota inferior de un algoritmo podemos asegurar que, en ningún caso, el tiempo empleado será de un orden inferior al de la cota.

Definición 1.2

Sea f: $\mathbf{N} \to [0,\infty)$. Se define el conjunto de funciones de orden Ω (Omega) de f como:

$$\Omega(f) = \{g:\mathbf{N} \to [0,\infty) \mid \exists c \in \mathbf{R}\ ,\ c>0,\ \exists n_0 \in \mathbf{N} \bullet\ g(n) \geq cf(n)\ \ \forall n \geq n_0\}.$$

Diremos que una función t:$\mathbf{N} \to [0,\infty)$ es de orden Ω de f si $t \in \Omega(f)$.

Intuitivamente, $t \in \Omega(f)$ indica que t está acotada inferiormente por algún múltiplo de f. Normalmente estaremos interesados en la mayor función f tal que t pertenezca a $\Omega(f)$, a la que denominaremos su cota inferior.

Obtener buenas cotas inferiores es en general muy difícil, aunque siempre existe una cota inferior trivial para cualquier algoritmo: al menos hay que leer los datos y luego escribirlos, de forma que ésa sería una primera cota inferior. Así, para ordenar n números una cota inferior sería n, y para multiplicar dos matrices de orden n sería n^2; sin embargo, los mejores algoritmos conocidos son de órdenes $n\log n$ y $n^{2.8}$ respectivamente.

Propiedades de Ω

Veamos las propiedades de la cota inferior Ω. La demostración de todas ellas se obtiene de forma simple aplicando la definición 1.2.

1. Para cualquier función f se tiene que $f \in \Omega(f)$.
2. $f \in \Omega(g) \Rightarrow \Omega(f) \subset \Omega(g)$.
3. $\Omega(f) = \Omega(g) \Leftrightarrow f \in \Omega(g)$ y $g \in \Omega(f)$.
4. Si $f \in \Omega(g)$ y $g \in \Omega(h) \Rightarrow f \in \Omega(h)$.
5. Si $f \in \Omega(g)$ y $f \in \Omega(h) \Rightarrow f \in \Omega(\max(g,h))$.
6. Regla de la suma: Si $f_1 \in \Omega(g)$ y $f_2 \in \Omega(h) \Rightarrow f_1 + f_2 \in \Omega(g + h)$.

7. Regla del producto: Si $f_1 \in \Omega(g)$ y $f_2 \in \Omega(h) \Rightarrow f_1 \cdot f_2 \in \Omega(g \cdot h)$.

8. Si existe $\lim_{n\to\infty} \frac{f(n)}{g(n)} = k$, dependiendo de los valores que tome k obtenemos:
   a) Si $k \neq 0$ y $k < \infty$ entonces $\Omega(f) = \Omega(g)$.
   b) Si $k = 0$ entonces $g \in \Omega(f)$, es decir, $\Omega(g) \subset \Omega(f)$, pero sin embargo se verifica que $f \notin \Omega(g)$.

De las propiedades anteriores se deduce que la relación $\sim_\Omega$, definida por $f \sim_\Omega g$ si y sólo si $\Omega(f) = \Omega(g)$, es una relación de equivalencia. Al igual que hacíamos para el caso de la cota superior O, siempre escogeremos el representante más sencillo para cada clase. Así los órdenes de complejidad Ω constante serán expresados por $\Omega(1)$, los lineales por $\Omega(n)$, etc.

1.3.3 Orden Exacto. Notación Θ

Como última cota asintótica, definiremos los conjuntos de funciones que crecen asintóticamente de la misma forma.

Definición 1.3
Sea f: $\mathbf{N} \to [0,\infty)$. Se define el conjunto de funciones de orden Θ (Theta) de f como:

$$\Theta(f) = O(f) \cap \Omega(f)$$

o, lo que es igual:

$$\Theta(f) = \{g:\mathbf{N}\to[0,\infty) \mid \exists c,d \in \mathbf{R}\ ,\ c,d>0,\ \exists n_0 \in \mathbf{N} \bullet cf(n) \le g(n) \le df(n)\ \forall n \ge n_0\}.$$

Diremos que una función t :$\mathbf{N}\to[0,\infty)$ es de orden Θ de f si $t \in \Theta(f)$.

Intuitivamente, $t \in \Theta(f)$ indica que t está acotada tanto superior como inferiormente por múltiplos de f, es decir, que t y f crecen de la misma forma.

Propiedades de Θ

Veamos las propiedades de la cota exacta. La demostración de todas ellas se obtiene también de forma simple aplicando la definición 1.3 y las propiedades de O y Ω.

1. Para cualquier función f se tiene que $f \in \Theta(f)$.
2. $f \in \Theta(g) \Rightarrow \Theta(f) = \Theta(g)$.
3. $\Theta(f) = \Theta(g) \Leftrightarrow f \in \Theta(g)$ y $g \in \Theta(f)$.
4. Si $f \in \Theta(g)$ y $g \in \Theta(h) \Rightarrow f \in \Theta(h)$.
5. Regla de la suma: Si $f_1 \in \Theta(g)$ y $f_2 \in \Theta(h) \Rightarrow f_1 + f_2 \in \Theta(\max(g,h))$.

6. Regla del producto: Si $f_1 \in \Theta(g)$ y $f_2 \in \Theta(h) \Rightarrow f_1 \cdot f_2 \in \Theta(g \cdot h)$.

7. Si existe $\lim_{n\to\infty} \frac{f(n)}{g(n)} = k$, dependiendo de los valores que tome k obtenemos:
   a) Si $k \neq 0$ y $k < \infty$ entonces $\Theta(f) = \Theta(g)$.
   b) Si $k = 0$ los órdenes exactos de f y g son distintos.

1.3.4 Observaciones sobre las cotas asintóticas

1. La utilización de las cotas asintóticas para comparar funciones de tiempo de ejecución se basa en la hipótesis de que son suficientes para decidir el mejor algoritmo, prescindiendo de las constantes de proporcionalidad. Sin embargo, esta hipótesis puede no ser cierta cuando el tamaño de la entrada es pequeño.
2. Para un algoritmo dado se pueden obtener tres funciones que miden su tiempo de ejecución, que corresponden a sus casos mejor, medio y peor, y que denominaremos respectivamente $T_m(n)$, $T_{1/2}(n)$ y $T_p(n)$. Para cada una de ellas podemos dar tres cotas asintóticas de crecimiento, por lo que se obtiene un total de nueve cotas para el algoritmo.
3. Para simplificar, dado un algoritmo diremos que su orden de complejidad es $O(f)$ si su tiempo de ejecución para el peor caso es de orden O de f, es decir, $T_p(n)$ es de orden $O(f)$. De forma análoga diremos que su orden de complejidad para el mejor caso es $\Omega(g)$ si su tiempo de ejecución para el mejor caso es de orden Ω de g, es decir, $T_m(n)$ es de orden $\Omega(g)$.
4. Por último, diremos que un algoritmo es de orden exacto $\Theta(f)$ si su tiempo de ejecución en el caso medio $T_{1/2}(n)$ es de este orden.

1.4 RESOLUCIÓN DE ECUACIONES EN RECURRENCIA

En las secciones anteriores hemos descrito cómo determinar el tiempo de ejecución de un algoritmo a partir del cómputo de sus operaciones elementales (OE). En general, este cómputo se reduce a un mero ejercicio de cálculo. Sin embargo, para los algoritmos recursivos nos vamos a encontrar con una dificultad añadida, pues la función que establece su tiempo de ejecución viene dada por una ecuación en recurrencia, es decir, $T(n) = E(n)$, en donde en la expresión E aparece la propia función T.

Resolver tal tipo de ecuaciones consiste en encontrar una expresión no recursiva de T, y por lo general no es una labor fácil. Lo que veremos en esta sección es cómo se pueden resolver algunos tipos concretos de ecuaciones en recurrencia, que son las que se dan con más frecuencia al estudiar el tiempo de ejecución de los algoritmos desarrollados según las técnicas aquí presentadas.

1.4.1 Recurrencias homogéneas

Son de la forma:

$$a_0 T(n) + a_1 T(n-1) + a_2 T(n-2) + ... + a_k T(n-k) = 0$$

donde los coeficientes a_i son números reales, y k es un número natural entre 1 y n. Para resolverlas vamos a buscar soluciones que sean combinaciones de funciones exponenciales de la forma:

$$T(n) = c_1 p_1(n) r_1^n + c_2 p_2(n) r_2^n + ... + c_k p_k(n) r_k^n = \sum_{i=1}^{k} c_i p_i(n) r_i^n ,$$

donde los valores c_1, c_2,...,c_n y r_1, r_2, ...,r_n son números reales, y $p_1(n)$,...,$p_k(n)$ son polinomios en n con coeficientes reales. Si bien es cierto que estas ecuaciones podrían tener soluciones más complejas que éstas, se conjetura que serían del mismo orden y por tanto no nos ocuparemos de ellas.

Para resolverlas haremos el cambio $x^n = T(n)$, con lo cual obtenemos la *ecuación característica* asociada:

$$a_0 x^k + a_1 x^{k-1} + a_2 x^{k-2} + ... + a_k = 0 .$$

Llamemos r_1, r_2,...,r_k a sus raíces, ya sean reales o complejas. Dependiendo del orden de multiplicidad de tales raíces, pueden darse los dos siguientes casos.

Caso 1: Raíces distintas

Si todas las raíces de la ecuación característica son distintas, esto es, $r_i \neq r_j$ si $i \neq j$, entonces la solución de la ecuación en recurrencia viene dada por la expresión:

$$T(n) = c_1 r_1^n + c_2 r_2^n + ... + c_k r_k^n = \sum_{i=1}^{k} c_i r_i^n$$

donde los coeficientes c_i se determinan a partir de las condiciones iniciales.

Como ejemplo, veamos lo que ocurre para la ecuación en recurrencia definida para la sucesión de Fibonacci:

$$T(n) = T(n-1) + T(n-2), \quad n \geq 2$$

con las condiciones iniciales $T(0) = 0$, $T(1) = 1$. Haciendo el cambio $x^2 = T(n)$ obtenemos su ecuación característica $x^2 = x + 1$, o lo que es igual, $x^2 - x - 1 = 0$, cuyas raíces son:

$$r_1 = \frac{1+\sqrt{5}}{2}, \quad r_2 = \frac{1-\sqrt{5}}{2}$$

y por tanto

$$T(n) = c_1 \left(\frac{1+\sqrt{5}}{2} \right)^n + c_2 \left(\frac{1-\sqrt{5}}{2} \right)^n .$$

Para calcular las constantes c_1 y c_2 necesitamos utilizar las condiciones iniciales de la ecuación original, obteniendo:

$$\left.\begin{aligned} T(0) &= c_1\left(\frac{1+\sqrt{5}}{2}\right)^0 + c_2\left(\frac{1-\sqrt{5}}{2}\right)^0 = c_1 + c_2 = 0 \\ T(1) &= c_1\left(\frac{1+\sqrt{5}}{2}\right)^1 + c_2\left(\frac{1-\sqrt{5}}{2}\right)^1 = 1 \end{aligned}\right\} \Rightarrow c_1 = -c_2 = \frac{1}{\sqrt{5}}.$$

Sustituyendo entonces en la ecuación anterior, obtenemos

$$T(n) = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n \in \mathrm{O}(\varphi^n).$$

Caso 2: Raíces con multiplicidad mayor que 1

Supongamos que alguna de las raíces (p.e. r_1) tiene multiplicidad m>1. Entonces la ecuación característica puede ser escrita en la forma

$$(\mathrm{x} - r_1)^m(x - r_2)...(x - r_{k-m+1})$$

en cuyo caso la solución de la ecuación en recurrencia viene dada por la expresión:

$$T(n) = \sum_{i=1}^{m} c_i n^{i-1} r_1^n + \sum_{i=m+1}^{k} c_i r_{i-m+1}^n$$

donde los coeficientes c_i se determinan a partir de las condiciones iniciales.

Veamos un ejemplo en el que la ecuación en recurrencia es:

$$T(n) = 5T(n-1) - 8T(n-2) + 4T(n-3), \quad n \geq 2$$

con las condiciones iniciales $T(k) = k$ para $k = 0, 1, 2$. La ecuación característica que se obtiene es $x^3 - 5x^2 + 8x - 4 = 0$, o lo que es igual $(x-2)^2(x-1) = 0$ y por tanto,

$$T(n) = c_1 2^n + c_2 n 2^n + c_3 1^n.$$

De las condiciones iniciales obtenemos $c_1 = 2$, $c_2 = -1/2$ y $c_3 = -2$, por lo que

$$T(n) = 2^{n+1} - n2^{n-1} - 2 \in \Theta(n2^n).$$

Este caso puede ser generalizado de la siguiente forma. Si $r_1, r_2, ..., r_k$ son las raíces de la ecuación característica de una ecuación en recurrencia homogénea, cada una de multiplicidad m_i, esto es, si la ecuación característica puede expresarse como:

$$(x - r_1)^{m_1} (x - r_2)^{m_2} ... (x - r_k)^{m_k} = 0,$$

entonces la solución a la ecuación en recurrencia viene dada por la expresión:

$$T(n) = \sum_{i=1}^{m_1} c_{1i} n^{i-1} r_1^n + \sum_{i=1}^{m_2} c_{2i} n^{i-1} r_2^n + \ldots + \sum_{i=1}^{m_k} c_{ki} n^{i-1} r_k^n \ .$$

1.4.2 Recurrencias no homogéneas

Consideremos una ecuación de la forma:

$$a_0 T(n) + a_1 T(n-1) + \ldots + a_k T(n-k) = b^n p(n)$$

donde los coeficientes a_i y b son números reales, y $p(n)$ es un polinomio en n de grado d. Una primera idea para resolver la ecuación es manipularla para convertirla en homogénea, como muestra el siguiente ejemplo.

Sea la ecuación $T(n) - 2T(n-1) = 3^n$ para $n \geq 2$, con las condiciones iniciales $T(0) = 0$ y $T(1) = 1$. En este caso $b = 3$ y $p(n) = 1$, polinomio en n de grado 0.

Podemos escribir la ecuación de dos formas distintas. En primer lugar, para $n+1$ tenemos que

$$T(n+1) - 2T(n) = 3^{n+1}.$$

Pero si multiplicamos por 3 la ecuación original obtenemos:

$$3T(n) - 6T(n-1) = 3^{n+1}$$

Restando ambas ecuaciones, conseguimos

$$T(n+1) - 5T(n) + 6T(n-1) = 0,$$

que resulta ser una ecuación homogénea cuya solución, aplicando lo visto anteriormente, es

$$T(n) = 3^n - 2^n \in \Theta(3^n).$$

Estos cambios son, en general, difíciles de ver. Afortunadamente, para este tipo de ecuaciones también existe una fórmula general para resolverlas, buscando sus soluciones entre las funciones que son combinaciones lineales de exponenciales, en donde se demuestra que la ecuación característica es de la forma:

$$(a_0 x^k + a_1 x^{k-1} + a_2 x^{k-2} + \ldots + a_k)(x-b)^{d+1} = 0\,,$$

lo que permite resolver el problema de forma similar a los casos anteriores.

Como ejemplo, veamos cómo se resuelve la ecuación en recurrencia que plantea el algoritmo de las torres de Hanoi:

$$T(n) = 2T(n-1) + n.$$

Su ecuación característica es entonces $(x-2)(x-1)^2 = 0$, y por tanto

$$T(n) = c_1 2^n + c_2 1^n + c_3 n 1^n \in \Theta(2^n).$$

Generalizando este proceso, supongamos ahora una ecuación de la forma:

$$a_0 T(n) + a_1 T(n-1) + ... + a_k T(n-k) = b_1^n p_1(n) + b_2^n p_2(n) + ... + b_s^n p_s(n)$$

donde como en el caso anterior, los coeficientes a_i y b_i son números reales y $p_j(n)$ son polinomios en n de grado d_j. En este caso también existe una forma general de la solución, en donde se demuestra que la ecuación característica es:

$$(a_0 x^k + a_1 x^{k-1} + a_2 x^{k-2} + ... + a_k)(x - b_1)^{d_1+1}(x - b_2)^{d_2+1}...(x - b_s)^{d_s+1} = 0.$$

Como ejemplo, supongamos la ecuación

$$T(n) = 2T(n-1) + n + 2^n,\ n \geq 1,$$

con la condición inicial $T(0) = 1$. En este caso tenemos que $b_1 = 1$, $p_1(n) = n$, $b_2 = 2$ y $p_2(n) = 1$, por lo que su ecuación característica es $(x-2)^2(x-1)^2 = 0$, lo que da lugar a la expresión final de $T(n)$:

$$T(n) = -2 - n + 2^{n+1} + n2^n \in \Theta(n2^n).$$

1.4.3 Cambio de Variable

Esta técnica se aplica cuando n es potencia de un número real a, esto es, $n = a^k$. Sea por ejemplo, para el caso $a = 2$, la ecuación $T(n) = 4T(n/2) + n$, donde n es una potencia de 2 ($n > 3$), $T(1) = 1$, y $T(2) = 6$.
Si $n = 2^k$ podemos escribir la ecuación como:

$$T(2^k) = 4T(2^{k-1}) + 2^k.$$

Haciendo el cambio de variable $t_k = T(2^k)$ obtenemos la ecuación

$$t_k = 4t_{k-1} + 2^k$$

que corresponde a una de las ecuaciones estudiadas anteriormente, cuya solución viene dada por la expresión

$$t_k = c_1(2^k)^2 + c_2 2^k.$$

Deshaciendo el cambio que realizamos al principio obtenemos que

$$T(n) = c_1 n^2 + c_2 n.$$

Calculando entonces las constantes a partir de las condiciones iniciales:

$$T(n) = 2n^2 - n \in \Theta(n^2).$$

1.4.4 Recurrencias No Lineales

En este caso, la ecuación que relaciona $T(n)$ con el resto de los términos no es lineal. Para resolverla intentaremos convertirla en una ecuación lineal como las que hemos estudiado hasta el momento.

Por ejemplo, sea la ecuación $T(n) = nT^2(n/2)$ para n potencia de 2, $n > 1$, con la condición inicial $T(1) = 1/3$. Llamando $t_k = T(2^k)$, la ecuación queda como

$$t_k = T(2^k) = 2^k T^2(2^{k-1}) = 2^k t_{k-1}^2,$$

que no corresponde a ninguno de los tipos estudiados. Necesitamos hacer un cambio más para transformar la ecuación. Tomando logaritmos a ambos lados y haciendo el cambio $u_k = \log t_k$ obtenemos

$$u_k - 2u_{k-1} = k,$$

ecuación en recurrencia no homogénea cuya ecuación característica asociada es $(x-2)(x-1)^2 = 0$. Por tanto,

$$u_k = c_1 2^k + c_2 + c_3 k.$$

Necesitamos ahora deshacer los cambios hechos. Primero $u_k = \log t_k$

$$t_k = 2^{c_1 2^k + c_2 + c_3 k}$$

y después $t_k = T(2^k)$. En consecuencia

$$T(n) = 2^{c_1 n + c_2 + c_3 \log n}.$$

Para calcular las constantes necesitamos las condiciones iniciales. Como sólo disponemos de una y tenemos tres incógnitas, usamos la ecuación en recurrencia original para obtener las restantes:

$$T(2) = 2T^2(1) = 2/9.$$
$$T(4) = 4T^2(2) = 16/81.$$

Con esto llegamos a que $c_1 = \log(4/3) = 2 - \log 3$, $c_2 = -2$, $c_3 = -1$ y por consiguiente:

$$T(n) = \frac{2^{2n}}{4n3^n}.$$

1.5 PROBLEMAS PROPUESTOS

1.1. De las siguientes afirmaciones, indicar cuales son ciertas y cuales no:

(*i*)	$n^2 \in O(n^3)$	(*ix*)	$n^2 \in \Omega(n^3)$
(*ii*)	$n^3 \in O(n^2)$	(*x*)	$n^3 \in \Omega(n^2)$
(*iii*)	$2^{n+1} \in O(2^n)$	(*xi*)	$2^{n+1} \in \Omega(2^n)$
(*iv*)	$(n+1)! \in O(n!)$	(*xii*)	$(n+1)! \in \Omega(n!)$
(*v*)	$f(n) \in O(n) \Rightarrow 2^{f(n)} \in O(2^n)$	(*xiii*)	$f(n) \in \Omega(n) \Rightarrow 2^{f(n)} \in \Omega(2^n)$
(*vi*)	$3^n \in O(2^n)$	(*xiv*)	$3^n \in \Omega(2^n)$
(*vii*)	$\log n \in O(n^{1/2})$	(*xv*)	$\log n \in \Omega(n^{1/2})$
(*viii*)	$n^{1/2} \in O(\log n)$	(*xvi*)	$n^{1/2} \in \Omega(\log n)$

1.2. Sea a una constante real, $0<a<1$. Usar las relaciones $\subset$ y = para ordenar los órdenes de complejidad de las siguientes funciones: $n\log n$, $n^2\log n$, n^8, n^{1+a}, $(1+a)^n$, $(n^2+8n+\log^3 n)^4$, $n^2/\log n$, 2^n.

1.3. La siguiente ecuación recurrente representa un caso típico de un algoritmo recursivo:

$$T(n) = \begin{cases} cn^k & \text{si } 1 \le n \le b \\ aT(n-b) + cn^k & \text{si } n > b \end{cases}$$

donde a,c,k son números reales, n,b son números naturales, y $a>0$, $c>0$, $k\ge 0$. En general, la constante a representa el número de llamadas recursivas que se realizan para un problema de tamaño n en cada ejecución del algoritmo; $n–b$ es el tamaño de los subproblemas generados; y cn^k representa el coste de las instrucciones del algoritmo que no son llamadas recursivas.

Demostrar que $T(n) \in \begin{cases} \Theta(n^k) & \text{si } a < 1 \\ \Theta(n^{k+1}) & \text{si } a = 1 \\ \Theta(a^{n \operatorname{div} b}) & \text{si } a > 1 \end{cases}$

1.4. La siguiente ecuación recurrente representa un caso típico de *Divide y Vencerás*:

$$T(n) = \begin{cases} cn^k & \text{si } 1 \le n < b \\ aT(n/b) + cn^k & \text{si } n \ge b \end{cases}$$

donde a,c,k son números reales, n,b son números naturales, y $a>0$, $c>0$, $k\ge 0$, $b>1$. La expresión cn^k representa en general el coste de descomponer el problema inicial en a subproblemas y el de componer las soluciones para producir la solución del problema original.

Demostrar que $T(n) \in \begin{cases} \Theta(n^k) & \text{si } a < b^k \\ \Theta(n^k \log n) & \text{si } a = b^k \\ \Theta(n^{\log_b a}) & \text{si } a > b^k \end{cases}$

1.5. Supongamos que disponemos de la siguiente definición de tipo:

```
CONST n = ...;
TYPE vector = ARRAY [1..n] OF INTEGER;
```

Consideramos entonces los procedimientos y funciones siguientes:

```
PROCEDURE Algoritmo1(VAR a:vector);
 VAR i,j:CARDINAL;
     temp:INTEGER;
BEGIN
 FOR i:=1 TO n-1 DO                  (* 1 *)
    FOR j:=n TO i+1 BY -1 DO         (* 2 *)
       IF a[j-1]>a[j] THEN           (* 3 *)
          temp:=a[j-1];              (* 4 *)
          a[j-1]:=a[j];              (* 5 *)
          a[j]:=temp                 (* 6 *)
       END                           (* 7 *)
    END                              (* 8 *)
 END                                 (* 9 *)
END Algoritmo1;


PROCEDURE Algoritmo2(VAR a:vector;c:INTEGER):CARDINAL;
 VAR inf,sup,i:CARDINAL;
BEGIN
 inf:=1; sup:=n;                     (* 1 *)
 WHILE (sup>=inf) DO                 (* 2 *)
    i:=(inf+sup) DIV 2;              (* 3 *)
    IF a[i]=c THEN RETURN i          (* 4 *)
    ELSIF c<a[i] THEN sup:=i-1       (* 5 *)
    ELSE inf:=i+1                    (* 6 *)
    END                              (* 7 *)
 END;                                (* 8 *)
 RETURN 0;                           (* 9 *)
END Algoritmo2;
```

```
PROCEDURE Euclides(m,n:CARDINAL):CARDINAL;
 VAR temp:CARDINAL;
BEGIN
 WHILE m>0 DO                          (* 1 *)
    temp:=m;                           (* 2 *)
    m:=n MOD m;                        (* 3 *)
    n:=temp                            (* 4 *)
 END;                                  (* 5 *)
 RETURN n                              (* 6 *)
END Euclides;


PROCEDURE Misterio(n:CARDINAL);
 VAR i,j,k,s:INTEGER;
BEGIN
 s:=0;                                 (* 1 *)
 FOR i:=1 TO n-1 DO                    (* 2 *)
    FOR j:=i+1 TO n DO                 (* 3 *)
       FOR k:=1 TO j DO                (* 4 *)
          s:=s+2                       (* 5 *)
       END                             (* 6 *)
    END                                (* 7 *)
 END                                   (* 8 *)
END Misterio;
```

a) Calcular sus tiempos de ejecución en el mejor, peor, y caso medio.
b) Dar cotas asintóticas O, Ω y Θ para las funciones anteriores.

1.6. Demostrar las siguientes inclusiones estrictas: $O(1) \subset O(\log n) \subset O(n) \subset O(n\log n) \subset O(n^2) \subset O(n^3) \subset O(n^k) \subset O(2^n) \subset O(n!)$.

1.7. a) Demostrar que $f \in O(g) \Leftrightarrow g \in \Omega(f)$.
b) Dar un ejemplo de funciones f y g tales que $f \in O(g)$ pero que $f \notin \Omega(g)$.
c) Demostrar que $\forall a,b>1$ se tiene que $\log_a n \in \Theta(\log_b n)$.

1.8. Considérense las siguientes funciones de n:

$$f_1(n) = n^2; \qquad f_2(n) = n^2 + 1000n;$$

$$f_3(n) = \begin{cases} n, & \text{si } n \text{ impar} \\ n^3, & \text{si } n \text{ par} \end{cases}; \quad f_4(n) = \begin{cases} n, & \text{si } n \leq 100 \\ n^3, & \text{si } n > 100 \end{cases}$$

Para cada posible valor de i,j indicar si $f_i \in O(f_j)$ y si $f_i \in \Omega(f_j)$.

1.9. Resolver las siguientes ecuaciones y dar su orden de complejidad:

a) $T(n)=3T(n-1)+4T(n-2)$ si $n>1$; $T(0)=0$; $T(1)=1$.

b) $T(n)=2T(n-1)-(n+5)3^n$ si $n>0$; $T(0)=0$.

c) $T(n)=4T(n/2)+n^2$ si $n>4$, n potencia de 2; $T(1)=1$; $T(2)=8$.

d) $T(n)=2T(n/2)+n\log n$ si $n>1$, n potencia de 2.

e) $T(n)=3T(n/2)+5n+3$ si $n>1$, n potencia de 2.

f) $T(n)=2T(n/2)+\log n$ si $n>1$, n potencia de 2.

g) $T(n)=2T(n^{1/2})+\log n$ con $n=2^{2^k}$; $T(2)=1$.

h) $T(n)=5T(n/2)+(n\log n)^2$ si $n>1$, n potencia de 2; $T(1)=1$.

i) $T(n)=T(n-1)+2T(n-2)-2T(n-3)$ si $n>2$; $T(n)=9n^2-15n+106$ si $n=0,1,2$.

j) $T(n)=(3/2)T(n/2)-(1/2)T(n/4)-(1/n)$ si $n>2$; $T(1)=1$; $T(2)=3/2$.

k) $T(n)=2T(n/4)+n^{1/2}$ si $n>4$, n potencia de 4.

l) $T(n)=4T(n/3)+n^2$ si $n>3$, n potencia de 3.

1.10. Suponiendo que $T_1 \in O(f)$ y que $T_2 \in O(f)$, indicar cuáles de las siguientes afirmaciones son ciertas:

a) $T_1 + T_2 \in O(f)$.

b) $T_1 - T_2 \in O(f)$.

c) $T_1 / T_2 \in O(1)$.

d) $T_1 \in O(T_2)$.

1.11. Encontrar dos funciones $f(n)$ y $g(n)$ tales que $f \notin O(g)$ y $g \notin O(f)$.

1.12. Demostrar que para cualquier constante k se verifica que $\log^k n \in O(n)$.

1.13. Consideremos los siguientes procedimientos y funciones sobre árboles. Calcular sus tiempos de ejecución y sus órdenes de complejidad.

```
PROCEDURE Inorden(t:arbol);     (* recorrido en inorden de t *)
BEGIN
 IF NOT Esvacio(t) THEN                                    (* 1
*)
    Inorden(Izq(t));                                      (* 2 *)
    Opera(Raiz(t));                                       (* 3 *)
    Inorden(Der(t))                                       (* 4 *)
 END;                                                     (* 5 *)
END Inorden;
```

```
PROCEDURE Altura(t:arbol):CARDINAL; (* altura de t *)
BEGIN
 IF Esvacio(t) THEN                                    (* 1 *)
    RETURN 0                                           (* 2 *)
 ELSE                                                  (* 3 *)
    RETURN 1+Max2(Altura(Izq(t)),Altura(Der(t)))       (* 4 *)
 END                                                   (* 5 *)
END Altura;

PROCEDURE Mezcla(t1,t2:arbol):arbol;
(* devuelve un arbol binario de busqueda con los elementos de
   los dos arboles binarios de busqueda t1 y t2. La funcion Ins
   inserta un elemento en un arbol binario de busqueda *)
BEGIN
 IF Esvacio(t1) THEN                                   (* 1 *)
  RETURN t2                                            (* 2 *)
 ELSIF Esvacio(t2) THEN                                (* 3 *)
  RETURN t1                                            (* 4 *)
 ELSE                                                  (* 5 *)
    RETURN Mezcla(Mezcla(Ins(t1,Raiz(t2)),Izq(t2)),
                  Der(t2))                             (* 6 *)
 END                                                   (* 7 *)
END Mezcla;
```

Supondremos que las operaciones básicas del tipo abstracto de datos *arbol* (*Raiz*, *Izq*, *Der*, *Esvacio*) son O(1), así como las operaciones *Opera* (que no es relevante lo que hace) y *Max2* (que calcula el máximo de dos números). Por otro lado, supondremos que la complejidad de la función *Ins* es O(log*n*).

1.14. Ordenar las siguientes funciones de acuerdo a su velocidad de crecimiento: n, $\sqrt{n}$, log$n$, loglog$n$, log^2n, n/log$n$, $\sqrt{n}$ log^2n, $(1/3)^n$, $(3/2)^n$, 17, n^2.

1.15. Resolver la ecuación $T(n) = \frac{1}{n}\left(\sum_{i=0}^{n-1} T(i)\right) + cn$, siendo $T(0) = 0$.

1.16. Consideremos las siguientes funciones:

```
CONST n = ...;
TYPE vector = ARRAY[1..n] OF INTEGER;
```

```
PROCEDURE BuscBin(VAR a:vector;
                  prim,ult:CARDINAL;x:INTEGER):BOOLEAN;
 VAR mitad:CARDINAL;
BEGIN
 IF (prim>=ult) THEN RETURN a[ult]=x              (* 1 *)
 ELSE                                             (* 2 *)
    mitad:=(prim+ult)DIV 2;                       (* 3 *)
    IF x=a[mitad] THEN RETURN TRUE                 (* 4 *)
    ELSIF (x<a[mitad]) THEN                       (* 5 *)
       RETURN BuscBin(a,prim,mitad-1,x)           (* 6 *)
    ELSE                                          (* 7 *)
       RETURN BuscBin(a,mitad+1,ult,x)            (* 8 *)
    END                                           (* 9 *)
 END                                              (* 10 *)
END BuscBin;

PROCEDURE Sumadigitos(num:CARDINAL):CARDINAL;
BEGIN
 IF num<10 THEN RETURN num                        (* 1 *)
 ELSE RETURN (num MOD 10)+Sumadigitos(num DIV 10) (* 2 *)
 END                                              (* 3 *)
END Sumadigitos;
```

a) Calcular sus tiempos de ejecución y sus órdenes de complejidad.
b) Modificar los algoritmos eliminando la recursión.
c) Calcular la complejidad de los algoritmos modificados y justificar para qué casos es más conveniente usar uno u otro.

1.17. Consideremos la siguiente función:

```
PROCEDURE Raro(VAR a:vector;prim,ult:CARDINAL):INTEGER;
 VAR mitad,terc:CARDINAL;
BEGIN
 IF (prim>=ult) THEN RETURN a[ult] END;
 mitad:=(prim+ult)DIV 2;     (* posicion central *)
 terc :=(ult-prim)DIV 3;     (* num. elementos DIV 3 *)
 RETURN a[mitad]+Raro(a,prim,prim+terc)+Raro(a,ult-terc,ult)
END Raro;
```

a) Calcular el tiempo de ejecución de la llamada a la función *Raro(a,*1*,n)*, suponiendo que n es potencia de 3.
b) Dar una cota de complejidad para dicho tiempo de ejecución.

1.6 SOLUCIÓN A LOS PROBLEMAS PROPUESTOS

Antes de comenzar con la resolución de los problemas es necesario hacer una aclaración sobre la notación utilizada para las funciones logarítmicas. A partir de ahora y a menos que se exprese explícitamente otra base, la función "log" hará referencia a logaritmos en base dos.

Solución al Problema 1.1 (☺/😐)

(i) $n^2 \in O(n^3)$ es cierto pues $lim_{n\to\infty} (n^2/n^3) = 0$.

(ii) $n^3 \in O(n^2)$ es falso pues $lim_{n\to\infty} (n^2/n^3) = 0$.

(iii) $2^{n+1} \in O(2^n)$ es cierto pues $lim_{n\to\infty} (2^{n+1}/2^n) = 2$.

(iv) $(n+1)! \in O(n!)$ es falso pues $lim_{n\to\infty} (n!/(n+1)!) = 0$.

(v) $f(n) \in O(n) \Rightarrow 2^{f(n)} \in O(2^n)$ es falso. Por ejemplo, sea $f(n) = 3n$; claramente $f(n) \in O(n)$ pero sin embargo $lim_{n\to\infty} (2^n/2^{3n}) = 0$, con lo cual $2^{3n} \notin O(2^n)$. De forma más general, resulta ser falso para cualquier función lineal de la forma $f(n) = \alpha n$ con $\alpha > 1$, y cierto para $f(n) = \beta n$ con $\beta \leq 1$.

(vi) $3^n \in O(2^n)$ es falso pues $lim_{n\to\infty} (2^n/3^n) = 0$.

(vii) $\log n \in O(n^{1/2})$ es cierto pues $lim_{n\to\infty} (\log n/n^{1/2}) = 0$.

(viii) $n^{1/2} \in O(\log n)$ es falso pues $lim_{n\to\infty} (\log n/n^{1/2}) = 0$.

(ix) $n^2 \in \Omega(n^3)$ es falso pues $lim_{n\to\infty} (n^2/n^3) = 0$.

(x) $n^3 \in \Omega(n^2)$ es cierto pues $lim_{n\to\infty} (n^2/n^3) = 0$.

(xi) $2^{n+1} \in \Omega(2^n)$ es cierto pues $lim_{n\to\infty} (2^{n+1}/2^n) = 2$.

(xii) $(n+1)! \in \Omega(n!)$ es cierto pues $lim_{n\to\infty} (n!/(n+1)!) = 0$.

(xiii) $f(n) \in \Omega(n) \Rightarrow 2^{f(n)} \in \Omega(2^n)$ es falso. Por ejemplo, sea $f(n) = (1/2)n$; claramente $f(n) \in O(n)$ pero sin embargo $lim_{n\to\infty} (2^{(1/2)n}/2^n) = 0$, con lo cual $2^{(1/2)n} \notin \Omega(2^n)$. De forma más general, resulta ser falso para cualquier función $f(n) = \alpha n$ con $\alpha < 1$, y cierto para $f(n) = \beta n$ con $\beta \geq 1$.

(xiv) $3^n \in \Omega(2^n)$ es cierto pues $lim_{n\to\infty} (2^n/3^n) = 0$.

(xv) $\log n \in \Omega(n^{1/2})$ es falso pues $lim_{n\to\infty}(\log n/n^{1/2}) = 0$.

(xvi) $n^{1/2} \in \Omega(\log n)$ es cierto pues $lim_{n\to\infty}(\log n/n^{1/2}) = 0$.

Solución al Problema 1.2 (☺)

- Respecto al orden de complejidad O tenemos que:

$$O(n\log n) \subset O(n^{1+a}) \subset O(n^2/\log n) \subset O(n^2\log n) \subset O(n^8) = O((n^2+8n+\log^3 n)^4) \subset O((1+a)^n) \subset O(2^n).$$

  Puesto que todas las funciones son continuas, para comprobar que $O(f)\subset O(g)$, basta ver que $lim_{n\to\infty}(f(n)/g(n)) = 0$, y para comprobar que $O(f) = O(g)$, basta ver que $lim_{n\to\infty}(f(n)/g(n))$ es finito y distinto de 0.

- Por otro lado, respecto al orden de complejidad Ω, obtenemos que:

$$\Omega(n\log n) \supset \Omega(n^{1+a}) \supset \Omega(n^2/\log n) \supset \Omega(n^2\log n) \supset \Omega(n^8) = \Omega((n^2+8n+\log^3 n)^4) \supset \Omega((1+a)^n) \supset \Omega(2^n)$$

  Para comprobar que $\Omega(f) \subset \Omega(g)$, basta ver que $lim_{n\to\infty}(g(n)/f(n)) = 0$, y para comprobar que $\Omega(f) = \Omega(g)$, basta ver que $lim_{n\to\infty}(f(n)/g(n))$ es finito y distinto de 0 puesto que al ser las funciones continuas tenemos garantizada la existencia de los límites.

- Y en lo relativo al orden de complejidad Θ, al definirse como la intersección de los órdenes O y Ω, sólo tenemos asegurado que:

$$\Theta(n^8) = \Theta((n^2+8n+\log^3 n)^4),$$

  siendo los órdenes Θ del resto de las funciones conjuntos no comparables.

Solución al Problema 1.3 (🖋)

La ecuación dada puede ser también escrita como $T(n) - aT(n-b) = cn^k$, ecuación en recurrencia no homogénea cuya ecuación característica es:

$$(x^b-a)(x-1)^{k+1} = 0.$$

- Para estudiar las raíces de esa ecuación, vamos a suponer primero que $a \neq 1$. En este caso, la ecuación tiene una raíz de multiplicidad $k+1$ (el 1), y b raíces

distintas $r_1,r_2,...,r_b$ (las b raíces b-ésimas de a[†]). Entonces la solución de la ecuación en recurrencia es de la forma:

$$T(n) = c_1 1^n + c_2 n 1^n + c_3 n^2 1^n + ... + c_{k+1} n^k 1^n + d_1 r_1^n + d_2 r_2^n + ... + d_b r_b^n =$$
$$= \left(\sum_{i=1}^{k+1} c_i n^{i-1} \right) + \left(\sum_{i=1}^{b} d_i r_i^n \right)$$

siendo c_i y d_i coeficientes reales.

– Si a<1, las raíces b-ésimas de a (esto es, las r_i) son menores en módulo que 1, con lo cual el segundo sumatorio tiende a cero cuando n tiende a ∞, y en consecuencia $T(n) \in \Theta(n^k)$ pues $\lim_{n\to\infty} \frac{T(n)}{n^k} = c_{k+1}$ es finito y distinto de cero.

  Para ver que efectivamente c_{k+1} es distinto de cero independientemente de las condiciones iniciales, sustituimos esta expresión de $T(n)$ en la ecuación original, $T(n) – aT(n–b) = cn^k$, y por tanto:

$$\left(\sum_{i=1}^{k+1} c_i n^{i-1} + \sum_{i=1}^{b} d_i r_i^n \right) - a \left(\sum_{i=1}^{k+1} c_i (n-b)^{i-1} + \sum_{i=1}^{b} d_i r_i^n \right) = cn^k .$$

  Igualando ahora los coeficientes que acompañan a n^k obtenemos que $c_{k+1} – ac_{k+1} = c$, o lo que es igual, $(1–a)c_{k+1} = c$. Ahora bien, como sabemos que $a < 1$ y $c > 0$, entonces c_{k+1} no puede ser cero.

– Si $a > 1$, las funciones del segundo sumatorio son exponenciales, mientras que las primeras se mantienen dentro de un orden polinomial, por lo que en este caso el orden de complejidad del algoritmo es exponencial. Ahora bien, como todas las raíces b-ésimas de a tienen el mismo módulo, todas crecen de la misma forma y por tanto todas son del mismo orden de complejidad, obteniendo que

$$\Theta(r_1^n) = \Theta(r_2^n) = ... = \Theta(r_b^n) .$$

  Como $\lim_{n\to\infty} \frac{T(n)}{r_1^n} = d_1$ es distinto de cero y finito, podemos concluir que:

$$T(n) \in \Theta(r_1^n) = \Theta\left(\left(\sqrt[b]{a} \right)^n \right) = \Theta(a^{n \operatorname{div} b}) .$$

Hemos supuesto que $d_1 \neq 0$. Esto no tiene por qué ser necesariamente cierto para todas las condiciones iniciales, aunque sin embargo sí es cierto que al menos uno de los coeficientes d_i ha de ser distinto de cero. Basta tomar ese sumando para demostrar lo anterior.

[†] Recordemos que dados dos números reales a y b, la solución de la ecuación $x^b – a = 0$ tiene b raíces distintas, que pueden ser expresadas como $a^{1/b}e^{2\pi ik/n}$, para k=0,1,2,...,n–1.

- Supongamos ahora que $a = 1$. En este caso la multiplicidad de la raíz 1 es k+2, con lo cual

$$T(n) = c_1 1^n + c_2 n 1^n + c_3 n^2 1^n + \ldots + c_{k+2} n^{k+1} 1^n + d_2 r_2^n + \ldots + d_b r_b^n =$$
$$= \left(\sum_{i=1}^{k+2} c_i n^{i-1} \right) + \left(\sum_{i=2}^{b} d_i r_i^n \right)$$

Pero las raíces $r_2, r_3, \ldots, r_b$ son todas de módulo 1 (obsérvese que r_1=1), y por tanto el segundo sumando de $T(n)$ es de complejidad $\Theta(1)$.

Así, el crecimiento de $T(n)$ coincide con el del primer sumando, que es un polinomio de grado k+1 con lo cual $T(n) \in \Theta(n^{k+1})$.

Solución al Problema 1.4 (𝒢)

Haciendo el cambio $n = b^m$, o lo que es igual, $m = \log_b n$, obtenemos que

$$T(b^m) = aT(b^{m-1}) + cb^{mk}.$$

Llamando $t_m = T(b^m)$, la ecuación queda como

$$t_m - at_{k-1} = c(b^k)^m,$$

ecuación en recurrencia no homogénea con ecuación característica $(x-a)(x-b^k) = 0$.

Para resolver esta ecuación, supongamos primero que $a = b^k$. Entonces, la ecuación característica es $(x-b^k)^2 = 0$ y por tanto

$$t_m = c_1 b^{km} + c_2 m b^{km}.$$

Necesitamos ahora deshacer los cambios hechos. Primero $t_m = T(b^m)$ con lo que

$$T(b^m) = c_1 b^{km} + c_2 m b^{km} = (c_1 + c_2 m) b^{km},$$

y después $n = b^m$, obteniendo finalmente que

$$T(n) = (c_1 + c_2 \log_b n) n^k \in \Theta(n^k \log n).^{\ddagger}$$

Supongamos ahora el caso contrario, $a \neq b^k$. Entonces la ecuación característica tiene dos raíces distintas, y por tanto

$$t_m = c_1 a^m + c_2 b^{km}.$$

Necesitamos deshacer los cambios hechos. Primero $t_m = T(b^m)$, con lo que

$$T(b^m) = c_1 a^m + c_2 b^{km},$$

y después $n = b^m$, obteniendo finalmente que

$$T(n) = c_1 a^{\log_b n} + c_2 n^k = c_1 n^{\log_b a} + c_2 n^k.$$

---

‡ Obsérvese que se hace uso de que $\log_b n \in \Theta(\log n)$, lo que se demuestra en el problema 1.7.

En consecuencia, si $\log_b a > k$ (si y sólo si $a > b^k$) entonces $T(n) \in \Theta(n^{\log_b a})$. Si no, es decir, $a < b^k$, entonces $T(n) \in \Theta(n^k)$.

Solución al Problema 1.5

Procedimiento *Algoritmo1* (☺)

a) Para obtener el tiempo de ejecución, calcularemos primero el número de operaciones elementales (OE) que se realizan:

- En la línea (1) se ejecutan 3 OE (una asignación, una resta y una comparación) en cada una de las iteraciones del bucle más otras 3 al final, cuando se efectúa la salida del *FOR*.
- Igual ocurre con la línea (2), también con 3 OE (una asignación, una suma y una comparación) por iteración, más otras 3 al final del bucle.
- En la línea (3) se efectúa una condición, con un total de 4 OE (una diferencia, dos accesos a un vector, y una comparación).
- Las líneas (4) a (6) sólo se ejecutan si se cumple la condición de la línea (3), y realizan un total de 9 OE: 3, 4 y 2 respectivamente.

Con esto:

En el *caso mejor* para el algoritmo la condición de la línea (3) será siempre falsa, y no se ejecutarán nunca las líneas (4), (5) y (6). Así, el bucle más interno realizará (n–i) iteraciones, cada una de ellas con 4 OE (línea 3), más las 3 OE de la línea (2). Por tanto, el bucle más interno realiza un total de

$$\left(\sum_{j=i+1}^{n}(4+3)\right)+3 = 7\left(\sum_{j=i+1}^{n}1\right)+3 = 7(n-i)+3$$

OE, siendo el 3 adicional por la condición de salida del bucle.

A su vez, el bucle externo repetirá esas $7(n–i)+3$ OE en cada iteración, lo que hace que el número de OE que se realizan en el algoritmo sea:

$$T(n) = \left(\sum_{i=1}^{n-1}\big(7(n-i)+3\big)+3\right)+3 = \frac{7}{2}n^2 + \frac{5}{2}n - 3\,.$$

- En el *caso peor*, la condición de la línea (3) será siempre verdadera, y las líneas (4), (5) y (6) se ejecutarán en todas las iteraciones. Por tanto, el bucle más interno realiza

$$\left(\sum_{j=i+1}^{n}(4+9+3)\right)+3 = 16(n-i)+3$$

OE. El bucle externo realiza aquí el mismo número de iteraciones que en el caso anterior, por lo que el número de OE en este caso es:

$$T(n) = \left(\sum_{i=1}^{n-1} \big(16(n-i)+3\big) + 3 \right) + 3 = 8n^2 - 2n - 3 .$$

- En el *caso medio*, la condición de la línea (3) será verdadera con probabilidad 1/2. Así, las líneas (4), (5) y (6) se ejecutarán en la mitad de las iteraciones del bucle más interno, y por tanto realiza

$$\left(\sum_{j=i+1}^{n} (4 + \frac{1}{2}9) + 3 \right) + 3 = \frac{23}{2}(n-i) + 3$$

  OE. El bucle externo realiza aquí el mismo número de iteraciones que en el caso anterior, por lo que el número de OE en este caso es:

$$T(n) = \left(\sum_{i=1}^{n-1} \left(\frac{23}{2}(n-i) + 3 \right) + 3 \right) + 3 = \frac{23}{4}n^2 + \frac{1}{4}n - 3 .$$

b) Como los tiempos de ejecución en los tres casos son polinomios de grado 2, la complejidad del algoritmo es cuadrática, independientemente de qué caso se trate.

Obsérvese cómo hemos analizado el tiempo de ejecución del algoritmo sólo en función de su código y no respecto a lo que hace, puesto que en muchos casos esto nos llevaría a conclusiones erróneas. Debe ser a posteriori cuando se analice el objetivo para el que fue diseñado el algoritmo.

En el caso que nos ocupa, un examen más detallado del código del procedimiento nos muestra que el algoritmo está diseñado para ordenar de forma creciente el vector que se le pasa como parámetro, siguiendo el método de la Burbuja. Lo que acabamos de ver es que sus casos mejor, peor y medio se producen respectivamente cuando el vector está inicialmente ordenado de forma creciente, decreciente y aleatoria.

Función *Algoritmo2* (𝒢)

a) Para calcular el tiempo de ejecución, calcularemos primero el número de operaciones elementales (OE) que se realizan:

- En la línea (1) se ejecutan 2 OE (dos asignaciones).
- En la línea (2) se efectúa la condición del bucle, que supone 1 OE (la comparación).
- Las líneas (3) a (6) componen el cuerpo del bucle, y contabilizan 3, 2+1, 2+2 y 2 OE respectivamente. Es importante hacer notar que el bucle también puede finalizar si se verifica la condición de la línea (4).
- Por último, la línea (9) supone 1 OE. A ella se llega cuando la condición del bucle *WHILE* deja de ser cierta.

Con esto:

- En el *caso mejor* se efectuarán solamente la líneas (1), (2), (3) y (4). En consecuencia, $T(n) = 2+1+3+3 = 9$.
- En el *caso peor* se efectúa la línea (1), y después se repite el bucle hasta que su condición sea falsa, acabando la función al ejecutarse la línea (9). Cada iteración del bucle está compuesta por las líneas (2) a (8), junto con una ejecución adicional de la línea (2) que es la que ocasiona la salida del bucle. En cada iteración se reducen a la mitad los elementos a considerar, por lo que el bucle se repite log$n$ veces. Por tanto,

$$T(n) = 2 + \left(\left(\sum_{i=1}^{\log n} (1+3+2+2+2) \right) + 1 \right) + 1 = 10 \log n + 4 .$$

- En el *caso medio*, necesitamos calcular el número medio de veces que se repite el bucle, y para esto veamos cuántas veces puede repetirse, y qué probabilidad tiene cada una de suceder.

  Por un lado, el bucle puede repetirse desde una vez hasta log$n$ veces, puesto que en cada iteración se divide por dos el número de elementos considerados. Si se repitiese una sola vez, es que el elemento ocuparía la posición $n/2$, lo que ocurre con una probabilidad $1/(n+1)$. Si el bucle se repitiese dos veces es que el elemento ocuparía alguna de las posiciones $n/4$ ó $3n/4$, lo cual ocurre con probabilidad $1/(n+1)+1/(n+1)=2/(n+1)$. En general, si se repitiese i veces es que el elemento ocuparía alguna de las posiciones $nk/2^i$, con k impar y $1 \le k < 2^i$.

  Es decir, el bucle se repite i veces con probabilidad $2^{i-1}/(n+1)$. Por tanto, el número medio de veces que se repite el ciclo vendrá dado por la expresión:

$$\sum_{i=1}^{\log n} i \frac{2^{i-1}}{n+1} = \frac{n \log n - n + 1}{n+1} .$$

  Con esto, la función ejecuta la línea (1) y después el bucle se repite ese número medio de veces, saliendo por la instrucción *RETURN* en la línea (4). Por consiguiente,

$$T(n) = 2 + \left(\frac{n \log n - n + 1}{n+1} \right)(1+3+2+2) + (1+3+3) = 9 + 8 \frac{n \log n - n + 1}{n+1}$$

c) En el caso mejor el tiempo de ejecución es una constante. Para los casos peor y medio, la complejidad resultante es de orden $\Theta(\log n)$ puesto que

$$\lim_{n \to \infty} \frac{T(n)}{\log n}$$

es una constante finita y distinta de cero en ambos casos (10 y 8 respectivamente).

Función *Euclides* (𝒢)

a) En este caso el análisis del tiempo de ejecución y la complejidad de la función sigue un proceso distinto al estudiado en los casos anteriores.

Lo primero es resaltar algunas características del algoritmo, siguiendo una línea de razonamiento similar a la de [BRA97]:

[1] Para cualquier par de enteros no negativos m y n tales que $n \geq m$, se verifica que n MOD $m < n/2$. Veámoslo:
   a) Si $m > n/2$ entonces $1 \leq n/m < 2$ y por tanto n DIV $m = 1$, lo que implica que n MOD $m = n - m(n \text{ DIV } m) = n - m < n - n/2 = n/2$.
   b) Por otro lado, si $m \leq n/2$ entonces n MOD $m < m \leq n/2$.

[2] Podemos suponer sin pérdida de generalidad que $n \geq m$. Si no, la primera iteración del bucle intercambia n con m ya que n MOD $m = n$ cuando $n < m$. Además, la condición $n \geq m$ se conserva siempre (es decir, es un invariante del bucle) pues n MOD m nunca es mayor que m.

[3] El cuerpo del bucle efectúa 4 OE, con lo cual el tiempo del algoritmo es del orden exacto del número de iteraciones que realiza el bucle. Por consiguiente, para determinar la complejidad del algoritmo es suficiente acotar este número.

[4] Una propiedad curiosa de este algoritmo es que no se produce un avance notable con cada iteración del bucle, sino que esto ocurre cada dos iteraciones. Consideremos lo que les ocurre a m y n cuando el ciclo se repite dos veces, suponiendo que no acaba antes. Sean m_0 y n_0 los valores originales de los parámetros, que podemos suponer $n_0 \geq m_0$ por [2]. Después de la primera iteración, m vale n_0 MOD m_0. Después de la segunda iteración, n toma ese valor, y por tanto ya es menor que $n_0/2$ (por [1]). En consecuencia, n vale menos de la mitad de lo que valía tras dos iteraciones del bucle. Como se sigue manteniendo que $n \geq m$, el mismo razonamiento se puede repetir para las siguientes dos iteraciones, y así sucesivamente.

El hecho de que n valga menos de la mitad cada dos iteraciones del bucle es el que nos permite intuir que el bucle se va a repetir del orden de $2\log n$ veces. Vamos a demostrar esto formalmente.

Para ello, vamos a tratar el bucle como si fuera un algoritmo recursivo. Sea $T(l)$ el número máximo de veces que se repite el bucle para valores iniciales m y n cuando $m \leq n \leq l$. En este caso l representa el tamaño de la entrada.

- Si $n \leq 2$ el bucle no se repite (si $m = 0$) o se hace una sola vez (si m es 1 ó 2).
- Si $n > 2$ y m=1 o bien m divide a n, el bucle se repite una sola vez.
- En otro caso ($n > 2$ y m no divide a n) el bucle se ejecuta dos veces, y por lo visto en [4], n vale a lo sumo la mitad de lo que valía inicialmente. En consecuencia $n \leq (l/2)$, y además m se sigue manteniendo por debajo de n.

Esto nos lleva a la ecuación en recurrencia $T(l) \leq 2 + T(l/2)$ si $l > 2$, $T(l) \leq 1$ si $l \leq 2$, lo que implica que el algoritmo de Euclides es de complejidad logarítmica respecto al tamaño de la entrada (l).

Nos preguntaremos la razón de usar $T(l)$ para acotar el número de iteraciones que realiza el algoritmo en vez de definir T directamente como una función de n, el mayor de los dos operandos, lo cual sería mucho más intuitivo.

El problema es que si definimos $T(n)$ como el número de iteraciones que realiza el algoritmo para los valores $m \leq n$, no podríamos concluir que $T(n) \leq 2 + T(n/2)$ del hecho de que n valga la mitad de su valor tras cada dos iteraciones del bucle.

Por ejemplo, para *Euclides(*8,13*)*, obtenemos que $T(13) = 5$ en el peor caso, mientras que $T(13/2) = T(6) = 2$. Esto ocurre porque tras dos iteraciones del bucle n no vale 6, sino 5 (y $m = 3$), y con esto sí es cierto que $T(13) \leq 2 + T(5)$ ya que $T(5) = 3$.

La raíz de este problema es que esta nueva definición más intuitiva de T no lleva a una función monótona no decreciente ($T(5) > T(6)$) y por tanto la existencia de algún $n' \leq n/2$ tal que $T(n) \leq 2 + T(n')$ no implica necesariamente que $T(n) \leq 2 + T(n/2)$.

En vez de esto, solamente podríamos afirmar que $T(n) \leq 2 + max\{T(n') | n' \leq n/2\}$, que es una ecuación en recurrencia bastante difícil de resolver. Esa es la razón de que escogiésemos nuestra función T de forma que fuera no decreciente y que expresara una cota superior del número de iteraciones.

Para acabar, es interesante hacer notar una característica curiosa de este algoritmo: se demuestra que su caso peor ocurre cuando m y n son dos términos consecutivos de la sucesión de Fibonacci.

b) $T(l) \in \Theta(\log l)$ como se deduce de la ecuación en recurencia que define el tiempo de ejecución del algoritmo.

Procedimiento *Misterio* (☺)

a) En este caso son tres bucles anidados los que se ejecutan, independientemente de los valores de la entrada, es decir, no existe peor, medio o mejor caso, sino un único caso.

Para calcular el tiempo de ejecución, veamos el número de operaciones elementales (OE) que se realizan:

- En la línea (1) se ejecuta 1 OE (una asignación).
- En la línea (2) se ejecutarán 3 OE (una asignación, una resta y una comparación) en cada una de las iteraciones del bucle más otras 3 al final, cuando se efectúa la salida del *FOR*.
- Igual ocurre con la línea (3), también con 3 OE (una asignación, una suma y una comparación) por iteración, más otras 3 al final del bucle.
- Y también en la línea (4), esta vez con 2 OE (asignación y comparación) más las 2 adicionales de terminación del bucle.
- Por último, la línea (5) supone 2 OE (un incremento y una asignación).

Con esto, el bucle interno se ejecutará j veces, el medio $(n–i)$ veces, y el bucle exterior $(n–1)$ veces, lo que conlleva un tiempo de ejecución de:

$$T(n) = 1 + \left(\sum_{i=1}^{n-1} \left(3 + \left(\sum_{j=i+1}^{n} \left(3 + \left(\sum_{k=1}^{j} (2+2) \right) + 2 \right) \right) + 3 \right) \right) + 3 =$$

$$1 + \left(\sum_{i=1}^{n-1} \left(3 + \left(\sum_{j=i+1}^{n} \left(3 + (4j) + 2\right) \right) + 3 \right) \right) + 3 = 1 + \left(\sum_{i=1}^{n-1} \left(3 + \left(\sum_{j=i+1}^{n} \left(4j + 5\right) \right) + 3 \right) \right) + 3 =$$

$$1 + \left(\sum_{i=1}^{n-1} \left(3 + \left(2(n+i) + 7\right)\left(n - i\right) + 3\right) \right) + 3 =$$

$$1 + \frac{8n^3 + 15n^2 + 13n - 36}{6} + 3 = \frac{4}{3}n^3 + \frac{15}{6}n^2 + \frac{13}{6}n - 2 .$$

b) Como el tiempo de ejecución es un polinomio de grado 3, la complejidad del algoritmo es de orden $\Theta(n^3)$.

Solución al Problema 1.6 (☺)

Para comprobar que $O(f) \subset O(g)$ en cada caso y que esa inclusión es estricta, basta ver que $lim_{n\to\infty}(f(n)/g(n)) = 0$, pues todas las funciones son continuas y por tanto los límites existen. Por consiguiente,

$O(1) \subset O(\log n) \subset O(n) \subset O(n\log n) \subset O(n^2) \subset O(n^3) \subset O(n^k) \subset O(2^n) \subset O(n!)$.

Solución al Problema 1.7 (☺)

a) Por la definición de O, sabemos que $f \in O(g)$ si y sólo si existen $c_1 > 0$ y n_1 tales que $f(n) \le c_1 g(n)$ para todo $n \ge n_1$.

Análogamente, por la definición de Ω tenemos que $g \in \Omega(f)$ si y sólo si existen $c_2 > 0$ y n_2 tales que $g(n) \ge c_2 f(n)$ para todo $n \ge n_2$. Por consiguiente,

$\Rightarrow$) Si $f \in O(g)$ basta tomar $c_2 = 1/c_1$ y $n_2 = n_1$ para ver que $g(n) \in \Omega(f)$.

$\Leftarrow$) Recíprocamente, si $g \in \Omega(f)$ basta tomar $c_1 = 1/c_2$ y $n_1 = n_2$ para que $f \in O(g)$.

Obsérvese que esto es posible pues c_1 y c_2 son ambos estrictamente mayores que cero, y por tanto poseen inverso.

b) Sean $f(n) = \begin{cases} n^2 & \text{si } n \text{ es par.} \\ 1 & \text{si } n \text{ es impar.} \end{cases}$ y $g(n) = n^2$.

Entonces $\Theta(g) = \Theta(n^2)$, y por otro lado $O(f) = O(n^2)$, con lo cual $f \in O(n^2) = O(g)$. Sin embargo, si n es impar no puede existir $c > 0$ tal que $f(n) = 1 \ge cn^2 = cg(n)$, y por consiguiente $f \notin \Omega(g)$.

Intuitivamente, lo que buscamos es una función f cuyo crecimiento asintótico estuviera acotado superiormente por g (es decir, que f no creciera "más deprisa" que g) y que sin embargo f no estuviera acotado inferiormente por g.

c) Veamos que $\log_a n \in \Theta(\log_b n)$.

Sabemos por las propiedades de los logaritmos que si a y b son números reales mayores que 1 se cumple que

$$\log_b n = \frac{\log_a n}{\log_a b}.$$

Con esto,

$$\lim_{n\to\infty} \frac{\log_a n}{\log_b n} = \lim_{n\to\infty} \log_a b = \log_a b,$$

que es una constante real finita distinta de cero (pues $a,b>1$), y por tanto $\Theta(\log_a n) = \Theta(\log_b n)$.

Solución al Problema 1.8 (☺)

Para justificar estas afirmaciones nos apoyaremos en la definición de O y Ω, y trataremos de encontrar la constante real c y el número natural n_0 que caracterizan las inecuaciones que definen a ambas cotas.

- $f_1 \in O(f_2)$ pues $n^2 \le n^2 + 1000n$ para todo n (podemos tomar $c = 1$, $n_0 = 1$).
- $f_1 \notin O(f_3)$ pues si n es impar no existen c y n_0 tales que $n^2 \le cn$.
- $f_1 \in O(f_4)$ pues $n^2 \le n^3$ si $n > 100$ (podemos tomar $c = 1$, $n_0 = 101$).
- $f_2 \in O(f_1)$ pues basta tomar c y n_0 tales que $c > 1 + 1000/n$ para todo $n \ge n_0$ que sabemos que existen pues $(1000/n)$ tiende a cero.
- $f_2 \notin O(f_3)$ pues si n es impar no existen c y n_0 tales que $n^2 + 1000n \le cn$.
- $f_2 \in O(f_4)$ pues $n^2 + 1000n \le n^3$ si $n > 100$ (podemos tomar $c = 1$, $n_0 = 101$).
- $f_3 \notin O(f_1)$ pues si n es par no existen c y n_0 tales que $n^3 \le cn^2$.
- $f_3 \notin O(f_2)$ pues si n es par no existen c y n_0 tales que $n^3 \le c(n^2 + 1000n)$.
- $f_3 \in O(f_4)$ pues $f_3(n) \le n^3 = f_4(n)$ si $n > 100$ (podemos tomar $c = 1$, $n_0 = 101$).
- $f_4 \notin O(f_1)$ pues si $n > 100$ no existen c y n_0 tales que $n^3 \le cn^2$.
- $f_4 \notin O(f_2)$ pues si $n > 100$ no existen c y n_0 tales que $n^3 \le c(n^2 + 1000n)$.
- $f_4 \notin O(f_3)$ pues si $n > 100$, n impar, no existen c y n_0 tales que $n^3 \le cn$.
- $f_1 \in \Omega(f_2)$ pues $n^2 \ge c(n^2 + 1000n)$ para c y n_0 tales que $c > 1 + 1000/n$ para todo $n \ge n_0$ que sabemos que existen pues $(1000/n)$ tiende a cero.
- $f_1 \notin \Omega(f_3)$ pues si n es par no existen c y n_0 tales que $n^2 \ge cn^3$.

– $f_1 \notin \Omega(f_4)$ pues si $n > 100$ no existen c y n_0 tales que $n^2 \geq cn^3$.
– $f_2 \in \Omega(f_1)$ pues $n^2 + 100n \geq n^2$ para todo n (podemos tomar $c = 1$, $n_0 = 1$).
– $f_2 \notin \Omega(f_3)$ pues si n es par no existen c y n_0 tales que $n^2 + 1000n \geq cn^3$.
– $f_2 \notin \Omega(f_4)$ pues si $n > 100$ no existen c y n_0 tales que $n^2 + 1000n \geq cn^3$.
– $f_3 \notin \Omega(f_1)$ pues si n es impar no existen c y n_0 tales que $n \geq cn^2$.
– $f_3 \notin \Omega(f_2)$ pues si n es impar no existen c y n_0 tales que $n \geq c(n^2 + 1000n)$.
– $f_3 \notin \Omega(f_4)$ pues si n es impar no existen c y n_0 tales que $n \geq cn^3$.
– $f_4 \in \Omega(f_1)$ pues $n^3 \geq n^2$ si $n > 100$ (podemos tomar $c = 1$, $n_0 = 101$).
– $f_4 \in \Omega(f_2)$ pues $n^3 \geq n^2 + 1000n$ si $n > 100$ (podemos tomar $c = 1$, $n_0 = 101$).
– $f_4 \in \Omega(f_3)$ pues $n^3 \geq f_3(n)$ si $n > 100$ (podemos tomar $c = 1$, $n_0 = 101$).

Obsérvese que $\Theta(f_1) = \Theta(f_2) = \Theta(n^2)$, $\Theta(f_4) = \Theta(n^3)$, $\Omega(f_3) = \Omega(n)$, y $O(f_3) = O(n^3)$.

Solución al Problema 1.9 (☺/😐)

Para resolver estas ecuaciones seguiremos generalmente el mismo método, basado en los resultados expuestos al comienzo del capítulo. Primero intentaremos transformar la ecuación en recurrencia en una ecuación de la forma:

$$a_0 T(n) + a_1 T(n-1) + ... + a_k T(n-k) = b_1^n p_1(n) + b_2^n p_2(n) + ... + b_s^n p_s(n),$$

para después resolver su ecuación característica asociada. Con las raíces de esta ecuación es fácil ya obtener el término general de la función buscada.

a) $T(n) = 3T(n–1) + 4T(n–2)$ si $n>1$; $T(0) = 0$; $T(1) = 1$.

Escribiendo la ecuación de otra forma:

$$T(n) – 3T(n–1) – 4T(n–2) = 0,$$

ecuación en recurrencia homogénea con ecuación característica $x^2–3x–4 = 0$. Resolviendo esta ecuación, sus raíces son 4 y –1, con lo cual:

$$T(n) = c_1 4^n + c_2(–1)^n.$$

Para calcular las constantes necesitamos las condiciones iniciales:

$$\left.\begin{aligned} 0 = T(0) = c_1 4^0 + c_2(-1)^0 = c_1 + c_2 \\ 1 = T(1) = c_1 4^1 + c_2(-1)^1 = 4c_1 - c_2 \end{aligned}\right\} \Rightarrow \left.\begin{aligned} c_1 = 1/5 \\ c_2 = -1/5 \end{aligned}\right\}$$

Sustituyendo entonces en la ecuación anterior, obtenemos

$$T(n) = \frac{1}{5}\left(4^n - (-1)^n\right) \in \mathrm{O}(4^n).$$

b) $T(n) = 2T(n-1) - (n+5)3^n$ si $n>0$; $T(0) = 0$.

Reescribiendo la ecuación obtenemos:

$$T(n) - 2T(n-1) = -(n+5)3^n,$$

que es una ecuación en recurrencia no homogénea cuya ecuación característica asociada es $(x-2)(x-3)^2 = 0$, de raíces 2 y 3 (esta última con grado de multiplicidad dos), con lo cual

$$T(n) = c_1 2^n + c_2 3^n + c_3 n 3^n.$$

Para calcular las constantes necesitamos las condiciones iniciales. Como sólo disponemos de una y tenemos tres incógnitas, usamos la ecuación en recurrencia para obtener las otras dos:

$$T(1) = 2T(0) - 6\cdot 3 = -18$$
$$T(2) = 2T(1) - 7\cdot 9 = -99$$

Con esto:

$$\left.\begin{aligned} 0 &= T(0) = c_1 2^0 + c_2 3^0 + c_3 0\cdot 3^0 = c_1 + c_2 \\ -18 &= T(1) = c_1 2^1 + c_2 3^1 + c_3 1\cdot 3^1 = 2c_1 + 3c_2 + 3c_3 \\ -99 &= T(2) = c_1 2^2 + c_2 3^2 + c_3 2\cdot 3^2 = 4c_1 + 9c_2 + 18c_3 \end{aligned}\right\} \Rightarrow \left.\begin{aligned} c_1 &= 9 \\ c_2 &= -9 \\ c_3 &= -3 \end{aligned}\right\}$$

Sustituyendo estos valores en la ecuación anterior, obtenemos

$$T(n) = 9\cdot 2^n - 9\cdot 3^n - 3n3^n \in \Theta(n3^n).$$

Existe otra forma de resolver este tipo de problemas, que se basa en manipular la ecuación original hasta convertirla en homogénea. Partiendo de la ecuación $T(n) - 2T(n-1) = (n+5)3^n$, necesitamos escribir un sistema de ecuaciones basado en ella que permita anular el término no dependiente de $T(n)$. Para ello:

- primero escribimos la recurrencia original,
- la segunda ecuación se obtiene reemplazando n por $n-1$ y multiplicando por -6,
- y la tercera se obtiene reemplazando n por $n-2$ y multiplicando por 9.

De esta forma obtenemos:

$$\begin{aligned} T(n) - 2T(n-1) &= (n+5)3^n \\ -6T(n-1) + 12T(n-2) &= -6(n+4)3^{n-1} \\ 9T(n-2) - 18T(n-3) &= 9(n+3)3^{n-2} \end{aligned}$$

Sumando estas tres ecuaciones conseguimos una ecuación homogénea:

$$T(n) - 8T(n-1) + 21T(n-2) - 18T(n-3) = 0$$

cuya ecuación característica es $x^3 - 8x^2 + 21x - 18 = (x-2)(x-3)^2$. A partir de aquí se puede resolver mediante el proceso descrito anteriormente.

Como puede observarse, este método es más intuitivo pero menos metódico y ordenado que el que hemos utilizado para solucionar ecuaciones en recurrencia. Además, no hay una única forma de plantear esta ecuaciones.

c) $T(n) = 4T(n/2) + n^2$ si $n>4$, n potencia de 2; $T(1) = 1$; $T(2) = 8$.

Haciendo el cambio $n = 2^k$ (o, lo que es igual, $k = \log n$) obtenemos

$$T(2^k) = 4T(2^{k-1}) + 2^{2k}.$$

Llamando $t_k = T(2^k)$, la ecuación final es

$$t_k = 4t_{k-1} + 4^k,$$

ecuación no homogénea con ecuación característica $(x-4)^2 = 0$. Por tanto,

$$t_k = c_1 4^k + c_2 k 4^k.$$

Necesitamos ahora deshacer los cambios hechos. Primero $t_k = T(2^k)$, con lo que

$$T(2^k) = c_1 4^k + c_2 k 4^k = c_1 2^{2k} + c_2 k 2^{2k}$$

y después $n = 2^k$, obteniendo finalmente

$$T(n) = c_1 n^2 + c_2 n^2 \log n.$$

Para calcular las constantes necesitamos las condiciones iniciales:

$$\left.\begin{aligned} 1 = T(1) &= c_1 1^2 + c_2 1^2 \cdot 0 = c_1 \\ 8 = T(2) &= c_1 2^2 + c_2 2^2 \cdot 1 = 4c_1 + 4c_2 \end{aligned}\right\} \Rightarrow \left.\begin{aligned} c_1 &= 1 \\ c_2 &= 1 \end{aligned}\right\}$$

Sustituyendo estos valores en la ecuación anterior, obtenemos

$$T(n) = n^2 + n^2 \log n \in \Theta(n^2 \log n).$$

Existe otra forma de resolver este tipo de problemas, mediante el desarrollo "telescópico" de la ecuación en recurrencia. Escribiremos la ecuación hasta llegar a una expresión en donde sólo aparezcan las condiciones iniciales:

$$T(n) = 4T\left(\frac{n}{2}\right) + n^2 = 4\left(4T\left(\frac{n}{4}\right) + \left(\frac{n}{2}\right)^2\right) + n^2 =$$

$$= 4^2 T\left(\frac{n}{4}\right) + 2n^2 = 4^2\left(4T\left(\frac{n}{8}\right) + \left(\frac{n}{4}\right)^2\right) + 2n^2 =$$

$$= 4^3 T\left(\frac{n}{8}\right) + 3n^2 = \ldots\ldots\ldots = 4^x T(1) + xn^2.$$

De esta forma hemos ido desarrollando los términos de esta sucesión, cada uno en función de términos anteriores. Sólo nos queda por calcular el número de términos (x) que hemos tenido que desarrollar.

Pero ese número x coincide con el número de términos de la sucesión $n/2$, $n/4$, $n/8$,...,4,2,1, que es log$n$ pues n es una potencia de 2. En consecuencia,

$$T(n) = 4^{\log n} T(1) + \log n \cdot n^2 = n^{\log 4} \cdot 1 + \log n \cdot n^2 = n^2 + \log n \cdot n^2 \in \Theta(n^2 \log n).$$

d) $T(n) = 2T(n/2) + n\log n$ si $n>1$, n potencia de 2.

Haciendo el cambio $n = 2^k$ (o, lo que es igual, $k = \log n$) obtenemos

$$T(2^k) = 2T(2^{k-1}) + k2^k.$$

Llamando $t_k = T(2^k)$, la ecuación final es

$$t_k = 2t_{k-1} + k2^k,$$

ecuación en recurrencia no homogénea con ecuación característica $(x-2)^3 = 0$. Por tanto,

$$t_k = c_1 2^k + c_2 k2^k + c_3 k^2 2^k.$$

Necesitamos ahora deshacer los cambios hechos. Primero $t_k = T(2^k)$, con lo que

$$T(2^k) = c_1 2^k + c_2 k2^k + c_3 k^2 2^k,$$

y después $n = 2^k$ ($k = \log n$), por lo cual

$$T(n) = c_1 n + c_2 n\log n + c_3 n\log^2 n.$$

De esta ecuación no conocemos condiciones iniciales para calcular todas las constantes, pero sí es posible intentar fijar alguna de ellas. Para eso, basta sustituir la expresión que hemos encontrado para $T(n)$ en la ecuación original:

$$n\log n = T(n) - 2T(n/2) = (c_3 - c_2)n + 2c_3 n\log n,$$

por lo que $c_3 = c_2$ y $2c_3 = 1$, de donde

$$T(n) = c_1 n + 1/2 n\log n + 1/2 n\log^2 n.$$

En consecuencia $T(n) \in \Theta(n\log^2 n)$ independientemente de las condiciones iniciales.

e) $T(n) = 3T(n/2) + 5n + 3$ si $n>1$, n potencia de 2.

Haciendo el cambio $n = 2^k$ (o, lo que es igual, $k = \log n$) obtenemos

$$T(2^k) = 3T(2^{k-1}) + 5 \cdot 2^k + 3.$$

Llamando $t_k = T(2^k)$, la ecuación final es:

$$t_k = 3t_{k-1} + 5\cdot 2^k + 3,$$

ecuación en recurrencia no homogénea cuya ecuación característica asociada es $(x-3)(x-2)(x-1) = 0$. Por tanto,

$$t_k = c_1 3^k + c_2 2^k + c_3.$$

Necesitamos ahora deshacer los cambios hechos. Primero $t_k = T(2^k)$, con lo que

$$T(2^k) = c_1 3^k + c_2 2^k + c_3$$

y después $n = 2^k$ ($k = \log n$), por lo cual

$$T(n) = c_1 3^{\log n} + c_2 n + c_3 = c_1 n^{\log 3} + c_2 n + c_3.$$

De esta ecuación no conocemos condiciones iniciales para calcular todas las constantes, pero sí es posible intentar fijar alguna de ellas. Para eso basta sustituir la expresión que hemos encontrado para $T(n)$ en la ecuación en recurrencia original, y obtenemos:

$$c_1 n^{\log 3} + c_2 n + c_3 = 3(c_1(n^{\log 3}/3) + c_2 n/2 + c_3) + 5n + 3.$$

Igualando los coeficientes de $n^{\log 3}$, n y los términos independientes obtenemos $c_3 = -3/2$ y $c_2 = -10$, de donde

$$T(n) = c_1 n^{\log 3} - 10n - 3/2.$$

Como $\log 3 > 1$, $T(n)$ será de complejidad $\Theta(n^{\log 3})$ si c_1 es distinto de cero, o bien $T(n) \in \Theta(n)$ si $c_1 = 0$.

Para ver cuándo c_1 vale cero estudiaremos los valores de las condiciones iniciales que le hacen tomar ese valor, en este caso $T(1)$. Por un lado, utilizando la ecuación original, tenemos que para $n = 2$:

$$T(2) = 3T(1) + 10 + 3.$$

Por otro lado, basándonos en la ecuación que hemos obtenido,

$$T(2) = 3c_1 - 20 - 3/2.$$

Igualando ambas ecuaciones, obtenemos que $c_1 = T(1) + 23/2$. Por tanto,

$$T(n) \in \begin{cases} \Theta(n^{\log 3}) & \text{si } T(1) \neq -23/2 \\ \Theta(n) & \text{si } T(1) = -23/2 \end{cases}$$

f) $T(n) = 2T(n/2) + \log n$ si $n>1$, n potencia de 2.

Haciendo el cambio $n = 2^k$ (o, lo que es igual, $k = \log n$) obtenemos

$$T(2^k) = 2T(2^{k-1}) + k.$$

Llamando $t_k = T(2^k)$, la ecuación final es

$$t_k = 2t_{k-1} + k,$$

ecuación en recurrencia no homogénea que puede ser expresada como

$$t_k - 2t_{k-1} = k$$

y cuya ecuación característica asociada es $(x-2)(x-1)^2 = 0$. Por tanto,

$$t_k = c_1 2^k + c_2 + c_3 k.$$

Necesitamos ahora deshacer los cambios hechos. Primero $t_k = T(2^k)$, con lo que

$$T(2^k) = c_1 2^k + c_2 + c_3 k$$

y después $n = 2^k$ ($k = \log n$), y por tanto

$$T(n) = c_1 n + c_2 + c_3 \log n.$$

De esta ecuación no conocemos condiciones iniciales para calcular todas las constantes, pero sí es posible intentar fijar alguna de ellas. Para eso, basta sustituir la expresión que hemos encontrado para $T(n)$ en la ecuación en recurrencia original, y obtenemos:

$$c_1 n + c_2 + c_3 \log n = 2(c_1 n/2 + c_2 + c_3 \log n - c_3) + \log n.$$

Igualando los coeficientes de $\log n$ y los términos independientes obtenemos que $c_3 = -1$ y $c_2 = -2$, de donde

$$T(n) = c_1 n - 2 - \log n.$$

Esta función será de orden de complejidad $\Theta(n)$ si c_1 es distinto de cero, o bien $T(n) \in \Theta(\log n)$ si $c_1 = 0$.

Para ver cuándo c_1 vale cero estudiaremos los valores de las condiciones iniciales que le hacen tomar ese valor, en este caso $T(1)$. Por un lado, utilizando la ecuación original, tenemos que para $n = 2$:

$$T(2) = 2T(1) + 1.$$

Por otro lado, basándonos en la ecuación que hemos obtenido

$$T(2) = 2c_1 - 2 - 1.$$

Igualando ambas ecuaciones, obtenemos que $c_1 = \mathrm{T}(1) + 2$. Por tanto,

$$T(n) \in \begin{cases} \Theta(\log n) & \text{si } T(1) = -2 \\ \Theta(n) & \text{si } T(1) \neq -2 \end{cases}$$

g) $T(n) = 2T(n^{1/2}) + \log n$ con $n = 2^{2^k}$; $T(2) = 1$.

Haciendo el cambio $n = 2^{2^k}$ ($k = \log\log n$) obtenemos la ecuación

$$T(2^{2^k}) = 2T(2^{2^{k-1}}) + \log 2^{2^k}.$$

Llamando $t_k = T(2^{2^k})$, la ecuación final es

$$t_k = 2t_{k-1} + 2^k,$$

ecuación en recurrencia no homogénea cuya ecuación característica es $(x-2)^2 = 0$. Por tanto,

$$t_k = c_1 2^k + c_2 k 2^k.$$

Necesitamos ahora deshacer los cambios hechos. Primero $t_k = T(2^{2^k})$, con lo que

$$T(2^{2^k}) = c_1 2^k + c_2 k 2^k$$

y después $n = 2^{2^k}$ ($k = \log\log n$, o bien $\log n = 2^k$), por lo cual tenemos que

$$T(n) = c_1 \log n + c_2 \log n \cdot \log\log n.$$

Para calcular las constantes necesitamos las condiciones iniciales. Como disponemos de sólo una y tenemos dos incógnitas, usamos la ecuación original para obtener la otra:

$$T(4) = 2T(2) + \log 4 = 4.$$

Con esto:

$$\left.\begin{aligned} 1 = T(2) = c_1 \log 2 + c_2 \log 2 \cdot 0 = c_1 \\ 4 = T(4) = c_1 \log 4 + c_2 \log 4 \cdot \log\log 4 = 2c_1 + 2c_2 \end{aligned}\right\} \Rightarrow \left.\begin{aligned} c_1 = 1 \\ c_2 = 1 \end{aligned}\right\}$$

Sustituyendo estos valores en la ecuación anterior, obtenemos

$$T(n) = \log n + \log n \cdot \log\log n \in \Theta(\log n \cdot \log\log n).$$

h) $T(n) = 5T(n/2) + (n\log n)^2$ si $n>1$, n potencia de 2; $T(1)=1$.

Haciendo el cambio $n = 2^k$ (o, lo que es igual, $k = \log n$) obtenemos

$$T(2^k) = 5T(2^{k-1}) + (k2^k)^2 = 5T(2^{k-1}) + k^2 4^k.$$

Llamando $t_k = T(2^k)$, la ecuación final es

$$t_k = 5t_{k-1} + k^2 4^k,$$

ecuación en recurrencia no homogénea que puede ser expresada como

$$t_k - 5\, t_{k-1} = k^2 4^k,$$

cuya ecuación característica asociada es $(x-5)(x-4)^3 = 0$. Por tanto,

$$t_k = c_1 5^k + c_2 4^k + c_3 k 4^k + c_4 k^2 4^k.$$

Necesitamos ahora deshacer los cambios hechos. Primero $t_k = T(2^k)$, con lo que

$$T(2^k) = c_1 5^k + c_2 4^k + c_3 k 4^k + c_4 k^2 4^k$$

y después $n = 2^k$ ($k = \log n$), por tanto

$$\begin{aligned} T(n) &= c_1 5^{\log n} + c_2 4^{\log n} + c_3 \log n 4^{\log n} + c_4 \log^2 n 4^{\log n} = \\ &c_1 n^{\log 5} + c_2 n^{\log 4} + c_3 \log n \cdot n^{\log 4} + c_4 \log^2 n \cdot n^{\log 4} = \\ &c_1 n^{\log 5} + c_2 n^2 + c_3 n^2 \log n + c_4 n^2 \log^2 n. \end{aligned}$$

Para calcular las constantes necesitamos las condiciones iniciales. Como sólo disponemos de una y tenemos tres incógnitas, usamos la ecuación en recurrencia para obtener las otras dos:

$$\begin{aligned} T(2) &= 5T(1) + 2^2 = 9; \\ T(4) &= 5T(2) + 8^2 = 109; \\ T(8) &= 5T(4) + 24^2 = 1121. \end{aligned}$$

Con esto:

$$\left.\begin{aligned} 1 &= T(1) = c_1 1 + c_2 1 + c_3 1 \cdot 0 + c_4 1 \cdot 0 = c_1 + c_2 \\ 9 &= T(2) = c_1 5 + c_2 4 + c_3 4 \cdot 1 + c_4 4 \cdot 1 = 5c_1 + 4c_2 + 4c_3 + 4c_4 \\ 109 &= T(4) = c_1 25 + c_2 16 + c_3 16 \cdot 2 + c_4 16 \cdot 4 = 25c_1 + 16c_2 + 32c_3 + 64c_4 \\ 1121 &= T(8) = c_1 125 + c_2 64 + c_3 64 \cdot 3 + c_4 64 \cdot 9 = 125c_1 + 64c_2 + 192c_3 + 576c_4 \end{aligned}\right\}$$

Solucionando el sistema de ecuaciones obtenemos los valores de las constantes:

$$\left.\begin{aligned} c_1 &= 181 \\ c_2 &= -180 \\ c_3 &= -40 \\ c_4 &= -4 \end{aligned}\right\}$$

y sustituyéndolos en la ecuación anterior, obtenemos

$$T(n) = 181 n^{\log 5} - 180 n^2 - 40 n^2 \log n - 4 n^2 \log^2 n \in \Theta(n^2 \log^2 n).$$

i) $T(n) = T(n-1) + 2T(n-2) - 2T(n-3)$ si $n>2$; $T(n) = 9n^2 - 15n + 106$ si $n=0,1,2$.

Reescribiendo la ecuación:

$$T(n) - T(n-1) - 2T(n-2) + 2T(n-3) = 0,$$

ecuación en recurrencia homogénea cuya ecuación característica asociada es

$$x^3 - x^2 - 2x + 2 = 0.$$

Resolviendo esa ecuación, sus raíces son 1, $\sqrt{2}$ y $-\sqrt{2}$, con lo cual

$$T(n) = c_1 + c_2(\sqrt{2})^n + c_3(-\sqrt{2})^n.$$

Para calcular las constantes necesitamos las condiciones iniciales:

$$\left.\begin{aligned} 106 &= T(0) = c_1 + c_2 + c_3 \\ 100 &= T(1) = c_1 + c_2\sqrt{2} - c_3\sqrt{2} \\ 112 &= T(2) = c_1 + 2c_2 + 2c_3 \end{aligned}\right\} \Rightarrow \left.\begin{aligned} c_1 &= 100 \\ c_2 &= 3 \\ c_3 &= 3 \end{aligned}\right\} \Rightarrow T(n) = 100 + 3\sqrt{2^n}\left(1 + (-1)^n\right)$$

.

En consecuencia, $T(n) \in \Theta(2^{n/2})$.

j) $T(n) = (3/2)T(n/2) - (1/2)T(n/4) - (1/n)$ si $n>2$, n potencia de 2; $T(1)=1$; $T(2)=3/2$.

Haciendo el cambio $n = 2^k$ (o, lo que es igual, $k = \log n$) obtenemos

$$T(2^k) = (3/2)T(2^{k-1}) - (1/2)T(2^{k-2}) - (1/2)^k.$$

Llamando $t_k = T(2^k)$, la ecuación final es

$$t_k = (3/2)t_{k-1} - (1/2)t_{k-2} - (1/2)^k$$

ecuación en recurrencia no homogénea en la forma

$$t_k - (3/2)t_{k-1} + (1/2)t_{k-2} = -(1/2)^k,$$

cuya ecuación característica asociada es $(x-1)(x-1/2)^2 = 0$. Por tanto,

$$t_k = c_1 + c_2 2^{-k} + c_3 k 2^{-k}.$$

Necesitamos ahora deshacer los cambios hechos. Primero $t_k = T(2^k)$, con lo que

$$T(2^k) = c_1 + c_2 2^{-k} + c_3 k 2^{-k}$$

después $n = 2^k$ ($k = \log n$), con lo cual

$$T(n) = c_1 + (1/n)c_2 + c_3(\log n/n).$$

Para calcular las constantes necesitamos las condiciones iniciales. Como sólo disponemos de dos y tenemos tres incógnitas, usamos la ecuación en recurrencia para obtener la tercera:

$$T(4) = (3/2)T(2) - (1/2)T(1) - (1/4) = 3/2.$$

Con esto:

$$\left.\begin{aligned} 1 &= T(1) = c_1 + c_2 \\ 3/2 &= T(2) = c_1 + c_2(1/2) + c_3(1/2) \\ 3/2 &= T(4) = c_1 + c_2(1/4) + c_3(1/2) \end{aligned}\right\} \Rightarrow \left.\begin{aligned} c_1 &= 1 \\ c_2 &= 0 \\ c_3 &= 1 \end{aligned}\right\} \Rightarrow T(n) = 1 + \frac{\log n}{n} \in \Theta(1).$$

k) $T(n) = 2T(n/4) + n^{1/2}$ si $n>4$, n potencia de 4.

Haciendo el cambio $n = 2^k$ (o, lo que es igual, $k = \log n$) obtenemos

$$T(2^k) = 2T(2^{k-2}) + 2^{k/2}.$$

Llamando $t_k = T(2^k)$, la ecuación final es

$$t_k = 2t_{k-2} + 2^{k/2},$$

ecuación en recurrencia no homogénea de la forma

$$t_k - 2t_{k-2} = (\sqrt{2})^k$$

cuya ecuación característica asociada es $(x^2-2)(x-\sqrt{2}) = 0$, o lo que es igual, $(x+\sqrt{2})(x-\sqrt{2})^2 = 0$. Por tanto,

$$t_k = c_1(-\sqrt{2})^k + c_2(\sqrt{2})^k + c_3k(\sqrt{2})^k.$$

Necesitamos ahora deshacer los cambios hechos. Primero $t_k = T(2^k)$, con lo que

$$T(2^k) = c_1(-\sqrt{2})^k + c_2(\sqrt{2})^k + c_3k(\sqrt{2})^k$$

después $n = 2^k$ ($k = \log n$), y por tanto obtenemos:

$$T(n) = \sqrt{n}\ (c_1(-1)^{\log n} + c_2 + c_3\log n).$$

Si n es múltiplo de 4 entonces $\log n$ es par, y por tanto $(-1)^{\log n}$ vale siempre 1. Esto nos permite afirmar, llamando $c_0 = c_1 + c_2$, que

$$T(n) = \sqrt{n}\ (c_0 + c_3\log n).$$

De esta ecuación no conocemos condiciones iniciales para calcular todas las constantes, pero sí es posible intentar fijar alguna de ellas. Para eso, basta sustituir la expresión que hemos encontrado para $T(n)$ en la ecuación original:

$$\sqrt{n}\ (c_0 + c_3\log n) = 2(\sqrt{n}/2\ (c_0 + c_3\log n - 2c_3)) + \sqrt{n}\ .$$

Igualando los coeficientes de $\sqrt{n}$, $\sqrt{n}\log n$ y los términos independientes obtenemos $c_3=1/2$, de donde

$$T(n) = \sqrt{n}\ (c_0 + 1/2\log n) \in \Theta(\sqrt{n}\log n).$$

Este problema también podría haberse solucionado mediante otro cambio, $n = 4^k$ (o, lo que es igual, $k = \log_4 n$) obteniendo la ecuación

$$T(4^k) = 2T(4^{k-1}) + 4^{k/2}.$$

Esta nos lleva, tras llamar $t_k = T(4^k)$, a la ecuación final

$$t_k = 2t_{k-1} + 2^k,$$

cuya resolución conduce a la misma solución que la obtenida mediante el primer cambio.

l) $T(n) = 4T(n/3) + n^2$ si $n>3$, n potencia de 3.

Haciendo el cambio $n = 3^k$ (o, lo que es igual, $k = \log_3 n$) obtenemos que

$$T(3^k) = 4T(3^{k-1}) + 9^k.$$

Llamando $t_k = T(3^k)$, la ecuación final es

$$t_k = 4t_{k-1} + 9^k,$$

ecuación en recurrencia no homogénea de la forma

$$t_k - 4t_{k-1} = 9^k,$$

cuya ecuación característica asociada es $(x-4)(x-9) = 0$. Por tanto,

$$t_k = c_1 4^k + c_2 9^k.$$

Necesitamos ahora deshacer los cambios hechos. Primero $t_k = T(3^k)$, con lo que

$$T(3^k) = c_1 4^k + c_2 3^{2k}$$

y después $n = 3^k$ ($k = \log_3 n$), y por tanto

$$T(n) = c_1 4^{\log_3 n} + c_2 n^2 = c_1 n^{\log_3 4} + c_2 n^2.$$

De esta ecuación no conocemos condiciones iniciales para calcular todas las constantes, pero sí es posible intentar fijar alguna de ellas. Para eso, basta sustituir la expresión que hemos encontrado para $T(n)$ en la ecuación en recurrencia original, y obtenemos:

$$c_1 n^{\log_3 4} + c_2 n^2 = 4\left(c_1 \left(\frac{n}{3}\right)^{\log_3 4} + c_2 \left(\frac{n}{3}\right)^2 \right) + n^2 = c_1 n^{\log_3 4} + \left(\frac{4}{9} c_2 + 1\right) n^2$$

Igualando los coeficientes de $n^{\log_3 4}$ y de n^2 obtenemos $c_2 = 9/5$, de donde

$$T(n) = c_1 n^{\log_3 4} + \frac{9}{5} n^2.$$

Como $\log_3 4 < 2$, entonces $T(n) \in \Theta(n^2)$.

Solución al Problema 1.10 (😐)

a) Cierto. Se deduce de la propiedad 6 del apartado 1.3.1, pero veamos una posible demostración directa:

Si $T_1 \in O(f)$, sabemos que existen $c_1 > 0$ y n_1 tales que $T_1(n) \leq c_1 f(n)$ para todo $n \geq n_1$. Análogamente, como $T_2 \in O(f)$, existen $c_2 > 0$ y n_2 tales que $T_2(n) \leq c_2 f(n)$ para $n \geq n_2$. [1.1]

Para comprobar que $T_1 + T_2 \in O(f)$, debemos encontrar una constante real $c > 0$ y un número natural n_0 tales que $T_1(n) + T_2(n) \leq cf(n)$ para todo $n \geq n_0$. [1.2]

Apoyándonos en [1.1], basta tomar $n_0 = máx\{n_1, n_2\}$ y $c = c_1 + c_2$, con las que se verifica la ecuación [1.2] para todo $n \geq n_0$.

Existe otra forma de demostrarlo, utilizando límites en caso de que estos existan, como sucede por ejemplo cuando las funciones son continuas:

Si $T_1 \in O(f)$, entonces $\lim_{n\to\infty} \frac{T_1(n)}{f(n)} = k_1 < \infty$.

Análogamente, como $T_2 \in O(f)$, $\lim_{n\to\infty} \frac{T_2(n)}{f(n)} = k_2 < \infty$. [1.3]

Veamos entonces que $\lim_{n\to\infty} \frac{T_1(n) + T_2(n)}{f(n)} = k < \infty$. [1.4]

Pero [1.4] es cierto pues, como los dos límites en [1.3] son finitos y positivos podemos conmutar la suma con el límite y obtenemos que

$$\lim_{n\to\infty} \frac{T_1(n) + T_2(n)}{f(n)} = \lim_{n\to\infty} \frac{T_1(n)}{f(n)} + \lim_{n\to\infty} \frac{T_2(n)}{f(n)} = k_1 + k_2 < \infty.$$

b) Cierto.

Análogamente a lo realizado en el apartado anterior, si $T_1 \in O(f)$, entonces $\lim_{n\to\infty} \frac{T_1(n)}{f(n)} = k_1 < \infty$. Igualmente, como $T_2 \in O(f)$, $\lim_{n\to\infty} \frac{T_2(n)}{f(n)} = k_2 < \infty$. [1.5]

Veamos entonces que $\lim_{n\to\infty} \frac{T_1(n) - T_2(n)}{f(n)} = k < \infty$. [1.6]

Pero [1.6] es cierto pues, como los dos límites en [1.5] existen y son finitos y positivos podemos conmutar la resta con el límite y obtenemos que

$$\lim_{n\to\infty} \frac{T_1(n) - T_2(n)}{f(n)} = \lim_{n\to\infty} \frac{T_1(n)}{f(n)} - \lim_{n\to\infty} \frac{T_2(n)}{f(n)} = k_1 - k_2 < \infty.$$

c) Falso.

Consideremos $T_1(n) = n^2$, $T_2(n) = n$, y $f(n) = n^3$. Tenemos por tanto que $T_1 \in O(f)$ y $T_2 \in O(f)$, pero sin embargo $T_1(n)/T_2(n) = n \notin O(1)$.

d) Falso.

Consideremos de nuevo $T_1(n) = n^2$, $T_2(n) = n$, y $f(n) = n^3$. Tenemos por tanto que $T_1 \in O(f)$ y $T_2 \in O(f)$, pero sin embargo $T_1 \notin O(T_2)$ pues $n^2 \notin O(n)$.

Solución al Problema 1.11 (☺)

Sean $f(n) = n$ y $g(n) = \begin{cases} n^2, & \text{si } n \text{ es par.} \\ 1, & \text{si } n \text{ es impar.} \end{cases}$

Si n es impar, no podemos encontrar ninguna constante c tal que

$$f(n) = n \leq cg(n) = c,$$

y por tanto $f \notin O(g)$. Por otro lado, si n es par no podemos encontrar ninguna constante c tal que

$$g(n) = n^2 \leq cf(n) = cn,$$

y por tanto $g \notin O(f)$.

Solución al Problema 1.12 (☺)

Para comprobar que $\log^k n \in O(n)$ basta ver que

$$\lim_{n\to\infty} \frac{\log^k n}{n} = 0$$

para todo k. Pero eso es cierto siempre. Obsérvese además que por esa misma razón $\log^k n \notin \Omega(n)$ para cualquier $k > 0$.

Solución al Problema 1.13

Vamos a suponer que los tiempos de ejecución de las funciones *Esvacio*, *Izq*, *Der* y *Raiz* es de c operaciones elementales (OE), que el tiempo de ejecución de *Opera* es d OE, y el de *Max2* es 1 OE.

Procedimiento *Inorden* (☺)

Para calcular el tiempo de ejecución, calcularemos primero el número de operaciones elementales (OE) que se realizan:

- En la línea (1) se ejecutan 2+c OE: la llamada a *Esvacio* (1 OE), el tiempo de ejecución de este procedimiento (c) y una negación.
- En la línea (2) se efectúa la llamada a *Izq* (1 OE), lo que tarda ésta en ejecutarse (c OE) más la llamada a *Inorden* (1 OE) y lo que tarde ésta en ejecutarse, que va a depender del número de elementos del árbol *Izq(t)*.
- En la línea (3) se ejecutan 2+c+d OE: dos llamadas a procedimientos y sus respectivos tiempos de ejecución.
- El número de OE de la línea (4) se calcula de forma análoga a la línea (2): 2+c más lo que tarda *Inorden* en ejecutarse con el número de elementos que hay en *Der(t)*.

Para estudiar el tiempo de ejecución, vamos a considerar dos casos extremos: que el árbol sea degenerado (es decir, una lista) y que sea equilibrado. Cualquier árbol se encuentra en una situación intermedia a estos dos casos.

- Si t es degenerado, podemos suponer sin pérdida de generalidad que *Esvacio(Izq(t))* y que para todo a subárbol de t se verifica que *Esvacio(Izq(a))*. Por tanto, el número de OE que se realizan en la ejecución de *Inorden(t)* para un árbol t con n elementos es:

$$T(n) = (2+c) + (2+c+T(0)) + (2+c+d) + (2+c+T(n-1)) = 8+4c+d+T(0)+T(n-1).$$
$$T(0) = 2+c.$$

  Con esto, $T(n) = 10 + 5c + d + T(n-1)$, ecuación en recurrencia no homogénea que podemos resolver desarrollándola telescópicamente:

$$T(n) = 10 + 5c + d + T(n-1) = (10 + 5c + d) + (10 + 5c + d) + T(n-2) = \ldots = (10 + 5c + d)n + (2+c) \in \Theta(n)$$

- Si t es equilibrado sus dos subárboles (izquierdo y derecho) tienen del orden de $n/2$ elementos y son a su vez equilibrados. Por tanto, el número de OE que se realizan en la ejecución de *Inorden(t)* para un árbol t con n elementos es:

$$T(n)= (2+c) + (2+c+T(n/2)) + (2+c+d) + (2+c+T(n/2)) = 8+4c+d+2T(n/2).$$
$$T(0)= 2+c.$$

  Para resolver esta ecuación en recurrencia se hace el cambio $t_k = T(2^k)$, con lo que obtenemos

$$t_k - 2t_{k-1} = 8 + 4c + d,$$

  ecuación no homogénea con ecuación característica $(x-2)(x-1) = 0$. Por tanto,

$$t_k = c_1 2^k + c_2$$

  y, deshaciendo los cambios,

$$T(n) = c_1 n + c_2.$$

Para calcular las constantes, nos apoyamos en la condición inicial $T(0)=2+c$, junto con el valor de $T(1)$, que puede ser calculado basándonos en la expresión de la ecuación en recurrencia: $T(1) = 8 + 4c + d + 2(2 + c)$, obteniendo

$$T(n) = (10 + 5c + d)n + (2+c) \in \Theta(n).$$

Función *Altura* (😐)

Para determinar el tiempo de ejecución, calcularemos primero el número de operaciones elementales (OE) que se realizan:

– En la línea (1) se ejecutan $1+c$ OE: la llamada a *Esvacio* (1 OE) más el tiempo de ejecución de este procedimiento (c OE).
– En la línea (2) se realiza 1 OE.
– En la línea (4) se efectúan:
  a) la llamada a *Izq* (1 OE), lo que tarda ésta en ejecutarse (c OE) más la llamada a *Altura* (1 OE) y lo que tarde ésta en ejecutarse, que va a depender del número de elementos del árbol *Izq(t)*; más
  b) la llamada a *Der* (1 OE), lo que tarda ésta en ejecutarse (c OE) más la llamada a *Altura* (1 OE) y lo que tarde ésta en ejecutarse, que va a depender del número de elementos del árbol *Der(t)*; más
  c) el cálculo del máximo de ambos números (1 OE), un incremento (1 OE) y el *RETURN* (1 OE).

Para estudiar el tiempo de ejecución de esta función consideraremos los mismos casos que para la función *Inorden*: que el árbol sea degenerado (es decir, una lista) o que sea equilibrado.

- Si t es degenerado, podemos suponer sin pérdida de generalidad que *Esvacio(Izq(t))* y que para todo a subárbol de t se verifica que *Esvacio(Izq(a))*. Por tanto, el número de OE que se realizan en la ejecución de *Altura(t)* para un árbol t con n elementos es:

  $T(n)= (1+c) + (1+c+1+T(0) + 1+c+1+T(n–1)) +3 = 8 + 3c + T(0) + T(n–1).$
  $T(0)= (1+c) + 1 = 2+c.$

  Con esto, $T(n)=10+4c+T(n–1)$, ecuación en recurrencia no homogénea que podemos resolver desarrollándola telescópicamente:

  $$T(n) = 10 + 4c + T(n–1) = (10 + 4c) + (10 + 4c) + T(n–2) = ... =$$
  $$(10 + 4c)n + (2 + c) \in \Theta(n)$$

- Si t es equilibrado sus dos subárboles tienen del orden de $n/2$ elementos y son también equilibrados. Por tanto, el número de OE que se realizan en la ejecución de *Altura(t)* para un árbol t con n elementos es:

  $T(n)= (1+c) + (1+c+1+T(n/2)) + 1+c+1+T(n/2)) +3 = 8 + 3c + 2T(n/2).$
  $T(0)= 2+c.$

Para resolver esta ecuación en recurrencia se hace el cambio $t_k=T(2^k)$, con lo que obtenemos

$$t_k - 2t_{k-1} = 8 + 3c,$$

ecuación no homogénea de ecuación característica $(x-2)(x-1) = 0$. Por tanto,

$$t_k = c_1 2^k + c_2.$$

Deshaciendo los cambios,

$$T(n) = c_1 n + c_2.$$

Para calcular las constantes, nos apoyamos en la condición inicial $T(0)=2+c$, junto con el valor de $T(1)$, que puede ser calculado basándonos en la expresión de la ecuación en recurrencia: $T(1) = 8 + 3c + 2(2 + c)$. Finalmente obtenemos

$$T(n) = (10 + 4c)n + (2 + c) \in \Theta(n).$$

Función *Mezcla* (✍)

Para resolver este problema vamos a suponer que el tiempo de ejecución del procedimiento *Ins*, que inserta un elemento en un árbol binario de búsqueda, es $A\log n+B$, siendo A y B dos constantes. Supongamos también que n y m son el número de elementos de *t1* y *t2* respectivamente.

Para estudiar el tiempo de ejecución $T(n,m)$ consideraremos, al igual que hicimos para la función anterior, dos casos extremos: que el árbol *t2* sea degenerado (es decir, una lista) o que sea equilibrado.

- Si *t2* es degenerado, podemos suponer sin pérdida de generalidad que *Esvacio(Izq(t2))* y que para todo *a* subárbol de *t2* se verifica que *Esvacio(Izq(a))*. Por tanto, vamos a ver el número de OE que se realizan en cada línea de la función en este caso*:*
  - En la línea (1) se invoca a *Esvacio(t1)*, lo que supone $1+c$ OE.
  - En la línea (2) se efectúa 1 OE.
  - Análogamente, las líneas (3) y (4) realizan $(1+c)$ y 1 respectivamente.
  - Para estudiar el número de OE que realiza la línea (6), vamos a dividirla en cuatro partes:
    a) *a1:=Ins(t1,Raiz(t2))*, siendo *a1* una variable auxiliar para efectuar los cálculos. Se efectúan $2+c+A\log n+B$ operaciones elementales: la llamada a *Raiz* (1), el tiempo que ésta tarda (c), la llamada a *Ins* (1 OE), y su tiempo de ejecución ($A\log n+B$).
    b) *a2:=Mezcla(a1,Izq(t2))*, siendo *a2* una variable auxiliar para efectuar los cálculos. Se efectúan aquí $2+c+T(n+1,0)$ operaciones elementales: llamada a I*zq* (1), el tiempo que ésta tarda (c), la llamada a M*ezcla* (1 OE), y su tiempo de ejecución, que será $T(n+1,0)$, pues estamos suponiendo que *Esvacio(Izq(a))* para todo *a* subárbol de *t2*.
    c) *a3:=Mezcla(a2,Der(t2))*, siendo *a3* una variable auxiliar para efectuar los cálculos. Se efectúan $2+c+T(n+1,m-1)$ operaciones elementales: la

llamada a *Der* (1), el tiempo que ésta tarda (c), la llamada a *Mezcla* (1 OE), y su tiempo de ejecución, que será $T(n+1,m-1)$, pues estamos suponiendo que *Esvacio(Izq(a))* para todo a subárbol de *t2* o, lo que es igual, que el número de elementos de *Der(t)* es $m-1$.

d) *RETURN a3*, que realiza 1 OE.

Por tanto, la ejecución de *Mezcla(t1,t2)* en este caso es :

$$T(n,m) = 9 + 5c + B + A\log n + T(n+1,0) + T(n+1,m-1)$$

con las condiciones iniciales $T(0,m) = 2 + c$ y $T(n,0) = 3 + 2c$. Para resolver la ecuación en recurrencia podemos expresarla como:

$$T(n,m) = 12 + 7c + B + A\log n + T(n+1,m-1)$$

haciendo uso de la segunda condición inicial. Desarrollando telescópicamente la ecuación:

$$\begin{aligned} T(n,m) &= 12 + 7c + B + A\log n + T(n+1,m-1) = \\ &= (12+7c+B+A\log n) + (12+7c+B+A\log(n+1)) + T(n+2,m-2) = \\ &\quad \text{.....} \\ &= m(12+7c+B) + \left(\sum_{i=0}^{m-1} A\log(n+i)\right) + T(n+m,0) = \\ &= m(12+7c+B) + 2c + 3 + A\left(\sum_{i=0}^{m-1} \log(n+i)\right). \end{aligned}$$

Pero como $\log(n+i) \le \log(n+m)$ para todo $0 \le i \le m$,

$$T(n,m) \le m(12 + 7c + B) + 2c + 3 + Am\log(n+m) \in O(m\log(n+m))$$

- El segundo caso es que *t2* sea equilibrado, para el que se demuestra de forma análoga que

$$T(n,m) \in O(m\log(n+m)).$$

Solución al Problema 1.14 (☺)

Para comprobar que $O(f) \subset O(g)$, basta ver que $lim_{n\to\infty}(f(n)/g(n)) = 0$ en cada caso pues las funciones son continuas, lo que implica la existencia de los límites. De esta forma se obtiene la siguiente ordenación:

$$\begin{aligned} O((1/3)^n) \subset O(17) \subset O(\log\log n) \subset O(\log n) \subset O(\log^2 n) \subset O(\sqrt{n}) \subset O(\sqrt{n}\log^2 n) \\ \subset O(n/\log n) \subset O(n) \subset O(n^2) \subset O((3/2)^n). \end{aligned}$$

Solución al Problema 1.15 (😐)

Para resolver la ecuación

$$T(n) = \frac{1}{n}\left(\sum_{i=0}^{n-1} T(i)\right) + cn\ ,$$

siendo $T(0) = 0$, podemos reescribirla como:

$$nT(n) = \sum_{i=0}^{n-1} T(i) + cn^2 \qquad [1.7]$$

Por otro lado, para n–1 obtenemos:

$$(n-1)T(n-1) = \sum_{i=0}^{n-2} T(i) + c(n-1)^2 \qquad [1.8]$$

Restando [1.7] y [1.8]:

$$nT(n) - nT(n–1) + T(n–1) = T(n–1) + c(2n–1) \Rightarrow nT(n) = nT(n–1) + c(2n–1) \Rightarrow$$

$$T(n) = T(n–1) + c(2–1/n).$$

Desarrollando telescópicamente la ecuación en recurrencia:

$$\begin{aligned} T(n) &= T(n–1) + c(2 - 1/n) = \\ &= T(n–2) + c(2 - 1/(n–1)) + c(2 - 1/n) = \\ &= T(n–3) + c(2 - 1/(n–2)) + c(2 - 1/(n–1)) + c(2 - 1/n) = \\ &\ldots\ldots\ldots \\ &= T(0) + c\sum_{i=1}^{n}\left(2 - \frac{1}{i}\right) = \\ &= c\sum_{i=1}^{n}\left(2 - \frac{1}{i}\right) \end{aligned}$$

ya que teníamos que $T(0) = 0$. Veamos cual es el orden de $T(n)$:

a) Como $(2–1/i) \le 2$ para todo $i > 0$, $T(n) \le c\sum_{i=1}^{n} 2 = 2cn \Rightarrow T(n) \in O(n)$.

b) Como $(2–1/i) \ge 1$ para todo $i > 0$, $T(n) \ge c\sum_{i=1}^{n} 1 = cn \Rightarrow T(n) \in \Omega(n)$.

Por tanto, $T(n) \in \Theta(n)$.

Solución al Problema 1.16

Función *BuscBin* (😐)

a) Para determinar su tiempo de ejecución, calcularemos primero el número de operaciones elementales (OE) que se realizan:
   - En la línea (1) se ejecutan la comparación del *IF* (1 OE), y un acceso a un vector (1 OE), una comparación (1 OE) y un *RETURN* (1 OE) si la condición es verdadera.
   - En la línea (3) se realizan 3 OE (suma, división y asignación).
   - En la línea (4) hay un acceso a un vector (1 OE) y una comparación (1 OE), y además 1 OE en caso de que la condición del *IF* sea verdadera.
   - En la línea (5) hay un acceso a un vector (1 OE) y una comparación (1 OE).
   - Las líneas (6) y (8) efectúan $3+T(n/2)$ cada una: una operación aritmética (incremento o decremento de 1), una llamada a la función *BuscBin* (lo que supone 1 OE), más lo que tarde en ejecutarse la función con la mitad de los elementos y un *RETURN* (1 OE).

   Por tanto obtenemos la ecuación en recurrencia $T(n) = 11 + T(n/2)$, con la condición inicial $T(1) = 4$. Para resolverla, haciendo el cambio $t_k = T(2^k)$ obtenemos

$$t_k - t_{k-1} = 11,$$

   ecuación no homogénea cuya ecuación característica es $(x-1)^2 = 0$. Por tanto,

$$t_k = c_1 k + c_2$$

   y, deshaciendo los cambios,

$$T(n) = c_1 \log n + c_2.$$

   Para calcular las constantes, nos basaremos en la condición inicial $T(1) = 4$, junto con el valor de $T(2)$, que podemos calcular apoyándonos en la expresión de la ecuación en recurrencia: $T(2) = 11 + 4 = 15$. Finalmente obtenemos

$$T(n) = 11\log n + 4 \in \Theta(\log n)$$

b) La recursión de este programa, por tratarse de un caso de recursión de cola, puede ser eliminada mediante un bucle que simule las llamadas recursivas a la función. La condición de terminación del bucle puede ser tomada del caso base de la función recursiva y el cuerpo de dicho bucle consiste en una preparación de los argumentos de la función recursiva y el cálculo que ésta realiza:

```
PROCEDURE BuscBIt(a:vector;prim,ult:CARDINAL;x:INTEGER):BOOLEAN;
   VAR mitad:CARDINAL;
BEGIN
   WHILE (prim<ult) DO                    (* 1 *)
      mitad:=(prim+ult)DIV 2;              (* 2 *)
      IF x=a[mitad] THEN RETURN TRUE       (* 3 *)
      ELSIF (x<a[mitad]) THEN              (* 4 *)
         ult:=mitad-1                      (* 5 *)
      ELSE                                 (* 6 *)
```

```
        prim:=mitad+1                    (* 7 *)
      END                                (* 8 *)
    END;                                 (* 9 *)
    RETURN x=a[ult]                      (* 10 *)
END BuscBIt;
```

c) Para el cálculo del tiempo de ejecución y la complejidad de la función no recursiva podemos seguir un proceso análogo al que seguimos para la función *Algoritmo2* (en el problema 1.5). Para determinar el tiempo de ejecución, calcularemos primero el número de operaciones elementales (OE) que se realizan en cada una de las líneas:
   - En la línea (1) se efectúa la condición del bucle, que supone 1 OE (la comparación).
   - Las líneas (2) a (9) componen el cuerpo del bucle, y contabilizan 3, 2+1, 2, 2, 0, 2, 0 y 0 OE respectivamente.
   - Por último, la línea (10) supone 3 OE. A ella se llega cuando la condición del bucle deja de verificarse.

   El bucle se repite hasta que su condición sea falsa, acabando la función al ejecutarse la línea (10). Cada iteración del bucle está compuesta por las líneas (1) a (9), junto con una ejecución adicional de la línea (1) que es la que ocasiona la salida del bucle. En cada iteración se reduce a la mitad los elementos a considerar, por lo que el bucle se repite log$n$ veces. Por tanto, en el peor caso,

$$T(n) = \left(\left(\sum_{i=1}^{\log n}(1+3+2+2+2)\right)+1\right)+3 = 10\log n + 4 \in \Theta(\log n).$$

   Como puede verse, el tiempo de ejecución de ambas funciones es prácticamente igual, lo que a priori implica que cualquiera de las dos pueden usarse indistintamente. Sin embargo, hay que tener en cuenta la mayor complejidad espacial que siempre suponen los procedimientos recursivos por la utilización de la pila, lo que hace que ante una igualdad de tiempos de ejecución, los procedimientos iterativos sean preferibles frente a los recursivos. Pero no sólo la complejidad ha de ser tenida en cuenta para la elección del algoritmo. La claridad y sencillez del código es un factor también a considerar, pues ello va a implicar una mejor legibilidad y una depuración del programa y mantenimiento más fácil, aspectos todos ellos muy importantes.

Función *Sumadigitos* (😐)

a) Para calcular el tiempo de ejecución, calcularemos primero el número de operaciones elementales (OE) que se realizan:
   - En la línea (1) se ejecutan una comparación (1 OE) y un *RETURN* (1 OE) si la condición es verdadera.
   - En la línea (2) se efectúa una división (1 OE), una llamada a la función *Sumadigitos* (1 OE), más lo que tarda ésta con un décimo del tamaño de su entrada, una suma (1 OE), un resto (1 OE), y un *RETURN* (1 OE).

Llamando n al parámetro *num* de la función, obtenemos la ecuación en recurrencia $T(n) = 6 + T(n/10)$, con la condición inicial $T(1) = 2$.

Para resolverla hacemos los cambios $n = 10^k$ (o, lo que es igual, $k = \log_{10} n$) y $t_k = T(10^k)$ y obtenemos

$$t_k - t_{k-1} = 6,$$

ecuación no homogénea cuya ecuación característica es $(x-1)^2 = 0$. Por tanto,

$$t_k = c_1 + c_2 k.$$

Deshaciendo los cambios,

$$T(n) = c_1 + c_2 \log_{10} n.$$

Para calcular las constantes, nos apoyamos en la condición inicial $T(1) = 2$, junto con el valor de $T(10)$, que puede ser calculado apoyándonos en la expresión de la ecuación en recurrencia: $T(10) = 6 + 2 = 8$. Finalmente obtenemos

$$T(n) = 6 \log_{10} n + 2 \in \Theta(\log n)$$

Como vemos, en esta caso la complejidad de la función depende del logaritmo en base 10 de su parámetro *num* (esto es, de su número de dígitos).

b) La recursión de este algoritmo puede ser eliminada mediante un bucle que simule las llamadas recursivas a la función, cuya condición de terminación puede ser tomada del caso base de la función recursiva, y cuyo cuerpo consiste en los cálculos que ésta realiza, junto con una preparación de los argumentos de la siguiente llamada. En concreto, el algoritmo que implementa el algoritmo no recursivo es el siguiente:

```
PROCEDURE Sumadigitos_it(num:CARDINAL):CARDINAL;
  VAR s:CARDINAL;
BEGIN
  s:=num MOD 10;                          (* 1 *)
  WHILE num>=10 DO                        (* 2 *)
     num:=num DIV 10;                     (* 3 *)
     s:=s+(num MOD 10)                    (* 4 *)
  END;                                    (* 5 *)
  RETURN s                                (* 6 *)
END Sumadigitos_it;
```

c) Para determinar el tiempo de ejecución, calcularemos primero el número de operaciones elementales (OE) que se realizan:

- En la línea (1) se ejecutan 2 OE (un resto y una asignación).
- En la línea (2) se efectúa la condición del bucle, que supone 1 OE.
- Las líneas (3) y (4) componen el cuerpo del bucle, y contabilizan 2 y 3 OE respectivamente.
- Por último, la línea (6) supone 1 OE. A ella se llega cuando la condición del bucle deja de verificarse.

Con esto, se efectúa la línea (1), y después se repite el bucle hasta que su condición sea falsa, acabando la función al ejecutarse la línea (6). Cada iteración del bucle está compuesta por las líneas (2) a (4), junto con una ejecución adicional de la línea (2) que es la que ocasiona la salida del bucle. En cada iteración se diezman los elementos a considerar, por lo que el bucle se repite $\log_{10} n$ veces.

Por tanto, $T(n) = 2 + \left(\left(\sum_{i=1}^{\log_{10} n} (1+2+3) \right) + 1 \right) + 1 = 6 \log_{10} n + 4 \in \Theta(\log n)$.

Llegados a este punto vemos que ocurre lo mismo que en el algoritmo anterior. Los tiempos de ejecución de las funciones recursiva e iterativa son similares. Es por tanto una decisión del usuario decantarse por el diseño generalmente más robusto ofrecido por los algoritmos recursivos, frente a la menor complejidad espacial que presentan los iterativos al no utilizar la pila de recursión.

Solución al Problema 1.17 (☺)

a) Para determinar el tiempo de ejecución del algoritmo, calcularemos primero el número de operaciones elementales (OE) que se realizan:

- En la línea (1) se puede ejecutar o sólo la condición (si ésta es falsa) con un total de 1 OE, o bien la sentencia entera, con un total de 3 OE.
- Las líneas (2) y (3) están compuestas por operaciones aritméticas y asignaciones, y se realizan un total de 3 OE en cada una.
- La línea (4) tiene tres partes: un acceso a *a*[*mitad*], que supone 1 OE; la llamada a *Raro(a,prim,prim+terc)*, que supone 2 + $T(n/3)$ (1 de la suma de *prim* y *terc*, 1 OE de la llamada a *Raro*, y luego el tiempo de ejecución de *Raro* para un tercio del número de elementos con los que fue invocada la función original); y la llamada a *Raro(a,ult–terc,ult)*, que supone 2 + $T(n/3)$ OE por la misma razón. El resultado de las tres partes ha de sumarse (lo que supone 2 OE) y luego hacer un *RETURN* (1 OE). En resumen, en la línea (4) se ejecutan un total de 1 + 2 + $T(n/3)$ + 2 + $T(n/3)$ + 2 + 1 = 8 + 2$T(n/3)$ OE.

Con esto:

1. Si $n = 1$ (caso base), se ejecuta sólo la línea (1) y por tanto $T(1) = 3$.
2. Si $n = 3^k$, con $k > 0$, se ejecuta sólo la condición del *IF* (1) y luego el resto de las líneas (2) a (4), con lo cual $T(n) = 15 + 2T(n/3)$.

Tenemos por tanto el tiempo de ejecución del algoritmo definido mediante una ecuación en recurrencia. Para resolverla, haciendo el cambio $n = 3^k$ queda

$$T(3^k) = 2T(3^{k-1}) + 15$$

y llamando t_k a $T(3^k)$ obtenemos

$$t_k = t_{k-1} + 15,$$

ecuación en recurrencia no homogénea de ecuación característica $(x-2)(x-1) = 0$ y consecuentemente

$$t_k = c_1 2^k + c_2 1^k = c_1 2^k + c_2.$$

Cambiando entonces t_k por $T(3^k)$ queda

$$T(3^k) = c_1 2^k + c_2,$$

y deshaciendo el cambio $n = 3^k$ (o, lo que es igual, $k = \log_3 n$), obtenemos finalmente

$$T(n) = c_1 2^{\log_3 n} + c_2 = c_1 n^{\log_3 2} + c_2.$$

Para calcular las constantes necesitamos resolver un sistema de dos ecuaciones con dos incógnitas (c_1 y c_2) basándonos en dos condiciones iniciales de la ecuación en recurrencia. Como de partida sólo disponemos de una ($T(1) = 3$), necesitamos obtener otra. Para ello, apoyándonos en la definición recursiva de T y para $n = 3$, obtenemos

$$T(3) = 15 + 2T(n/3) = 15 + 2T(1) = 21.$$

Por tanto

$$\left.\begin{aligned} 3 = T(1) = c_1 + c_2 \\ 21 = T(3) = 2c_1 + c_2 \end{aligned}\right\} \Rightarrow \left.\begin{aligned} c_1 = 18 \\ c_2 = -15 \end{aligned}\right\}$$

Sustituyendo estos valores en la ecuación, obtenemos finalmente

$$T(n) = 18n^{\log_3 2} - 15.$$

b) $T(n) \in \Theta(n^{\log_3 2})$.

Para justificarlo, basándonos en las propiedades de Θ, basta ver que

$$\lim_{n\to\infty} \frac{T(n)}{n^{\log_3 2}}$$

existe, es acotado y distinto de cero. Pero eso es cierto ya que

$$\lim_{n\to\infty} \frac{T(n)}{n^{\log_3 2}} = \lim_{n\to\infty} \frac{18n^{\log_3 2} - 15}{n^{\log_3 2}} = 18.$$

Capítulo 2

ORDENACIÓN

2.1 INTRODUCCIÓN

Dado un conjunto de n elementos a_1, a_2,..., a_n y una relación de orden total ($\leq$) sobre ellos, el problema de la ordenación consiste en encontrar una permutación de esos elementos ordenada de forma creciente.

Aunque tanto el tipo y tamaño de los elementos como el dispositivo en donde se encuentran almacenados pueden influir en el método que utilicemos para ordenarlos, en este tema vamos a solucionar el caso en que los elementos son números enteros y se encuentran almacenados en un vector.

Si bien existen distintos criterios para clasificar a los algoritmos de ordenación, una posibilidad es atendiendo a su eficiencia. De esta forma, en función de la complejidad que presentan en el caso medio, podemos establecer la siguiente clasificación:

- $\Theta(n^2)$: Burbuja, Inserción, Selección.
- $\Theta(n\log n)$: Mezcla, Montículos, Quicksort.
- Otros: Incrementos $\Theta(n^{1.25})$, Cubetas $\Theta(n)$, Residuos $\Theta(n)$.

En el presente capítulo desarrollaremos todos ellos con detenimiento, prestando especial atención a su complejidad, no sólo en el caso medio sino también en los casos mejor y peor, pues para algunos existen diferencias significativas. Hemos dedicado también una sección a problemas, que recogen muchas de las cuestiones y variaciones que se plantean durante el estudio de los distintos métodos.

Como hemos mencionado anteriormente, nos centraremos en la ordenación de enteros, muchos de los problemas de ordenación que nos encontramos en la práctica son de ordenación de registros mucho más complicados. Sin embargo este problema puede ser fácilmente reducido al de ordenación de números enteros utilizando las claves de los registros, o bien índices. Por otro lado, puede que los datos a ordenar excedan la capacidad de memoria del ordenador, y por tanto deban residir en dispositivos externos. Aunque este problema, denominado *ordenación externa*, presenta ciertas dificultades específicas (véase [AHO87]), los métodos utilizados para resolverlo se basan fundamentalmente en los algoritmos que aquí presentamos.

Antes de pasar a desarrollar los principales algoritmos, hemos considerado necesario precisar algunos detalles de implementación.

- Consideraremos que el tamaño máximo de la entrada y el vector a ordenar vienen dados por las siguientes definiciones:

```
CONST n =...; (* numero maximo de elementos *)
TYPE vector = ARRAY [1..n] OF INTEGER;
```

- Los procedimientos de ordenación que presentamos en este capítulo están diseñados para ordenar cualquier subvector de un vector dado $a[1..n]$. Por eso generalmente poseerán tres parámetros: el nombre del vector que contiene a los elementos (*a*) y las posiciones de comienzo y fin del subvector, como por ejemplo *Seleccion(a,prim,ult)*. Para ordenar todo el vector, basta con invocar al procedimiento con los valores *prim*= 1, *ult* = *n*.
- Haremos uso de dos funciones que permiten determinar la posición de los elementos máximo y mínimo de un subvector dado:

```
PROCEDURE PosMaximo(VAR a:vector;i,j:CARDINAL):CARDINAL;
(* devuelve la posicion del maximo elemento de a[i..j] *)
   VAR pmax,k:CARDINAL;
BEGIN
   pmax:=i;
   FOR k:=i+1 TO j DO
      IF a[k]>a[pmax] THEN
         pmax:=k
      END
   END;
   RETURN pmax;
END PosMaximo;

PROCEDURE PosMinimo(VAR a:vector;i,j:CARDINAL):CARDINAL;
(* devuelve la posicion del minimo elemento de a[i..j] *)
   VAR pmin,k:CARDINAL;
BEGIN
   pmin:=i;
   FOR k:=i+1 TO j DO
      IF a[k]<a[pmin] THEN
         pmin:=k
      END
   END;
   RETURN pmin;
END PosMinimo;
```

- Y utilizaremos un procedimiento para intercambiar dos elementos de un vector:

```
PROCEDURE Intercambia(VAR a:vector;i,j:CARDINAL);
(* intercambia a[i] con a[j] *)
   VAR temp:INTEGER;
BEGIN
   temp:=a[i];
   a[i]:=a[j];
   a[j]:=temp
END Intercambia;
```

Veamos los tiempos de ejecución de cada una de ellas:

a) El tiempo de ejecución de la función *PosMaximo* va a depender, además del tamaño del subvector de entrada, de su ordenación inicial, y por tanto distinguiremos tres casos:
   – En el caso mejor, la condición del *IF* es siempre falsa. Por tanto:

$$T(n) = T(j - i + 1) = 1 + \left(\left(\sum_{k=i+1}^{j} (3+3) \right) + 3 \right) + 1 = 5 + 6(j - i).$$

   – En el caso peor, la condición del *IF* es siempre verdadera. Por consiguiente:

$$T(n) = T(j - i + 1) = 1 + \left(\left(\sum_{k=i+1}^{j} (3+3+1) \right) + 3 \right) + 1 = 5 + 7(j - i).$$

   – En el caso medio, vamos a suponer que la condición del *IF* será verdadera en el 50% de los casos. Por tanto:

$$T(n) = T(j - i + 1) = 1 + \left(\left(\sum_{k=i+1}^{j} (3+3+\frac{1}{2}) \right) + 3 \right) + 1 = 5 + \frac{13}{2}(j - i).$$

   Estos casos corresponden respectivamente a cuando el elemento máximo se encuentra en la primera posición, en la última y el vector está ordenado de forma creciente, o cuando consideramos equiprobables cada una de las n posiciones en donde puede encontrarse el máximo.

   Como podemos apreciar, en cualquiera de los tres casos su complejidad es lineal con respecto al tamaño de la entrada.

b) El tiempo de ejecución de la función *PosMinimo* es exactamente igual al de la función *PosMaximo*.

c) La función *Intercambia* realiza 7 operaciones elementales (3 asignaciones y 4 accesos al vector), independientemente de los datos de entrada.

Nótese que en las funciones *PosMaximo* y *PosMinimo* hemos utilizado el paso del vector *a* por referencia en vez de por valor (mediante el uso de *VAR*) para evitar la copia del vector en la pila de ejecución, lo que incrementaría la complejidad del algoritmo resultante, pues esa copia es de orden O(*n*).

2.2 ORDENACIÓN POR INSERCIÓN

El método de Inserción realiza *n*–1 iteraciones sobre el vector, dejando en la *i*-ésima etapa ($2 \le i \le n$) ordenado el subvector $a[1..i]$. La forma de hacerlo es colocando en cada iteración el elemento $a[i]$ en su sitio correcto, aprovechando el hecho de que el subvector $a[1..i-1]$ ya ha sido previamente ordenado. Este método puede ser implementado de forma iterativa como sigue:

```
PROCEDURE Insercion(VAR a:vector;prim,ult:CARDINAL);
   VAR i,j:CARDINAL; x:INTEGER;
BEGIN
   FOR i:=prim+1 TO ult DO
      x:=a[i]; j:=i-1;
      WHILE (j>=prim) AND (x<a[j]) DO
         a[j+1]:=a[j]; DEC(j)
      END;
      a[j+1]:=x
   END
END Insercion;
```

Para estudiar su complejidad, vamos a estudiar los casos mejor, peor y medio de la llamada al procedimiento *Insercion(a,*1*,n)*.

– En el caso mejor el bucle interno no se realiza nunca, y por tanto:

$$T(n) = \left(\sum_{i=2}^{n} (3+4+4+3) \right) + 3 = 14n - 11 .$$

– En el caso peor hay que llevar cada elemento hasta su posición final, con lo que el bucle interno se realiza siempre de *i*–1 veces. Así, en este caso:

$$T(n) = \left(\sum_{i=2}^{n} \left(3+4+\left(\sum_{j=1}^{i-1} (4+5) \right) +1+3 \right) \right) + 3 = \frac{9}{2} n^2 + \frac{13}{2} n - 10 .$$

– En el caso medio, supondremos equiprobable la posición de cada elemento dentro del vector. Por tanto para cada valor de *i*, la probabilidad de que el elemento se sitúe en alguna posición *k* de las *i* primeras será de 1/*i*. El número de veces que se repetirá el bucle *WHILE* en este caso es (*i*–*k*), con lo cual el número medio de operaciones que se realizan en el bucle es:

$$\left(\frac{1}{i}\sum_{k=1}^{i}9(i-k)\right)+4=\frac{9}{2}i-\frac{1}{2}.$$

Por tanto, el tiempo de ejecución en el caso medio es:

$$T(n)=\left(\sum_{i=2}^{n}\left(3+4+\left(\frac{9}{2}i-\frac{1}{2}\right)+3\right)\right)+3=\frac{9}{4}n^2+\frac{47}{4}n-11.$$

Por el modo en que funciona el algoritmo, tales casos van a corresponder a cuando el vector se encuentra ordenado de forma creciente, decreciente o aleatoria.

Como podemos ver, en este método los órdenes de complejidad de los casos peor, mejor y medio difieren bastante. Así en el mejor caso el orden de complejidad resulta ser lineal, mientras que en los casos peor y medio su complejidad es cuadrática.

Este método se muestra muy adecuado para aquellas situaciones en donde necesitamos ordenar un vector del que ya conocemos que está casi ordenado, como suele suceder en aquellas aplicaciones de inserción de elementos en bancos de datos previamente ordenados cuya ordenación total se realiza periódicamente.

2.3 ORDENACIÓN POR SELECCIÓN

En cada paso (i=1...n–1) este método busca el mínimo elemento del subvector $a[i..n]$ y lo intercambia con el elemento en la posición i:

```
PROCEDURE Seleccion(VAR a:vector;prim,ult:CARDINAL);
   VAR i:CARDINAL;
BEGIN
   FOR i:=prim TO ult-1 DO
      Intercambia(a,i,PosMinimo(a,i,ult))
   END
END Seleccion;
```

En cuanto a su complejidad, vamos a estudiar los casos mejor, peor y medio de la llamada al procedimiento *Seleccion(a,*1*,n)*, que van a coincidir con los mismos casos (mejor, peor y medio) que los de la función *PosMinimo*.

– En el caso mejor:

$$T(n)=\left(\sum_{i=1}^{n-1}\left(3+1+\left(5+6(n-i)\right)+1+7\right)\right)+3=3n^2+14n-14.$$

– En el caso peor:

$$T(n)=\left(\sum_{i=1}^{n-1}\left(3+1+\left(5+7(n-i)\right)+1+7\right)\right)+3=\frac{7}{2}n^2+\frac{27}{2}n-14.$$

– En el caso medio:

$$T(n)=\left(\sum_{i=1}^{n-1}\left(3+1+\left(5+\frac{13}{2}(n-i)\right)+1+7\right)\right)+3=\frac{13}{4}n^2+\frac{55}{4}n-14.$$

En consecuencia, el algoritmo es de complejidad cuadrática.

Este método, por el número de operaciones de comparación e intercambio que realiza, es el más adecuado para ordenar pocos registros de gran tamaño. Si el tipo base del vector a ordenar no es entero, sino un tipo más complejo (guías telefónicas, índices de libros, historiales hospitalarios, etc.) deberemos darle mayor importancia al intercambio de valores que a la comparación entre ellos en la valoración del algoritmo por el coste que suponen. En este sentido, analizando el número de intercambios que realiza el método de Selección vemos que es de orden O(n), frente al orden O(n^2) de intercambios que presentan los métodos de Inserción o Burbuja.

2.4 ORDENACIÓN BURBUJA

Este método de ordenación consiste en recorrer los elementos siempre en la misma dirección, intercambiando elementos adyacentes si fuera necesario:

```
PROCEDURE Burbuja (VAR a:vector;prim,ult:CARDINAL);
   VAR i,j:CARDINAL;
BEGIN
   FOR i:=prim TO ult-1 DO
      FOR j:=ult TO i+1 BY -1 DO
         IF (a[j-1]>a[j]) THEN
            Intercambia(a,j-1,j)
         END
      END
   END
END Burbuja;
```

El nombre de este algoritmo trata de reflejar cómo el elemento mínimo “sube”, a modo de burbuja, hasta el principio del subvector.

Respecto a su complejidad, vamos a estudiar los casos mejor, peor y medio de la llamada al procedimiento *Burbuja(a,*1*,n)*.

– En el caso mejor:

$$T(n)=\left(\sum_{i=1}^{n-1}\left(3+\sum_{j=i+1}^{n}(3+4)+3\right)\right)+3=\frac{7}{2}n^2+\frac{5}{2}n-3.$$

– En el caso peor:

$$T(n)=\left(\sum_{i=1}^{n-1}\left(3+\sum_{j=i+1}^{n}(3+4+2+7)+3\right)\right)+3=8n^2-2n-1.$$

– En el caso medio:

$$T(n)=\left(\sum_{i=1}^{n-1}\left(3+\sum_{j=i+1}^{n}\left(3+4+\frac{2+7}{2}\right)+3\right)\right)+3=\frac{23}{4}n^2+\frac{1}{4}n-1.$$

En consecuencia, el algoritmo es de complejidad cuadrática.

Este algoritmo funciona de forma parecida al de Selección, pero haciendo más trabajo para llevar cada elemento a su posición. De hecho es el peor de los tres vistos hasta ahora, no sólo en cuanto al tiempo de ejecución, sino también respecto al número de comparaciones y de intercambios que realiza.

Una posible mejora que puede admitir este algoritmo es el control de la existencia de una pasada sin intercambios; en ese momento el vector estará ordenado.

2.5 ORDENACIÓN POR MECLA (MERGESORT)

Este método utiliza la técnica de Divide y Vencerás para realizar la ordenación del vector *a*. Su estrategia consiste en dividir el vector en dos subvectores, ordenarlos mediante llamadas recursivas, y finalmente combinar los dos subvectores ya ordenados. Esta idea da lugar a la siguiente implementación:

```
PROCEDURE Mezcla(VAR a,b:vector;prim,ult:CARDINAL);
(* utiliza el vector b como auxiliar para realizar la mezcla *)
   VAR mitad:CARDINAL;
BEGIN
   IF prim<ult THEN
      mitad:=(prim+ult)DIV 2;
      Mezcla(a,b,prim,mitad);
      Mezcla(a,b,mitad+1,ult);
      Combinar(a,b,prim,mitad,mitad+1,ult)
   END
END Mezcla;
```

Una posible implementación de la función que lleva a cabo el proceso de mezcla vuelca primero los elementos a ordenar en el vector auxiliar para después, utilizando dos índices, uno para cada subvector, rellenar el vector ordenadamente. Nótese que el algoritmo utiliza el hecho de que los dos subvectores están ya ordenados y que son además consecutivos.

```
PROCEDURE Combinar(VAR a,b:vector;p1,u1,p2,u2:CARDINAL);
(* mezcla ordenadamente los subvectores a[p1..u1] y a[p2..u2]
   suponiendo que estos estan ya ordenados y que son consecutivos
   (p2=u1+1), utilizando el vector auxiliar b *)
   VAR i1,i2,k:CARDINAL;
BEGIN
   IF (p1>u1) OR (p2>u2) THEN RETURN END;
   FOR k:=p1 TO u2 DO b[k]:=a[k] END;  (* volcamos a en b *)
   i1:=p1;i2:=p2;  (* cada indice se encarga de un subvector *)
   FOR k:=p1 TO u2 DO
      IF b[i1]<=b[i2] THEN
         a[k]:=b[i1];
         IF i1<u1 THEN INC(i1) ELSE b[i1]:=MAX(INTEGER) END
      ELSE
         a[k]:=b[i2];
         IF i2<u2 THEN INC(i2) ELSE b[i2]:=MAX(INTEGER) END
      END
   END
END Combinar;
```

En cuanto al estudio de su complejidad, siguiendo el mismo método que hemos utilizado en los problemas del primer capítulo, se llega a que el tiempo de ejecución de *Mezcla(a,b,*1*,n)* puede expresarse mediante una ecuación en recurrencia:

$$T(n) = 2T(n/2) + 16n + 17$$

con la condición inicial $T(1) = 1$. Ésta es una ecuación en recurrencia no homogénea cuya ecuación característica asociada es $(x-2)^2(x-1) = 0$, lo que permite expresar $T(n)$ como:

$$T(n) = c_1 n + c_2 n\log n + c_3.$$

El cálculo de las constantes puede hacerse en base a la condición inicial, lo que nos lleva a la expresión final:

$$T(n) = 16n\log n + 18n - 17 \in \Theta(n\log n).$$

Obsérvese que este método ordena n elementos en tiempo $\Theta(n\log n)$ en cualquiera de los casos (peor, mejor o medio). Sin embargo tiene una complejidad espacial, en cuanto a memoria, mayor que los demás (del orden de n).

Otras versiones de este algoritmo no utilizan el vector auxiliar *b*, sino que trabajan sobre el propio vector a ordenar, combinando sobre él los subvectores

obtenidos de las etapas anteriores. Si bien es cierto que esto consigue ahorrar espacio (un vector auxiliar), también complica el código del algoritmo resultante.

El método de ordenación por Mezcla se adapta muy bien a distintas circunstancias, por lo que es comúnmente utilizado no sólo para la ordenación de vectores. Por ejemplo, el método puede ser también implementado de forma que el acceso a los datos se realice de forma secuencial, por lo que hay diversas estructuras (como las listas enlazadas) para las que es especialmente apropiado. También se utiliza para realizar ordenación externa, en donde el vector a ordenar reside en dispositivos externos de acceso secuencial (i.e. ficheros).

2.6 ORDENACIÓN MEDIANTE MONTÍCULOS (HEAPSORT)

La filosofía de este método de ordenación consiste en aprovechar la estructura particular de los montículos (heaps), que son árboles binarios completos (todos sus niveles están llenos salvo a lo sumo el último, que se rellena de izquierda a derecha) y cuyos nodos verifican la propiedad del montículo: todo nodo es mayor o igual que cualquiera de sus hijos. En consecuencia, en la raíz se encuentra siempre el elemento mayor.

Estas estructuras admiten una representación muy sencilla, compacta y eficiente mediante vectores (por ser árboles completos). Así, en un vector que represente una implementación de un montículo se cumple que el "padre" del i-ésimo elemento del vector se encuentra en la posición $i \div 2$ (menos la raíz, claro) y sus "hijos", si es que los tiene, estarán en las posiciones $2i$ y $2i+1$ respectivamente.

La idea es construir, con los elementos a ordenar, un montículo sobre el propio vector. Una vez construido el montículo, su elemento mayor se encuentra en la primera posición del vector ($a[prim]$). Se intercambia entonces con el último ($a[ult]$) y se repite el proceso para el subvector $a[prim..ult-1]$. Así sucesivamente hasta recorrer el vector completo. Esto nos lleva a un algoritmo de orden de complejidad $O(n \log n)$ cuya implementación puede ser la siguiente:

```
PROCEDURE Monticulos(VAR a:vector;prim,ult:CARDINAL);
   VAR i:CARDINAL;
BEGIN
   HacerMonticulo(a,prim,ult);
   FOR i:=ult TO prim+1 BY -1 DO
      Intercambia(a,prim,i);
      Empujar(a,prim,i-1,prim)
   END
END Monticulos;
```

Los procedimientos *HacerMontículo* y *Empujar* son, respectivamente, el que construye un montículo a partir del subvector $a[prim..ult]$ dado, y el que "empuja" un elemento hasta su posición definitiva en el montículo, reconstruyendo la estructura de montículo en el subvector $a[prim..ult-1]$:

```
PROCEDURE HacerMonticulo(VAR a:vector;prim,ult:CARDINAL);
(* construye un monticulo a partir de a[prim..ult] *)
   VAR i:CARDINAL;
BEGIN
   FOR i:=(ult-prim+1)DIV 2 TO 1 BY -1 DO
      Empujar(a,prim,ult,prim+i-1)
   END
END HacerMonticulo;


PROCEDURE Empujar(VAR a:vector;prim,ult,i:CARDINAL);
(* empuja el elemento en posicion i hasta su posicion final *)
   VAR j,k:CARDINAL;
BEGIN
   k:=i-prim+1;
   REPEAT
      j:=k;
      IF (2*j<=ult-prim+1) AND (a[2*j+prim-1]>a[k+prim-1]) THEN
         k:=2*j
      END;
      IF (2*j<ult-prim+1) AND (a[2*j+prim]>a[k+prim-1]) THEN
         k:=2*j+1
      END;
      Intercambia(a,j+prim-1,k+prim-1);
   UNTIL j=k
END Empujar;
```

Para estudiar la complejidad del algoritmo hemos de considerar dos partes. La primera es la que construye inicialmente el montículo a partir de los elementos a ordenar y la segunda va recorriendo en cada iteración un subvector más pequeño, colocando el elemento raíz en su posición correcta dentro del montículo. En ambos casos nos basamos en la función que "empuja" elementos en el montículo.

Observando el comportamiento del algoritmo, la diferencia básica entre el caso peor y el mejor está en la profundidad que hay que recorrer cada vez que necesitamos "empujar" un elemento. Si el elemento es menor que todos los demás, necesitaremos recorrer todo el árbol (profundidad: log$n$); si el elemento es mayor o igual que el resto, no será necesario.

El procedimiento *HacerMonticulo* es de complejidad O(n) en el peor caso, puesto que si k es la altura del montículo (k = log$n$), el algoritmo transforma primero cada uno de los dos subárboles que cuelgan de la raíz en montículos de altura a lo más k–1 (el subárbol derecho puede tener altura k–2), y después empuja la raíz hacia abajo, por un camino que a lo más es de longitud k. Esto lleva a lo más un tiempo $t(k)$ de orden de complejidad O(k) con lo cual

$$T(k) \leq 2T(k-1) + t(k),$$

ecuación en recurrencia cuya solución verifica que $T(k) \in O(2^k)$. Como k = log$n$, la complejidad de *HacerMonticulo* es lineal en el peor caso. Este caso ocurre cuando

hay que recorrer siempre la máxima profundidad al empujar a cada elemento, lo que sucede si el vector está originalmente ordenado de forma creciente.

Respecto al mejor caso de *HacerMonticulo*, éste se presenta cuando la profundidad a la que hay que empujar cada elemento es cero. Esto se da, por ejemplo, si todos los elementos del vector son iguales. En esta situación la complejidad del algoritmo es O(1).

Estudiemos ahora los casos mejor y peor del resto del algoritmo *Montículos*. En esta parte hay un bucle que se ejecuta siempre $n-1$ veces, y la complejidad de la función que intercambia dos elementos es O(1). Todo va a depender del procedimiento *Empujar*, es decir, de la profundidad a la que haya que empujar la raíz del montículo en cada iteración, sabiendo que cada montículo tiene $n-i$ elementos, y por tanto una altura de $\log(n-i)$, siendo i el número de la iteración.

En el peor caso, la profundidad a la que hay que empujar las raíces respectivas es la máxima, y por tanto la complejidad de esta segunda parte del algoritmo es $O(n\log n)$. ¿Cuándo ocurre esto? Cuando el elemento es menor que todos los demás. Pero esto sucede siempre que los elementos a ordenar sean distintos, por la forma en la que se van escogiendo las nuevas raíces.

En el caso mejor, aunque el bucle se sigue repitiendo $n-1$ veces, las raíces no descienden, por ser mayores o iguales que el resto de los elementos del montículo. Así, la complejidad de esta parte del algoritmo es de orden $O(n)$. Pero este caso sólo se dará si los elementos del vector son iguales, por la forma en la que originariamente se construyó el montículo y por cómo se escoge la nueva raíz en cada iteración (el último de los elementos, que en un montículo ha de ser de los menores).

2.7 ORDENACIÓN RÁPIDA DE HOARE (QUICKSORT)

Este método es probablemente el algoritmo de ordenación más utilizado, pues es muy fácil de implementar, trabaja bien en casi todas las situaciones y consume en general menos recursos (memoria y tiempo) que otros métodos.

Su diseño está basado en la técnica de Divide y Vencerás, que estudiaremos en el siguiente capítulo, y consta de dos partes:

a) En primer lugar el vector a ordenar $a[prim..ult]$ es dividido en dos subvectores no vacíos $a[prim..l-1]$ y $a[l+1..ult]$, tal que todos los elementos del primero son menores que los del segundo. El elemento de índice l se denomina *pivote* y se calcula como parte del procedimiento de partición.

b) A continuación, los dos subvectores son ordenados mediante llamadas recursivas a Quicksort. Como los subvectores se ordenan sobre ellos mismos, no es necesario realizar ninguna operación de combinación.

Esto da lugar al siguiente procedimiento, que constituye la versión clásica del algoritmo de ordenación rápida de Hoare:

```
PROCEDURE Quicksort(VAR a:vector;prim,ult:CARDINAL);
   VAR l:CARDINAL;
BEGIN
   IF prim<ult THEN
      l:=Pivote(a,a[prim],prim,ult);
      Quicksort(a,prim,l-1);
      Quicksort(a,l+1,ult)
   END
END Quicksort;
```

La función *Pivote* parte del elemento pivote y permuta los elementos del vector de forma que al finalizar la función, todos los elementos menores o iguales que el pivote estén a su izquierda, y los elementos mayores que él a su derecha. Devuelve la posición en la que ha quedado situado el pivote *p*:

```
PROCEDURE Pivote(VAR a:vector;p:INTEGER;prim,ult:CARDINAL)
                                                    :CARDINAL;
  (* permuta los elementos de a[prim..ult] y devuelve una
     posicion l tal que prim<=l<=ult, a[i]<=p si prim<=i<l,
     a[l]=p, y a[i]>p si l<i<=ult, donde p es el valor inicial
     de a[prim] *)
   VAR i,l:CARDINAL;
BEGIN
   i:=prim; l:=ult+1;
   REPEAT INC(i) UNTIL (a[i]>p) OR (i>=ult);
   REPEAT DEC(l) UNTIL (a[l]<=p);
   WHILE i<l DO
      Intercambia(a,i,l);
      REPEAT INC(i) UNTIL (a[i]>p);
      REPEAT DEC(l) UNTIL (a[l]<=p)
   END;
   Intercambia(a,prim,l);
   RETURN l
END Pivote;
```

Este método es de orden de complejidad $\Theta(n^2)$ en el peor caso y $\Theta(n\log n)$ en los casos mejor y medio. Para ver su tiempo de ejecución, utilizando los mecanismos expuestos en el primer capítulo, obtenemos la siguiente ecuación en recurrencia:

$$T(n) = 8 + T(a) + T(b) + T_{Pivote}(n)$$

donde a y b son los tamaños en los que la función *Pivote* divide al vector (por tanto podemos tomar que $a + b = n$), y $T_{Pivote}(n)$ es la función que define el tiempo de ejecución de la función *Pivote*.

El procedimiento *Quicksort* "rompe" la filosofía de caso mejor, peor y medio de los algoritmos clásicos de ordenación, pues aquí tales casos no dependen de la ordenación inicial del vector, sino de la elección del pivote.

Así, el mejor caso ocurre cuando $a = b = n/2$ en todas las invocaciones recursivas del procedimiento, pues en este caso obtenemos $T_{Pivote}(n) = 13 + 4n$, y por tanto:

$$T(n) = 21 + 4n + 2T(n/2).$$

Resolviendo esta ecuación en recurrencia y teniendo en cuenta las condiciones iniciales $T(0) = 1$ y $T(1) = 27$ se obtiene la expresión final de $T(n)$, en este caso:

$$T(n) = 15n\log n + 26n + 1.$$

Ahora bien, si $a = 0$ y $b = n-1$ (o viceversa) en todas las invocaciones recursivas del procedimiento, $T_{Pivote}(n) = 11 + 39/8n$, obteniendo:

$$T(n) = 19 + 39/8n + T(n-1).$$

Resolviendo la ecuación para las mismas condiciones iniciales, nos encontramos con una desagradable sorpresa:

$$T(n) = \frac{3}{8}n^2 + \frac{213}{8}n + 1 \in \Theta(n^2).$$

En consecuencia, la elección idónea para el pivote es la mediana del vector en cada etapa, lo que ocurre es que encontrarla requiere un tiempo extra que hace que el algoritmo se vuelva más ineficiente en la mayoría de los casos (ver problema 2.15). Por esa razón como pivote suele escogerse un elemento cualquiera, a menos que se conozca la naturaleza de los elementos a ordenar. En nuestro caso, como a priori suponemos equiprobable cualquier ordenación inicial del vector, hemos escogido el primer elemento del vector, que es el que se le pasa como segundo argumento a la función *Pivote*.

Esta elección lleva a tres casos desfavorables para el algoritmo: cuando los elementos son todos iguales y cuando el vector está inicialmente ordenado en orden creciente o decreciente. En estos casos la complejidad es cuadrática puesto que la partición se realiza de forma totalmente descompensada.

A pesar de ello suele ser el algoritmo más utilizado, y se demuestra que su tiempo promedio es menor, en una cantidad constante, al de todos los algoritmos de ordenación de complejidad $O(n\log n)$. En todo esto es importante hacer notar, como hemos indicado antes, la relevancia que toma una buena elección del pivote, pues de su elección depende considerablemente el tiempo de ejecución del algoritmo.

Sobre el algoritmo expuesto anteriormente pueden realizarse varias mejoras:

1. Respecto a la elección del pivote. En vez de tomar como pivote el primer elemento, puede seguirse alguna estrategia del tipo:

   - Tomar al azar tres elementos seguidos del vector y escoger como pivote el elemento medio de los tres.
   - Tomar k elementos al azar, clasificarlos por cualquier método, y elegir el elemento medio como pivote.

2. Con respecto al tamaño de los subvectores a ordenar. Cuando el tamaño de éstos sea pequeño (menor que una cota dada), es posible utilizar otro algoritmo de ordenación en vez de invocar recursivamente a Quicksort. Esta idea utiliza el hecho de que algunos métodos, como Selección o Inserción, se comportan muy bien cuando el número de datos a ordenar son pocos, por disponer de constantes multiplicativas pequeñas. Aun siendo de orden de complejidad cuadrática, son más eficientes que los de complejidad $n\log n$ para valores pequeños de n.

En los problemas propuestos y resueltos se desarollan más a fondo estas ideas.

2.8 ORDENACIÓN POR INCREMENTOS (SHELLSORT)

La ordenación por inserción puede resultar lenta pues sólo intercambia elementos adyacentes. Así, si por ejemplo el elemento menor está al final del vector, hacen falta n pasos para colocarlo donde corresponde. El método de Incrementos es una extensión muy simple y eficiente del método de Inserción en el que cada elemento se coloca *casi* en su posición definitiva en la primera pasada.

El algoritmo consiste básicamente en dividir el vector a en h subvectores:

$$a[k],\ a[k+h],\ a[k+2h],\ a[k+3h],\ ...$$

y ordenar por inserción cada uno de esos subvectores (k=1,2,...,h–1).

Un vector de esta forma, es decir, compuesto por h subvectores ordenados intercalados, se denomina h-ordenado. Haciendo h-ordenaciones de a para valores grandes de h permitimos que los elementos puedan moverse grandes distancias dentro del vector, facilitando así las h-ordenaciones para valores más pequeños de h. A h se le denomina incremento.

Con esto, el método de ordenación por Incrementos consiste en hacer h-ordenaciones de a para valores de h decreciendo hasta llegar a uno.

El número de comparaciones que se realizan en este algoritmo va a depender de la secuencia de incrementos h, y será mayor que en el método clásico de Inserción (que se ejecuta finalmente para h = 1), pero la potencia de este método consiste en conseguir un número de intercambios mucho menor que con la Inserción clásica.

El procedimiento presentado a continuación utiliza la secuencia de incrementos h = ..., 1093, 364, 121, 40, 13, 1. Otras secuencias pueden ser utilizadas, pero la elección ha de hacerse con cuidado. Por ejemplo la secuencia ..., 64, 32, 16, 8, 4, 2, 1 es muy ineficiente pues los elementos en posiciones pares e impares no son comparados hasta el último momento. En el ejercicio 2.7 se discute más a fondo esta circunstancia.

```
PROCEDURE Incrementos(VAR a:vector;prim,ult:CARDINAL);
   VAR i,j,h,N:CARDINAL; v:INTEGER;
BEGIN
   N:=(ult-prim+1);  (* numero de elementos *)
   h:=1;
```

```
   REPEAT h:=3*h+1 UNTIL h>N;  (* construimos la secuencia *)
   REPEAT
      h:=h DIV 3;
      FOR i:=h+1 TO N DO
         v:=a[i]; j:=i;
         WHILE (j>h) AND (a[j-h+prim-1]>v) DO
            a[j+prim-1]:=a[j-h+prim-1];
            DEC(j,h)
         END;
         a[j+prim-1]:=v;
      END
   UNTIL h=1
END Incrementos;
```

En cuanto al estudio de su complejidad, este método es diferente al resto de los procedimientos vistos en este capítulo. Su complejidad es difícil de calcular y depende mucho de la secuencia de incrementos que utilice. Por ejemplo, para la secuencia dada existen dos conjeturas en cuanto a su orden de complejidad: $n\log^2 n$ y $n^{1.25}$. En general este método es el escogido para muchas aplicaciones reales por ser muy simple teniendo un tiempo de ejecución aceptable incluso para grandes valores de n.

2.9 OTROS ALGORITMOS DE ORDENACIÓN

Los algoritmos vistos hasta ahora se basan en la ordenación de vectores de números enteros cualesquiera, sin ningún tipo de restricción. En este apartado veremos cómo pueden encontrarse algoritmos de orden O(n) cuando dispongamos de información adicional sobre los valores a ordenar.

2.9.1 Ordenación por Cubetas (Binsort)

Suponemos que los datos a ordenar son números naturales, todos distintos y comprendidos en el intervalo [1,n]. Es decir, nuestro problema es ordenar un vector con los n primeros números naturales. Bajo esas circunstancias es posible implementar un algoritmo de complejidad temporal O(n). Es el método de ordenación por Cubetas, en donde en cada iteración se sitúa un elemento en su posición definitiva:

```
PROCEDURE Cubetas(VAR a:vector);
   VAR i:CARDINAL;
BEGIN
   FOR i:=1 TO n DO
      WHILE a[i]<>i DO
```

```
            Intercambia(a,i,a[i])
          END
        END
    END Cubetas;
```

2.9.2 Ordenación por Residuos (Radix)

Este método puede utilizarse cuando los valores a ordenar están compuestos por secuencias de letras o dígitos que admiten un orden lexicográfico. Éste es el caso de palabras, números (cuyos dígitos admiten este orden) o bien fechas.

El método consiste en definir k colas (numeradas de 0 a k–1) siendo k los posibles valores que puede tomar cada uno de los dígitos que componen la secuencia. Una vez tengamos las colas habría que repetir, para i a partir de 0 y hasta llegar al número máximo de dígitos o letras de nuestras cadenas:

1. Distribuir los elementos en las colas en función del dígito i.
2. Extraer ordenada y consecutivamente los elementos de las colas, introduciéndolos de nuevo en el vector.

Los elementos quedan ordenados sin haber realizado ninguna comparación. Veamos un ejemplo de este método. Supongamos el vector:

[0, 1, 81, 64, 23, 27, 4, 25, 36, 16, 9, 49].

En este caso se trata de números naturales en base 10, que no son sino secuencias de dígitos. Como cada uno de los dígitos puede tomar 10 valores (del 0 al 9), necesitaremos 10 colas. En la primera pasada introducimos los elementos en las colas de acuerdo a su dígito menos significativo:

Cola	0	1	2	3	4	5	6	7	8	9
	0	81,1		23	4,64	25	16,36	27		49,9

y ahora extraemos ordenada y sucesivamente los valores, obteniendo el vector:

[0, 81, 1, 23, 4, 64, 25, 16, 36, 27, 49, 9].

Volvemos a realizar otra pasada, esta vez fijándonos en el segundo dígito menos significativo:

Cola	0	1	2	3	4	5	6	7	8	9
	9,4,1,0	16	27,25,23	36	49		64		81	

Volviendo a extraer ordenada y sucesivamente los valores obtenemos el vector [0, 1, 4, 9, 16, 23, 25, 27, 36, 49, 64, 81]. Como el máximo de dígitos de los números a ordenar era de dos, con dos pasadas hemos tenido suficiente.

La implementación de este método queda resuelta en el problema 2.9, en donde se discute también su complejidad espacial, inconveniente principal de estos métodos tan eficientes.

2.10 PROBLEMAS PROPUESTOS

2.1. En los algoritmos Burbuja o Selección, el elemento más pequeño de $a[i..n]$ es colocado en la posición i mediante intercambios, para valores sucesivos de i. Otra posibilidad es colocar el elemento máximo de $a[1..j]$ en la posición j, para valores de j entre n y 1. A este algoritmo se le denomina ordenación por Ladrillos (Bricksort). Implementar dicho algoritmo y estudiar su complejidad.

2.2. Una variante curiosa de los algoritmos anteriores resulta al combinar los métodos de la Burbuja y de los Ladrillos. La idea es ir colocando alternativamente el mayor valor de $a[1..j]$ en $a[j]$, y el menor valor de $a[i..n]$ en $a[i]$. Implementar dicho algoritmo, conocido como Sacudidas (Shakersort), y estudiar su complejidad.

2.3. Modificar el algoritmo de ordenación por Selección de forma que se intercambien los elementos únicamente si son distintos. ¿Qué impacto tiene esta modificación sobre la complejidad del algoritmo?

2.4. Modificar los algoritmos Quicksort y Mezcla de forma que sustituyan las llamadas recursivas por llamadas al procedimiento Selección cuando el tamaño del vector a ordenar sea menor que una cota dada M.

2.5. ¿Cuándo se presentan en el método de ordenación mediante Montículos el mejor y el peor caso?

2.6. Realizar implementaciones iterativas para los procedimientos Quicksort y Mezcla. Estudiar sus complejidades (espacio y tiempo) y comparar los resultados con los obtenidos para las versiones recursivas.

2.7. Para el algoritmo de ordenación por Incrementos, dar un ejemplo que muestre que 2^k,...,8,4,2,1 no es una buena secuencia de incrementos.

2.8. En el método de ordenación Quicksort, ¿qué ocurre si todos los elementos son iguales? ¿Cómo puede modificarse el algoritmo para optimizar este caso especial?

2.9. Implementar el método *Residuos* para ordenar números naturales a partir de su representación (*a*) decimal y (*b*) binaria. Estudiar detalladamente las complejidades de los algoritmos resultantes.

2.10. Un algoritmo de ordenación se denomina *estable* si, dados dos elementos con claves iguales, después de ordenarlos tienen el mismo orden que tenían antes de la clasificación. La estabilidad es importante cuando un vector ha sido ya ordenado por una clave y necesita ser ordenado por otra. Averiguar cuales de los métodos siguientes son estables y cuales no: Selección, Inserción, Burbuja, Incrementos, Quicksort, Mezcla, Montículos, Ladrillos

y Sacudidas. Para aquellos que no lo sean, dar un ejemplo que corrobore la afirmación y proponer modificaciones al método para convertirlo en estable.

2.11. Modificar el método de Inserción de manera que use la búsqueda binaria para localizar dónde introducir el siguiente elemento. Estudiar el impacto de esta mejora en la complejidad del algoritmo y decidir si es rentable o no.

2.12. El algoritmo de ordenación por Rastreo de un vector funciona de la siguiente manera: comienza por el principio del vector y se mueve hacia el final del vector, comparando los pares de elementos adyacentes hasta encontrar uno que no esté en orden correcto. Lo intercambia y comienza a moverse hacia el principio, intercambiando pares hasta encontrar un par en el orden correcto. Entonces se limita a cambiar de dirección y comienza otra vez hacia el final del vector, buscando de nuevo un par fuera de orden. Una vez que alcanza el extremo final del vector, su misión ha terminado. Implementar dicho algoritmo y calcular su tiempo de ejecución en los casos mejor, peor y medio.

2.13. En el método de ordenación por Mezcla, en vez de dividir el vector $a[1..n]$ en dos mitades, podríamos dividirlo en tres subvectores de tamaños $n \div 3$, $(n+1) \div 3$ y $(n+2) \div 3$, ordenarlos recursivamente, y luego combinarlos. Implementar este algoritmo, calcular su complejidad y compararlo con Mezcla.

2.14. Supongamos el siguiente procedimiento:

```
PROCEDURE OrdenarTres(VAR a:vector;prim,ult:CARDINAL);
   VAR k:CARDINAL; temp:INTEGER;
BEGIN
   IF a[prim]>a[ult] THEN
      temp:=a[prim]; a[prim]:=a[ult]; a[ult]:=temp
   END;
   IF prim+1>=ult THEN RETURN END;
   k:=(ult-prim+1)DIV 3;
   OrdenarTres(a,prim,ult-k); (* primeros 2/3 *)
   OrdenarTres(a,prim+k,ult); (* ultimos  2/3 *)
   OrdenarTres(a,prim,ult-k); (* otra vez los primeros 2/3 *)
END OrdenarTres;
```

Calcular el tiempo de ejecución de *OrdenarTres(a,*1*,n)* en función de *n* y su orden de complejidad, comparándolo con los otros métodos de ordenación.

2.15. *El problema del k-ésimo elemento*: Dado un vector de enteros, queremos encontrar el elemento que ocuparía la posición k si el vector estuviera ordenado en orden creciente (esto es, el k-ésimo menor elemento). Una primera idea para resolver este problema consiste en ordenar primero el vector y después escoger el elemento en la posición k, pero la complejidad de este algoritmo es $O(n\log n)$. ¿Puede hacerse de alguna forma más eficiente? Considerar las dos siguientes ideas y comparar sus complejidades:

- Ordenar el vector sólo hasta la posición k, utilizando un método incremental como el de Selección.
- Utilizar un procedimiento basado en la idea de Quicksort, escogiendo como pivote el elemento en la posición k del vector.

2.16. Un vector contiene n elementos. Se desea encontrar los m elementos más pequeños del vector, con $m<n$. Indicar cuál de las siguientes opciones es la mejor, justificando la respuesta:

a) Ordenar el vector entero y escoger los m primeros elementos.

b) Ordenar los m primeros elementos del vector usando repetidamente el procedimiento de Selección.

c) Invocar m veces al procedimiento que encuentra el k-ésimo elemento (problema 2.15), con los subvectores apropiados.

d) Mediante otro método.

2.17. Supongamos un vector como en el problema anterior, pero ahora queremos encontrar los elementos que ocuparían las posiciones $n\div 2$, $(n\div 2)+1,\ldots,(n\div 2)+m-1$ si el vector estuviese ordenado. Indicar cuál de las siguientes opciones es la mejor, justificando la respuesta:

a) Ordenar el vector entero y escoger los m elementos indicados.

b) Ordenar los elementos apropiados del vector usando repetidamente el procedimiento de Selección.

c) Invocar m veces al procedimiento que encuentra el k-ésimo elemento (problema 2.15), con los subvectores apropiados.

d) Mediante otro método.

2.11 SOLUCIÓN A LOS PROBLEMAS PROPUESTOS

Solución al Problema 2.1. (☺)

Haciendo uso de las funciones presentadas en la introducción de este capítulo, el procedimiento de ordenación por Ladrillos puede ser implementado como sigue:

```
PROCEDURE Ladrillos(VAR a:vector;prim,ult:CARDINAL);
   VAR j:CARDINAL;
BEGIN
   FOR j:=ult TO prim+1 BY -1 DO
      Intercambia(a,j,PosMaximo(a,prim,j))
   END
END Ladrillos;
```

En cuanto a su complejidad, vamos a estudiar los casos mejor, peor y medio de la llamada al procedimiento *Ladrillos(a,*1*,n)*, que van a coincidir con los mismos casos (mejor, peor y medio) que los de la función *PosMaximo*.

– En el caso mejor:

$$T(n)=\left(\sum_{i=2}^{n}\left(3+1+\left(5+6(i-1)\right)+1+7\right)\right)+3=3n^2+14n-14.$$

– En el caso peor:

$$T(n)=\left(\sum_{i=2}^{n}\left(3+1+\left(5+7(i-1)\right)+1+7\right)\right)+3=\frac{7}{2}n^2+\frac{27}{2}n-14.$$

– En el caso medio:

$$T(n)=\left(\sum_{i=2}^{n}\left(3+1+\left(5+\frac{13}{2}(i-1)\right)+1+7\right)\right)+3=\frac{13}{4}n^2+\frac{55}{4}n-14.$$

En consecuencia, el algoritmo es de complejidad cuadrática.

Solución al Problema 2.2. (☺)

El procedimiento que realiza la ordenación por Sacudidas puede ser implementado colocando alternativamente en su posición definitiva el máximo y el mínimo de un subvector cada vez más pequeño, como muestra el siguiente algoritmo:

```
PROCEDURE Sacudidas(VAR a:vector;prim,ult:CARDINAL);
BEGIN
   WHILE prim<ult DO
      Intercambia(a,ult,PosMaximo(a,prim,ult));
      DEC(ult);
      Intercambia(a,prim,PosMinimo(a,prim,ult));
      INC(prim)
   END
END Sacudidas;
```

Para estudiar su complejidad, vamos a fijarnos en la llamada al procedimiento *Sacudidas(a,*1*,n)*. Sus casos mejor, peor y medio corresponden a los de las funciones que encuentran el máximo y el mínimo del vector. El valor de $T(n)$ resulta ser:

– En el mejor caso:

$$\left(\sum_{k=1}^{n/2}\left(1+\left(1+\left(6(2k-1)+5\right)+1+7\right)+1+\left(1+\left(6(2k-2)+5\right)+1+7\right)+1\right)\right)+1=$$

$$=3n^2+\frac{25}{2}n+1.$$

– En el peor caso:

$$\left(\sum_{k=1}^{n/2}\left(1+\left(1+\left(7(2k-1)+5\right)+1+7\right)+1+\left(1+\left(7(2k-2)+5\right)+1+7\right)+1\right)\right)+1=$$

$$=\frac{7}{2}n^2+12n+1.$$

– En el caso medio:

$$\left(\sum_{k=1}^{n/2}\left(1+\left(1+\left(\frac{13}{2}(2k-1)+5\right)+1+7\right)+1+\left(1+\left(\frac{13}{2}(2k-2)+5\right)+1+7\right)+1\right)\right)+1$$

$$=\frac{13}{4}n^2+\frac{49}{4}n+1.$$

Por consiguiente, podemos concluir que el algoritmo Sacudidas es de complejidad cuadrática.

Solución al Problema 2.3. (☺)

Observando la implementación realizada del procedimiento *Selección* al principio del capítulo podemos ver que en él se intercambian elementos adyacentes independientemente de que sean iguales o no. Podemos realizar la modificación

pedida del algoritmo preguntando antes de cada intercambio si es necesario, obteniendo:

```
PROCEDURE Seleccion2(VAR a:vector;prim,ult:CARDINAL);
   VAR i,j:CARDINAL;
BEGIN
   FOR i:=prim TO ult-1 DO
      j:=PosMinimo(a,i,ult);
      IF a[i]<>a[j] THEN
         Intercambia(a,i,j)
      END
   END
END Seleccion2;
```

Para el cálculo de su complejidad, vamos a considerar sus casos mejor, peor y medio de la llamada al procedimiento *Seleccion2(a,*1,*n)*:

– En el caso mejor el elemento mínimo se encuentra en la primera posición. En consecuencia, la condición es siempre falsa. En este caso:

$$T(n) = \left(\sum_{i=1}^{n-1} \left(3 + 6(n-i) + 5 + 1 + 1 + 3\right) \right) + 3 = 3n^2 + 10n - 10 .$$

– En el caso peor el elemento máximo se encuentra en la última posición, y la condición es siempre verdadera. Así, en este caso:

$$T(n) = \left(\sum_{i=1}^{n-1} \left(3 + 7(n-i) + 5 + 1 + 1 + 3 + 1 + 7\right) \right) + 3 = \frac{7}{2} n^2 + \frac{35}{2} n - 18 .$$

– En el caso medio, el elemento mínimo puede estar de forma equiprobable en cualquiera de las posiciones del subvector en el que lo buscamos, y supondremos además que la condición se verifica la mitad de las veces. Por tanto:

$$T(n) = \left(\sum_{i=1}^{n-1} \left(3 + \frac{13}{2}(n-i) + 5 + 1 + 1 + 3 + \frac{(1+7)}{2} \right) \right) + 3 = \frac{13}{4} n^2 + \frac{55}{4} n - 14 .$$

Nótese que los casos mejor y peor corresponden por tanto a las situaciones en donde el vector a ordenar está ya ordenado o cuando está ordenado en sentido inverso.

Para comparar los dos algoritmos, lo primero es observar que ambos son de complejidad cuadrática y, más aún, que los límites de los cocientes de las funciones $T(n)$ en los tres casos valen exactamente 1, lo que implica que en los tres casos los algoritmos convergen asintóticamente de la misma forma.

En cualquier caso, y cara a afinar más el análisis, podemos restar los tiempos de ejecución de ambos algoritmos en los tres casos y estudiar el signo y la magnitud de la diferencia, obteniendo:

a) En el caso mejor $T_{Seleccion}(n) - T_{Seleccion2}(n) = -4(n-1)$.
b) En el caso peor $T_{Seleccion}(n) - T_{Seleccion2}(n) = -4(n-1)$.
c) En el caso medio $T_{Seleccion}(n) - T_{Seleccion2}(n) = 0$.

Como puede observarse, el algoritmo modificado (*Seleccion2*) es un poco mejor en dos de los tres casos. Sin embargo, las diferencias encontradas no son demasiado significativas.

Solución al Problema 2.4. (☺)

Supongamos que disponemos de una cota dada *M*:

```
CONST M =...;
```

que indica el número de elementos mínimo a partir del cual Quicksort debe invocar al procedimiento *Seleccion*. Es fácil modificar el algoritmo para tener en cuenta este hecho:

```
PROCEDURE Quicksort2(VAR a:vector;prim,ult:CARDINAL);
   VAR l:CARDINAL;
BEGIN
   IF prim<ult THEN
      IF ult-prim<M THEN
         Seleccion(a,prim,ult)
      ELSE
         l:=Pivote(a,a[prim],prim,ult);
         Quicksort2(a,prim,l-1);
         Quicksort2(a,l+1,ult)
      END
   END
END Quicksort2;
```

El procedimiento *Seleccion* es la que implementa el método de ordenación por Selección, expuesta al comienzo de este capítulo.

Primero analiza el caso base dado (que el vector tenga menos de *M* elementos), terminando su ejecución tras ordenar el vector en ese caso. Si el vector a ordenar tiene más de los elementos indicados, el algoritmo continúa como antes.

Dada ahora la constante *M*, la modificación al procedimiento *Mezcla* no plantea tampoco mayor dificultad:

```
PROCEDURE Mezcla2(VAR a,b:vector;prim,ult:CARDINAL);
```

```
(* utiliza el vector b como auxiliar para realizar la mezcla *)
   VAR mitad:CARDINAL;
BEGIN
   IF prim<ult THEN
      IF (ult-prim)<M THEN Seleccion(a,prim,ult)
      ELSE
         mitad:=(prim+ult)DIV 2;
         Mezcla2(a,b,prim,mitad);
         Mezcla2(a,b,mitad+1,ult);
         Combinar(a,b,prim,mitad,mitad+1,ult)
      END
   END
END Mezcla2;
```

Solución al Problema 2.5. (😐)

Tras el estudio que se hizo en el apartado 2.6 sobre la complejidad del método, podemos ya seleccionar dos casos en donde el algoritmo se va a comportar de forma especial:

– Pensemos lo que ocurre cuando todos los elementos del vector a ordenar son iguales. En este caso la función que "empuja" es O(1), con lo cual la complejidad del algoritmo es:

$$(n/2)O(1) + nO(1)O(1) \in O(n)$$

– Cuando los elementos del vector son todos distintos y están ya ordenados en forma creciente, el procedimiento que construye el montículo es de complejidad O(n), y además nos deja un montículo en donde en cada iteración se va a tomar como nueva raíz el mínimo del vector, lo que implica que ésta habrá de ser empujada a lo largo de todo el montículo. Esto hace que su complejidad sea:

$$O(n) + O(n\log n) \in O(n\log n)$$

En resumen, el caso mejor para el algoritmo de ordenación por Montículos ocurre cuando los elementos a ordenar son todos iguales, y el peor cuando son todos distintos y además el vector está ya ordenado.

Es interesante que el lector compare estos casos con los casos desfavorables para Quicksort, estudiados en el problema 2.8.

Solución al Problema 2.6. (😐)

El código de los procedimientos Mezcla y Quicksort en su versión recursiva ya ha sido visto en las secciones correspondientes del presente capítulo. Veamos por tanto la versión iterativa de ambos métodos.

Procedimiento Mezcla Iterativo

Comenzaremos con el procedimiento de ordenación por Mezcla, que admite una versión iterativa que no se basa directamente en una eliminación de la recursión.

La idea es ir dando pasadas por el vector haciendo mezclas. En la primera pasada se mezclan los elementos adyacentes para formar subvectores ordenados de tamaño dos. En la siguiente pasada se mezclan los subvectores adyacentes de tamaño dos para formar subvectores ordenados de tamaño cuatro. Así se continúa hasta obtener un sólo vector ordenado de tamaño *n*. En general van a hacer falta log*n* pasadas para ordenar un vector de *n* elementos. Esto da lugar a la siguiente implementación del algoritmo:

```
PROCEDURE Mezcla_it(VAR a:vector;prim,ult:CARDINAL);
   VAR l,p1,p2,u1,u2:CARDINAL;
BEGIN
   l:=1; (* l es el num. elementos de la mezcla en cada paso *)
   WHILE l<(ult-prim+1) DO
      p1:=prim;
      WHILE p1<ult DO
         u1:=p1+l-1;
         p2:=u1+1;
         u2:=p2+l-1;
         IF p2<=ult THEN
            IF u2>ult THEN u2:=ult END;
            Combinar(a,b,p1,u1,p2,u2)
         END;
         p1:=u2+1
      END;
      l:=2*l
   END
END Mezcla_it;
```

El procedimiento *Combinar* es el mismo que fue implementado para el método *Mezcla*.

Procedimiento Quicksort Iterativo

La versión iterativa de Quicksort se basa en el mecanismo de eliminación de la recursión mediante el uso de una pila que va a contener el trabajo pendiente en cada momento.

Como puede observarse en el algoritmo que presentamos a continuación, vamos a ir dividiendo el vector en dos subvectores (con la función *Pivote*) e insertando en una pila que hemos creado al efecto las posiciones de comienzo y fin de un subvector que más tarde ordenaremos.

Con el otro subvector, en vez de almacenarlo en la pila, lo iremos dividiendo y guardando sus mitades en la pila. Por tratarse de un caso de recursión de cola, podremos eliminar la llamada recursiva mediante un bucle en donde en cada iteración calculamos los valores de las variables para la siguiente.

Una vez acabamos con una mitad, extraemos otro subvector de la pila y repetimos el proceso con él, y así hasta que vaciemos la pila, lo que indicará que ya

hemos ordenado el vector (hemos ordenado cada uno de los "trozos" en los que lo habíamos dividido).

```
FROM PILAS IMPORT pila,Crear,Destruir,Insertar,Extraer,Esvacia;
(* usamos pilas de CARDINAL para almacenar posiciones *)

PROCEDURE Quicksort_it(VAR a:vector;prim,ult:CARDINAL);
   VAR i:CARDINAL; p:pila;
BEGIN
   Crear(p);
   LOOP
      WHILE ult>prim DO
         i:=Pivote(a,a[prim],prim,ult);
         Insertar(p,prim);
         Insertar(p,i-1);
         prim:=i+1
      END;
      IF Esvacia(p) THEN
         Destruir(p);
         RETURN
      END;
      ult:=Extraer(p);
      prim:=Extraer(p)
   END
END Quicksort_it;
```

Obsérvese que es la función *Pivote* la encargada de mover los elementos del vector, y por eso no aparece ninguna instrucción de intercambio en el código que presentamos.

Este algoritmo procesa los mismos subvectores que su versión recursiva, aunque en orden distinto. Además, esta versión iterativa supone un incremento en la eficiencia y admite alguna variante o mejora interesante:

- Por un lado, si en vez de meter en la pila el primero de los dos subvectores metiéramos el de mayor tamaño, conseguiríamos que el tamaño de la pila no fuera nunca mayor que log$n$, puesto que cada entrada en la pila después de la primera debe representar un subvector con menos de la mitad de los elementos de la entrada anterior. Ésta es una mejora interesante con respecto a la versión recursiva de Quicksort, pues en el peor de sus casos la pila de recursión usada puede alcanzar las n entradas (por ejemplo, cuando el vector está ordenado inicialmente). Así se minimiza el riesgo que siempre conlleva el desbordar la memoria existente por un crecimiento desmesurado de la pila de recursión.
- Por otro lado, también podemos mejorar esta versión iterativa para tratar el caso trivial que se obtiene cuando partimos un subvector en dos subvectores de tamaño 1. En este caso se inserta una entrada en la pila para ser extraída inmediatamente y después descartada. Es muy fácil cambiar el algoritmo para

tener en cuenta este hecho, pero quizá sea también mejor tratar los subvectores de tamaño pequeño por separado (como se muestra en el problema 2.4).

Solución al Problema 2.7. (😐)

Respecto a la secuencia de incrementos, lo que parece deseable es que los números que la componen sean coprimos entre sí (dos números se dicen coprimos si su máximo común divisor es 1), pues si no pueden producirse iteraciones en donde no haya intercambios de elementos aunque sí un gran número de comparaciones. Con esto en mente es fácil encontrar el ejemplo pedido. Suponiendo el vector:

[12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

Después de las sucesivas pasadas obtenemos:

		Número de Comparaciones	Número de Intercambios
h=8	[4, 3, 2, 1, 8, 7, 6, 5, 12, 11, 10, 9]	4	4
h=4	[4, 3, 2, 1, 8, 7, 6, 5, 12, 11, 10, 9]	8	0
h=2	[2, 1, 4, 3, 6, 5, 8, 7, 10, 9, 12, 11]	10	4
h=1	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]	11	6

Como puede verse en este ejemplo, el problema es el número de comparaciones frente al número de intercambios. El primero es fijo para esta secuencia (del orden de n^2) pero sin embargo no consigue hacer disminuir el número de intercambios de elementos. Puede verse en el ejemplo cómo los números pares e impares se ordenan por separado, pero ninguno en su posición definitiva hasta el último paso (h=1), en el que lo hacen todos los elementos del vector. Este inconveniente hace que el algoritmo pierda en este caso su competitividad al compararlo con algoritmos aún más sencillos de codificar y depurar como son el de Inserción o Selección, y no digamos si la comparación se realiza frente a procedimientos de complejidad $n\log n$ como pueden ser Quicksort o Mezcla.

Solución al Problema 2.8. (😐)

En la implementación que hemos expuesto del método Quicksort podemos ver que si los elementos son todos iguales la complejidad del algoritmo es de orden cuadrático. En efecto, la partición por el pivote consume un tiempo del orden de O(n), pero el pivote resulta ser siempre el primer elemento, con lo cual una de las dos llamadas a Quicksort tiene tamaño cero y la otra tamaño n–1. Este desequilibrio hace que la complejidad del algoritmo resultante sea de orden O(n^2).

- La primera idea para eliminar ese efecto indeseable es la de comprobar tal condición antes de comenzar a ejecutar el algoritmo, lo que da lugar al siguiente procedimiento:

```
PROCEDURE Quicksort3(VAR a:vector;prim,ult:CARDINAL);
```

```
   VAR l:CARDINAL;
BEGIN
   IF prim<ult THEN
      IF SonTodosIguales(a,prim,ult) THEN RETURN END;
      l:=Pivote(a,a[prim],prim,ult);
      Quicksort3(a,prim,l-1);
      Quicksort3(a,l+1,ult)
   END
END Quicksort3;
```

Naturalmente, la función *SonTodosIguales* devuelve *TRUE* si todos los elementos del subvector *a*[*prim..ult*] son iguales, *FALSE* en otro caso, y es de complejidad O(*n*).

Esta modificación es simple y fácil de implementar, y mantiene en O(*n*log*n*) la complejidad del algoritmo para su caso medio, eliminando el caso conflictivo. Sin embargo, la constante que se obtiene en su tiempo de ejecución hace que la modificación no sea ya rentable frente a otros métodos como Montículos o Mezcla. ¿Existe otra posible solución?

- Una idea alternativa para eliminar este caso conflictivo es menos intuitiva, pero mucho más efectiva. Se basa en modificar la función *Pivote*, de forma que ahora divida el vector inicial en tres partes. Al finalizar la función, los elementos del vector *a* habrán sido permutados de forma que todos los elementos menores que el pivote *p* estén a su izquierda, los elementos iguales a *p* se encuentren todos juntos en el centro, y los elementos mayores que *p* a su derecha. La función devuelve dos posiciones, que son las que marcan las tres partes en las que hemos dividido el vector *a*:

```
PROCEDURE Pivote2(VAR a:vector; p:INTEGER; prim,ult:CARDINAL;
                  VAR k,l:CARDINAL);
(* permuta los elementos de a[prim..ult] y devuelve dos
   posiciones k,l tales que prim-1<=k<l<=ult+1, a[i]<p si
   prim<=i<=k, a[i]=p si k<i<l, y a[i]>p si l<=i<=ult, donde p es
   el pivote y aparece como argumento *)
   VAR m:CARDINAL;
BEGIN
   k:=prim; l:=ult+1;
   (* primero buscamos l *)
   REPEAT INC(k) UNTIL (a[k]>p) OR (k>=ult);
   REPEAT DEC(l) UNTIL (a[l]<=p);
   WHILE k<l DO
      Intercambia(a,k,l);
      REPEAT INC(k) UNTIL (a[k]>p);
      REPEAT DEC(l) UNTIL (a[l]<=p)
   END;
   Intercambia(a,prim,l);
```

```
      INC(l); (* ya tenemos l *)
      (* ahora buscamos el valor de k *)
      k:=prim-1;
      m:=l;
      REPEAT INC(k) UNTIL (a[k]=p) OR (k>=l);
      REPEAT DEC(m) UNTIL (a[m]<>p) OR (m<prim);
      WHILE k<m DO
         Intercambia(a,k,m);
         REPEAT INC(k) UNTIL (a[k]=p);
         REPEAT DEC(m) UNTIL (a[m]<>p);
      END;
   END Pivote2;
```

Una vez disponemos de ese procedimiento, cuya complejidad es lineal, sólo queda modificar ligeramente el código de *Quicksort*:

```
   PROCEDURE Quicksort4(VAR a:vector;prim,ult:CARDINAL);
      VAR k,l:CARDINAL;
   BEGIN
      IF prim<ult THEN
         Pivote2(a,a[prim],prim,ult,k,l);
         Quicksort4(a,prim,k);
         Quicksort4(a,l,ult)
      END
   END Quicksort4;
```

Con esta modificación Quicksort ofrece una complejidad lineal cuando los elementos del vector a ordenar son todos iguales (en este caso se tiene que $k = prim$ y $l = ult$ a la salida del procedimiento *Pivote2*). Sin embargo, ¿cuál ha sido el precio? Para el caso medio hemos "empeorado" el tiempo de ejecución del algoritmo; aunque sigue siendo de complejidad $O(n\log n)$, las constantes que acompañan a los coeficientes de la función de su tiempo de ejecución lo han convertido en un algoritmo más ineficiente que los métodos de ordenación por Montículos y Mezcla en el caso medio.

Ya hemos visto cómo solucionar uno de los casos desfavorables para Quicksort, que es cuando los elementos del vector a ordenar son todos iguales. Sin embargo, la implementación que hemos realizado de Quicksort posee otros casos desfavorables: cuando el vector a ordenar está ya ordenado en orden creciente o decreciente.

En ambos casos se repite lo que ocurría cuando los elementos eran iguales. Se produce un desequilibrio en los subproblemas que resultan de la división del vector, ya que de las dos llamadas recursivas a Quicksort una es invocada con 0 elementos y la otra con $n-1$. Esto lleva a que la complejidad del algoritmo en este caso sea también del orden de $O(n^2)$.

Sin embargo, este problema admite una solución más fácil que el que hemos visto anteriormente. Basta coger como pivote la mediana del vector, en vez de su

primer elemento. Así las llamadas recursivas estarán siempre equilibradas y si la búsqueda de la mediana se hace en tiempo lineal la complejidad del algoritmo se mantendrá del orden de O($n$log$n$). Para ver cómo puede determinarse la mediana de un vector, consúltese el problema 2.15.

Solución al Problema 2.9. (😐)

El procedimiento de ordenación por Residuos para números naturales utiliza una serie de colas auxiliares donde irá clasificando los elementos del vector. Se definen tantas colas como números distintos posea la base de representación usada (que llamaremos B). Por ejemplo, si decidimos utilizar el método para una representación decimal (B = 10), dispondremos de 10 colas (numeradas del 0 al 9); si la representación es binaria dispondremos de 2 colas (0 y 1).

Este método funciona mediante un bucle en el cual en cada iteración considera un dígito distinto de cada uno de los elementos del vector, de derecha a izquierda, insertando el elemento en cuestión en la cola correspondiente al valor de su dígito. Por ejemplo, para una representación binaria, en la primera iteración insertará los elementos en cada una de las dos colas de acuerdo a su dígito menos significativo. En la segunda respecto a su segundo dígito menos significativo, y así sucesivamente.

Tras cada iteración, una vez los elementos hayan sido clasificados utilizando las colas, el método extrae los elementos de las colas en orden, volcándolos de nuevo en el vector. Por tanto, en cada iteración los elementos están en orden respecto al dígito correspondiente a tal iteración. Como utilizamos estructuras de colas (con estrategia de acceso FIFO) se respeta el orden relativo de los elementos de una iteración a otra, lo que hace que el método consiga ordenar finalmente el vector. Naturalmente, el número de iteraciones a realizar va a coincidir con el logaritmo en base B del mayor elemento del vector.

Una posible implementación de este algoritmo para una base B general es el que sigue, donde *MAXB* es una constante que indica el valor máximo para B:

```
TYPE colaitem = RECORD elems:vector; cont:CARDINAL END;

PROCEDURE Residuos(VAR a:vector;B,prim,ult:CARDINAL);
   VAR  i,j,k,h,digito:CARDINAL;
        iter:INTEGER;
        sigo:BOOLEAN;
        colas: ARRAY [0..MAXB-1] OF colaitem;
BEGIN
   iter:=1;                (* iter va acumulando B^1,B^2,B^3,... *)
   REPEAT                  (* para cada uno de los digitos *)
      FOR i:=0 TO B-1 DO   (* inicializamos las colas *)
         colas[i].cont:=1  (* primera posicion libre *)
      END;
      (* clasificacion de los elementos en las colas *)
      sigo:=FALSE;(* indica si quedan numeros con mas digitos *)
      FOR i:=prim TO ult DO
         sigo:=((a[i] DIV iter)>=INTEGER(B)) OR sigo;
```

```
            digito:=(a[i] DIV iter) MOD INTEGER(B); (* num cola *)
            colas[digito].elems[colas[digito].cont]:=a[i];
            INC(colas[digito].cont)    (* inserta a[i] en su cola *)
         END;
         iter:=iter*INTEGER(B);
         j:=prim; (*ahora volcamos las colas sobre el vector *)
         FOR i:=0 TO B-1 DO
            h:=colas[i].cont-1;   (* num de elementos en esa cola *)
            FOR k:=1 TO h DO a[j]:=colas[i].elems[k]; INC(j) END
         END
      UNTIL NOT sigo;
   END Residuos;
```

Como puede observarse, el método realiza un ciclo principal para cada uno de los dígitos, en donde existen tres bucles por cada iteración: primero se inicializan las colas, después se clasifican los elementos en las colas y por último se vuelcan ordenadamente las colas en el vector.

Estudiaremos a continuación la complejidad (temporal) de tal algoritmo. Para ello, si llamamos x al mayor de los elementos del vector a, el número de operaciones elementales que se efectúan en una llamada a *Residuos(a,B,*1*,n)* es:

$$\begin{aligned} T_B(n) &= 1+(5B+2+1+2+(2+5+5+7+3)n+2+1+2+8n+8B)\log_B x = \\ &= 1+(10+30n+13B)\log_B x. \end{aligned}$$

En esta ecuación, la cantidad $\log_B x$ indica el número de dígitos del mayor de los elementos del vector, cuando éste se expresa en base B. Para tratar de cuantificar el valor de la ecuación, podemos acotar el valor de $\log_B x$ por la cantidad $\log_B C$, donde C es la constante que indica el máximo número entero que puede representarse en nuestra máquina. De esta forma, para un ordenador cuya palabra sea de 16 bits tenemos que $\log_2 C = 15$ y $\log_{10} C = 5$, por lo que:

$$T_2(n) \le 1+(36+30n)\log_2 C = 1+(36+30n)\cdot 15 = 450n+541.$$

$$T_{10}(n) \le 1+(140+30n)\log_{10} C = 1+(140+30n)\cdot 5 = 150n+701.$$

Ambos tiempos de ejecución son lineales, y puede observarse que el segundo es menor que el primero. Pero, ¿cuál es el precio que estamos pagando en ambos casos?

La respuesta a esta pregunta viene dada por la complejidad espacial de los algoritmos. Es cierto que ambos consiguen ordenar un vector muy eficientemente, pero también tienen un orden de complejidad espacial lineal. De hecho, el primer algoritmo utiliza del orden de $2n$ elementos auxiliares mientras que el segundo utiliza del orden de $10n$.

Sería posible mejorar esta complejidad espacial mediante el uso de estructuras dinámicas para implementar las colas. Sin embargo este cambio trae consigo un aumento de la complejidad temporal del algoritmo, pues el acceso a estas estructuras es más lento.

Solución al Problema 2.10. (☺)

Para dar respuesta a este problema, es importante hacer notar que la estabilidad de un método depende en gran medida de su implementación. Un ejemplo claro es el método de Selección, en donde la forma de encontrar el elemento mínimo de entre los que quedan por ordenar es clave a la hora de la estabilidad del método. Por eso nos basaremos en las implementaciones de los métodos que se presentan en este capítulo.

Los métodos estables son:

Inserción: Cada elemento se coloca en su lugar, respetando el orden de llegada y por tanto su orden parcial.

Burbuja: Este método es análogo a Inserción y como él, respeta el orden relativo de los elementos.

Mezcla: Como en este método los elementos sólo se intercambian en el procedimiento *Combinar*, es suficiente probar que las mezclas son estables. Pero esto es cierto puesto que la forma de mezclar los subvectores respeta los órdenes relativos de los elementos que los componen.

Los métodos no estables son:

Selección: Aunque la búsqueda del elemento mínimo se hace en el mismo sentido en el que se está ordenando el vector y se escoge siempre el primero de ellos en caso de haber más de uno, este método no resulta ser estable. Por ejemplo, para el vector $[2_A, 2_B, 1, 3]$ (indicamos el orden relativo original de cada elemento mediante el subíndice) tras la primera vuelta, en donde se intercambian los elementos 2_A y 1, obtenemos el vector $[1, 2_B, 2_A, 3]$ que ya no se altera en ninguna pasada.

Incrementos: Este método puede ser visto como una sucesión de aplicaciones del método de Inserción sobre cada uno de los *h*-vectores. Para encontrar un ejemplo que demuestre que el método no es estable, basta coger dos elementos iguales que correspondan a dos *h*-vectores distintos para un valor dado de *h*. Al ordenarse estos dos *h*-vectores por separado, no se respeta el orden parcial de los elementos. Por ejemplo, para una secuencia de incrementos (*h*) que acabe en 4 y 1, podemos tomar el vector $[2_A, 2_B, 1_A, 1_B, 1_C]$. Aplicando el algoritmo *Incrementos* implementado en el apartado 2.8 obtenemos el vector $[1_C, 1_A, 1_B, 2_B, 2_A]$ que, como puede comprobarse, no conserva el orden relativo de los elementos.

Quicksort: Los elementos se intercambian en este método sólo en la función *Pivote*, pero es fácil ver que ésta no respeta su orden relativo, por la forma en que va intercambiando los elementos al hacerlos "saltar" de

un lado del pivote al otro. De esta forma, al ordenar el vector [2_A, 2_B, 1_A, 1_B, 3] obtiene como resultado [1_A, 1_B, 2_B, 2_A, 3].

Montículos: Este método no es estable por la filosofía que tiene de intercambiar el primer elemento del montículo (que es el mayor) con el último. Así, el vector [2_A, 2_B, 1_A, 1_B] que ordenado como [1_B, 1_A, 2_B, 2_A] aplicando el algoritmo implementado en el apartado 2.6.

Ladrillos: Este método no es estable por la forma en la que se escoge el máximo en cada iteración. La función *PosMaximo* elige el elemento máximo con el menor subíndice, y el algoritmo *Ladrillos* lo coloca al final: realmente este algoritmo está invirtiendo el orden relativo inicial de los elementos. Como ejemplo, el vector [2_A, 2_B, 1_A, 1_B, 3] queda ordenado como [1_B, 1_A, 2_B, 2_A, 3].

Sacudidas: Este método se encuentra en la misma circunstancia que Ladrillos, por el mismo motivo. El resultado que obtiene tras ordenar el vector [2_A, 2_B, 2_C, 1_A, 1_B] es [1_B, 1_A, 2_C, 2_B, 2_A].

Por último, para algunos de los métodos no estables es fácil sugerir modificaciones que los conviertan en estables, mientras que para otros no es posible debido a la filosofía general de funcionamiento de cada uno de ellos. Por ejemplo, en los métodos de ordenación por Incrementos y Montículos no se puede mantener el orden relativo de los elementos sin alterar el diseño de los métodos.

Sin embargo, sí es fácil conseguir versiones estables de los demás. Por ejemplo, para el método de Selección basta con ir copiando el vector a otro, de forma que en cada pasada se copie el elemento mínimo:

```
PROCEDURE Seleccion3(VAR a:vector;prim,ult:CARDINAL):CARDINAL;
   VAR i,j:CARDINAL; b:vector;
BEGIN
   FOR i:=prim TO ult DO b[i]:=a[i] END; (* copiamos a en b *)
   FOR i:=prim TO ult DO
      j:=PosMinimo(b,prim,ult);
      a[i]:=b[j]
      b[j]:=MAX(INTEGER); (* para no tenerlo en cuenta mas *)
   END;
END Seleccion3;
```

Para *Ladrillos* y *Sacudidas*, no sólo basta con que utilicen una función de selección del máximo que escoja el elemento mayor que aparezca en última posición, sino que también tendrían que incorporar un cambio similar al sugerido para el método de Selección.

Respecto a *Quicksort*, la única modificación a realizar es en la función *Pivote*, de forma que respete el orden parcial de los elementos cuando los hace “saltar” de un lado del pivote a otro.

Solución al Problema 2.11. (✍)

El procedimiento de Inserción modificado puede ser implementado como sigue:

```
PROCEDURE Insercion2(VAR a:vector;prim,ult:CARDINAL);
   VAR i,j,k:CARDINAL; x:INTEGER;
BEGIN
   FOR i:=prim+1 TO ult DO
      x:=a[i]; k:=Posicion(a,prim,i-1,x);
      FOR j:=i-1 TO k+1 BY -1 DO
         a[j+1]:=a[j]
      END;
      a[k]:=x
   END
END Insercion2;
```

Nos apoyamos en una función que calcula la posición en donde hemos de insertar un nuevo elemento dentro de un subvector previamente ordenado, utilizando búsqueda binaria:

```
PROCEDURE
Posicion(a:vector;prim,ult:CARDINAL;x:INTEGER):CARDINAL;
   VAR mitad:CARDINAL;
BEGIN
   WHILE (prim<=ult) DO
      mitad:=(prim+ult) DIV 2;
      IF x=a[mitad] THEN RETURN mitad
      ELSIF x<a[mitad] THEN ult:=mitad-1
      ELSE prim:=mitad+1
      END
   END;
   RETURN mitad
END Posicion;
```

Para calcular la complejidad del nuevo procedimiento de Inserción, necesitamos conocer primero el tiempo de ejecución de la función *Posicion*. Pero éste es conocido, pues es igual al de la función *BuscBIt* del problema 1.16 (Búsqueda binaria implementada de forma iterativa). Con esto, la complejidad de *Insercion2* es como sigue:

– Para el estudio del caso mejor, tenemos que éste puede ocurrir cuando (*a*) se da el caso mejor de la función *Posicion*, con lo que el bucle siguiente se efectúa para la mitad de los elementos considerados. O bien cuando (*b*) el bucle se efectúa una vez por ser $k = i - 1$; que es también el peor caso de *Posicion*. Calculando el tiempo de ejecución de cada una de las opciones ($T_a(n)$ y $T_b(n)$)*:*

$$T_a(n) = \left(\sum_{i=1}^{n-1} (11 + 9 + 6(i-1)/2) \right) + 2 = \frac{3}{2} n^2 + \frac{31}{2} n - 15.$$

$$T_b(n) = \left(\sum_{i=1}^{n-1} (11 + 10 \log i + 4 + 0) \right) + 2 = 15n - 13 + 10 \sum_{i=1}^{n-1} \log i.$$

Como $T_a(n) > T_b(n)$ para valores grandes de n, el mejor de los casos de *Insercion2* corresponde a la opción (*b*), que se produce cuando el vector está previamente ordenado.

– El caso peor ocurre cuando se dan simultáneamente los casos peores de la función *Posicion* y del bucle *WHILE* (y esto ocurre cuando el vector está ordenado en forma inversa a como queremos ordenarlo). Llamando (*p*) a esta opción obtenemos:

$$T_p(n) = \left(\sum_{i=1}^{n-1} (11 + 10 \log i + 4 + 6(i-1)) \right) + 2 = 3n^2 + 6n - 7 + 10 \sum_{i=1}^{n-1} \log i.$$

– Por último, el caso medio ocurre cuando se da el caso medio de la función *Posicion* y el bucle *WHILE* se ejecuta, en media, $i/2$ veces:

$$T_{1/2}(n) = \left(\sum_{i=1}^{n-1} \left(11 + 8 \frac{i \log i - i + 1}{i+1} + 6 + 6(i-1)/2 \right) \right) + 2.$$

Respecto a los órdenes de complejidad de tales funciones, todos son cuadráticos menos en la opción (*b*), de complejidad O($n$log$n$) por ser de este orden la expresión $\sum_{i=1}^{n-1} \log i$.

Para valorar los dos algoritmos necesitamos comparar sus tiempos de ejecución siempre que estos ocurran bajo las mismas circunstancias. Pero así sucede en este algoritmo: el caso peor de ambos métodos ocurre cuando el vector está ordenado en orden inverso al deseado, el caso medio cuando cualquier permutación del vector es inicialmente equiprobable, y el caso mejor cuando el vector a ordenar está ya ordenado. Podemos comparar entonces sus tiempos de ejecución y obtenemos:

- En el caso mejor $T_{Insercion}(n) < T_{Insercion2}(n)$, siendo incluso el segundo de un orden de complejidad mayor al primero (n frente a $n$log$n$).
- En el caso peor $T_{Insercion}(n) > T_{Insercion2}(n)$ para $n > 15$, aunque ambos algoritmos son de complejidad cuadrática.
- En el caso medio $T_{Insercion}(n) > T_{Insercion2}(n)$ para $n > 23$, aunque ambos algoritmos son de complejidad cuadrática.

Como puede observarse, el algoritmo modificado (*Insercion2*) es más eficiente (respecto a su tiempo de ejecución) que el algoritmo inicial para los casos peor y medio. Esto era de esperar porque, aunque no se rebaja el número de intercambios que dejan al nuevo elemento en su posición final, sí se consigue disminuir el número de comparaciones necesario para buscar tal posición.

Sin embargo, en el caso mejor estamos introduciendo una serie de comparaciones adicionales por la forma en la que buscamos. Esto hace aumentar la complejidad del algoritmo en un orden de magnitud, lo cual no es despreciable.

Resumiendo, con esta modificación conseguimos una mejora en la mayoría de los casos respecto al tiempo de ejecución. Pero obsérvese que esto siempre lleva asociado un precio. La simplicidad del algoritmo original se ve comprometida en esta nueva versión, lo que suele conllevar problemas de depuración, verificación y mantenimiento del nuevo código. Como siempre, dejamos en manos del usuario del algoritmo la decisión a tomar, pues ésta va a depender mucho de los factores particulares de su sistema o aplicación.

Solución al Problema 2.12. (😐)

El procedimiento *Rastreo* puede ser implementado como sigue:

```
PROCEDURE Rastreo (VAR a:vector;prim,ult:CARDINAL);
   VAR i:CARDINAL;
BEGIN
   IF prim>=ult THEN RETURN END;
   i:=prim;
   LOOP
      WHILE a[i]<=a[i+1] DO
         INC(i); IF i=ult THEN RETURN END
      END;
      Intercambia(a,i,i+1);
      WHILE (i>prim) AND (a[i-1]>a[i]) DO
         Intercambia(a,i,i-1);
         DEC(i)
      END
   END
END Rastreo;
```

En cuanto a su complejidad, vamos a estudiar los casos mejor, peor y medio de la llamada al procedimiento *Rastreo(a,*1*,n)*.

– El caso mejor ocurre cuando el algoritmo nunca tiene que ir "hacia atrás", limitándose a ejecutar sólo el primer ciclo *WHILE*, resultando:

$$T(n) = 1 + 1 + \left(\sum_{i=1}^{n-1} (1 + 1 + 6) \right) + 1 = 8n - 5 .$$

Obsérvese que este caso mejor va a coincidir cuando el vector está inicialmente ordenado y por tanto sólo se efectúa el primer bucle.

– En el caso peor los dos bucles *WHILE* se ejecutan siempre en su totalidad. Así, obtenemos:

$$T(n) = 1+1+\sum_{i=1}^{n-1}\left(\left(\sum_{k=1}^{i-1}(1+1+6)\right)+6+7+2+\left(\sum_{k=1}^{i-1}(1+7+1+1+6)\right)+6\right)$$
$$= 12n^2 - 15n + 5\,.$$

Esta situación sucede cuando el vector está inicialmente ordenado en forma inversa a como queremos ordenarlo.

– En el caso medio, cualquier ordenación de los elementos es igualmente probable, y por tanto el número de veces que se repite cada uno de los bucles *WHILE* es:

$$T(n) = 1+1+\sum_{i=1}^{n-1}\left(\frac{1}{2}\left(\sum_{k=1}^{i-1}(1+1+6)\right)+6+7+2+\frac{1}{2}\left(\sum_{k=1}^{i-1}(1+7+1+1+6)\right)+6\right)$$
$$= 6n^2 + 3n - 7\,.$$

Solución al Problema 2.13. (😐)

De forma análoga al procedimiento *Mezcla* expuesto al principio del capítulo, el procedimiento pedido puede ser implementado como sigue:

```
PROCEDURE MezclaTres(VAR a,b:vector;prim,ult:CARDINAL);
(* utiliza el vector b como auxiliar para realizar la mezcla *)
  VAR terc1,terc2:CARDINAL;
BEGIN
  IF ult<=prim THEN RETURN END;
  IF ult-prim>=2 THEN
    (* primero divide el vector en tres subvectores:
       a[prim..terc1], a[terc1+1..terc2] y a[terc2+1..ult] *)
    terc1:=prim-1+(ult-prim+1)DIV 3;
    terc2:=terc1+(ult-prim+2)DIV 3;
    MezclaTres(a,b,prim,terc1);
    MezclaTres(a,b,terc1+1,terc2);
    MezclaTres(a,b,terc2+1,ult);
    (* y luego los mezcla *)
    CombinarTres(a,b,prim,terc1,terc1+1,terc2,terc2+1,ult)
  ELSE  (* caso base *)
    IF a[prim]>a[ult] THEN Intercambia(a,prim,ult) END
  END
END MezclaTres;
```

En cuanto al procedimiento que realiza la mezcla, es una versión análoga a la del algoritmo original pero esta vez combinando tres subvectores consecutivos y ya ordenados.

Primero vuelca los elementos a ordenar en el vector auxiliar y luego, utilizando un índice para cada subvector, va escogiendo el menor de los elementos apuntados por esos índices e incrementando el índice correspondiente hasta alcanzar el final.

Nótese que el algoritmo utiliza el hecho de que los subvectores están ya ordenados y que además son consecutivos.

Su implementación es como sigue:

```
PROCEDURE CombinarTres(VAR a,b:vector;
                       p1,u1,p2,u2,p3,u3:CARDINAL);
(* mezcla ordenadamente los subvectores a[p1..u1], a[p2..u2] y
   a[p3..u3] suponiendo que estos estan ya ordenados y que son
   consecutivos (es decir, p2=u1+1 y p3=u2+1), y utilizando el
   vector b como auxiliar. *)
   VAR i1,i2,i3,k:CARDINAL;
BEGIN
   FOR k:=p1 TO u3 DO b[k]:=a[k] END;
   i1:=p1;i2:=p2; i3:=p3;   (* cada indice se encarga de un
subvector *)
   FOR k:=p1 TO u3 DO
      CASE NumMin(b[i1],b[i2],b[i3]) OF
      |1:a[k]:=b[i1];
         IF i1<u1 THEN INC(i1) ELSE b[i1]:=MAX(INTEGER) END
      |2: a[k]:=b[i2];
         IF i2<u2 THEN INC(i2) ELSE b[i2]:=MAX(INTEGER) END
      |3: a[k]:=b[i3];
         IF i3<u3 THEN INC(i3) ELSE b[i3]:=MAX(INTEGER) END
      END
   END
END CombinarTres;
```

Por otro lado, para escoger cuál de los elementos es menor utiliza una función auxiliar que calcula el orden relativo del mínimo de tres elementos:

```
PROCEDURE NumMin(a,b,c:INTEGER):CARDINAL;
BEGIN
   IF (a<=b) AND (a<=c) THEN RETURN 1 END; (* a es el menor *)
   IF (b<=a) AND (b<=c) THEN RETURN 2 END; (* b es el menor *)
   RETURN 3; (* c es el menor *)
END NumMin;
```

Para calcular la complejidad de este algoritmo, determinaremos primero su tiempo de ejecución en función del número de operaciones elementales que realiza dependiendo del tamaño de la entrada (número de elementos a ordenar).

Siguiendo el mismo método que hemos utilizado en los problemas del primer capítulo, se llega a que el tiempo de ejecución de *MezclaTres(a,b,*1*,n)* puede expresarse mediante una ecuación en recurrencia:

$$T_3(n) = 3T_3(n/3) + 21n + 29$$

con la condición inicial $T_3(1) = 2$. Ésta es una ecuación en recurrencia no homogénea cuya ecuación característica es $(x-3)^2(x-1) = 0$, lo que permite expresar $T_3(n)$ como:

$$T_3(n) = c_1 n + c_2 n\log_3 n + c_3.$$

El cálculo de las constantes puede hacerse a partir de la condición inicial, lo que nos lleva a la expresión final:

$$T_3(n) = 21n\log_3 n + (33/2)n - (29/2) \in \Theta(n\log n).$$

Comparemos a continuación este procedimiento con el de Mezcla clásico. Para eso necesitamos calcular su tiempo de ejecución, y nos apoyaremos en la implementación que hemos dado al principio de este capítulo.

El tiempo de ejecución de *Mezcla(a,b,*1*,n)* ya lo calculamos cuando expusimos tal método, obteniendo la siguiente expresión:

$$T_2(n) = 16n\log n + 18n - 17 \in \Theta(n\log n).$$

Como podemos observar, ambos métodos son del mismo orden de complejidad. Ahora bien, nos planteamos si uno de ellos es mejor que el otro, en qué casos y cuánto mejor.

Para responder a estas preguntas tendremos que comparar las funciones que definen los tiempos de ejecución de ambos algoritmos. Para eso definimos:

$$T_d(n) = T_2(n) - T_3(n).$$

Puede comprobarse que $T_d(n) > 0$ para todo $n > 2$. Esto implica que el algoritmo *MezclaTres* se comporta mejor que el de Mezcla original, aunque siempre teniendo en cuenta que ambos son del mismo orden de complejidad.

$T_d(n)$ nos ofrece una medida absoluta del grado de mejora que supone un método frente a otro. Por otro lado, la expresión

$$\lim_{n\to\infty} \frac{T_2(n)}{T_3(n)} = 1.20759$$

nos indica que, para valores grandes de n, el método de Mezcla clásico es hasta un 20% peor que el segundo algoritmo.

Sin embargo, ¿cuál es el precio que hemos pagado para conseguir esta mejora? Sin lugar a dudas este precio se refleja en el código resultante, más complicado, difícil de diseñar y mantener. Este aspecto, como ya hemos mencionado anteriormente, debe tenerse siempre en cuenta y explica la razón por la que se utiliza normalmente el procedimiento que parte en dos a pesar de saber que el que parte en tres es un poco más eficiente (en tiempo de ejecución).

Tras el resultado obtenido en este problema podemos plantearnos qué ocurriría si partiésemos el vector en cuatro partes. ¿Volveríamos a obtener una mejora? ¿Y en cinco partes? ¿Existe un número óptimo de partes en las que dividir el vector? Aunque no entraremos en detalle para responder estas cuestiones, sí queremos dar una idea intuitiva de lo que ocurre en estos casos.

En primer lugar, hemos visto que el tiempo de ejecución del método clásico es de la forma $T_2(n) = an\log_2 n + b$, y el de *MezclaTres* puede expresarse como:

$$T_3(n) = cn\log_3 n + d = (c/\log 3)n\log_2 n + d,$$

donde a, b, c y d son constantes. En general, el tiempo de ejecución del método basado en dividir el vector en k partes va a ser

$$T_k(n) = skn\log_k n + t = s(k/\log k)n\log_2 n + d,$$

siendo s y t constantes y en donde vamos a poder conseguir que el valor de s sea muy similar al de c por la estructura del algoritmo. La razón por la que se introduce la k como constante multiplicativa es debido a la función que ha de calcular el mínimo de los elementos del vector auxiliar b. Para el caso de Mezcla clásico es el mínimo de dos elementos; para *MezclaTres* han de considerarse tres elementos, y para "*MezclaK*" es necesario encontrar el mínimo de k elementos no necesariamente ordenados, procedimiento de orden de complejidad lineal –O(k)–.

Esto nos lleva a que todos los métodos van a ser del mismo orden de complejidad. Sin embargo, las funciones $T_k(n)$ son cada vez mayores conforme k crece, puesto que

$$\lim_{k\to\infty} \frac{k}{\log_2 k} = \infty .$$

En resumidas cuentas, aunque inicialmente $T_2(n)$ fuera mayor que $T_3(n)$ esto no va a ocurrir siempre, pues para valores grandes de k tenemos que $T_k(n) < T_{k+1}(n)$.

El punto donde se alcanza el mínimo de tal sucesión de funciones va a depender de la implementación que se realice del procedimiento general, pero si se sigue un esquema similar al que nosotros hemos implementado aquí, la constante s resulta ser del orden de ocho, alcanzándose el mínimo para $k = 3$.

Entonces, ¿por qué no se enseña este método a los alumnos en vez del Mezcla clásico? La respuesta viene una vez más dada por la evaluación de la ganancia que se obtiene (en cuanto a tiempo de ejecución) frente a las desventajas de este nuevo método respecto a la dificultad y mantenimiento del código obtenido. La naturalidad y claridad del primero lo hacen preferible.

Solución al Problema 2.14. (☺)

Calculando el número de operaciones elementales que realiza el algoritmo obtenemos la ecuación en recurrencia:

$$T(n) = 22 + 3T(2n/3).$$

Esta ecuación es fácil de resolver si hacemos el cambio $n = (3/2)^k$, mediante el cual obtenemos

$$t_k = 22 + 3t_{k-1},$$

ecuación en recurrencia no homogénea con ecuación característica $(x-3)(x-1) = 0$, y por tanto la solución es:

$$t_k = c_1 3^k + c_2.$$

Deshaciendo el cambio, obtenemos finalmente:

$$T(n) = c_1 3^{\log_{3/2} n} + c_2 = c_1 n^{\log_{3/2} 3} + c_2.$$

Para el cálculo de las constantes, tomaremos dos condiciones iniciales: $T(1) = 6$ y $T(2) = 13$. Con ambas es fácil ver que $c_2 = 6$ y $c_1 = 1.07017$. Como $c_1 > 0$ podemos afirmar que:

$$T(n) \in \Theta(n^{\log_{3/2} 3}).$$

Ahora bien, $\log_{3/2}3 = 2.7095113$, con lo cual este método resulta ser de un orden de complejidad muy superior al del resto de los métodos de ordenación vistos en el presente capítulo y por tanto no rentable frente a ellos.

Solución al Problema 2.15. (😐)

Este problema es una generalización del que intenta encontrar la mediana de un vector dado (para $k = (n+1)\div 2$), conocido también como el problema de la *Selección*. Sin embargo hemos preferido referirnos a él como el problema del *k-ésimo elemento* para no confundirlo con el algoritmo de ordenación del mismo nombre.

- Efectivamente, una primera idea consiste en ordenar el vector $a[1..n]$ por algún método eficiente y luego escoger el elemento $a[k]$. Sabemos ya que este procedimiento es de complejidad $O(n\log n)$.
- Si decidiéramos modificar el procedimiento de Selección de forma que parase cuando hubiera ordenado hasta la posición k conseguiríamos cierta mejora para algunos casos, pues este algoritmo sería de complejidad $O(nk)$.
- Otra idea interesante es la de utilizar el método de ordenación por montículos, modificándolo como en el caso anterior para que pare cuando tenga ordenado hasta la posición k. Así logramos hacerlo mejor para algunos valores de k, pues este procedimiento es de complejidad $(n-k)\log n$ (ya que el método de montículos ordena de atrás hacia adelante). También podemos usar una variante en donde los montículos están ordenados de menor a mayor. Con esto conseguimos un procedimiento de complejidad $O(k\log n)$.
- ¿Puede usarse una modificación de Quicksort para resolver este problema? Observando cómo funciona este método, nos damos cuenta que la función

Pivote nos puede ayudar: tras su ejecución ha modificado el vector de forma que los elementos anteriores a la posición *l* que nos devuelve son todos menores o iguales que a[*l*], y los posteriores a *l* son mayores que a[*l*]. La idea es pues invocar repetidamente a la función *Pivote* hasta que la posición *l* coincida con la que buscamos (*k*). Con este procedimiento no es necesario realizar ninguna ordenación explícita:

```
PROCEDURE Kesimo(VAR a:vector;prim,ult,k:CARDINAL):INTEGER;
   VAR l:CARDINAL;
BEGIN
   IF prim<ult THEN
      l:=Pivote(a,a[prim],prim,ult);
      IF l>(prim+k-1) THEN RETURN Kesimo(a,prim,l-1,k) END;
      IF l<(prim+k-1) THEN RETURN Kesimo(a,l+1,ult,k-l+prim-1)
END;
      RETURN a[l]
   ELSE
      RETURN a[ult]
   END
END Kesimo;
```

Con este método conseguimos resolver el problema, y tiene un orden de complejidad lineal para la mayoría de los casos, lo cual es mejor de lo conseguido hasta ahora.

Sin embargo, por estar basado en Quicksort, este método va a heredar sus casos conflictivos: cuando el vector está ya ordenado o cuando todos los elementos del vector son iguales. En ambos casos nuestro procedimiento resulta ineficiente, pues su complejidad es cuadrática.

¿Podemos corregir de alguna forma estos casos extremos? Nuestra siguiente idea es obvia tras haber realizado el problema 2.8. Podríamos usar la función *Pivote2* para eliminar el caso en que todos los elementos son iguales, con lo que obtendríamos:

```
PROCEDURE Kesimo2(VAR a:vector;prim,ult,k:CARDINAL):INTEGER;
   VAR i,d:CARDINAL;
BEGIN
   IF prim<ult THEN
      Pivote2(a,a[prim],prim,ult,i,d);
      IF (prim+k-1)<i THEN RETURN Kesimo2(a,prim,i-1,k) END;
      IF d<=(prim+k-1) THEN RETURN Kesimo2(a,d,ult,k-d+prim) END;
      RETURN a[i]
   ELSE
      RETURN a[ult]
   END
END Kesimo2;
```

Sin embargo, esta función deja pendiente el caso en que el vector esté ordenado (de forma creciente o decreciente), pues escogemos como pivote el

primer elemento del vector. Lo mejor sería escoger la mediana, que es el elemento que queda justo en medio, asegurándonos así que en cada iteración dividimos por dos los elementos a considerar; esto conllevaría un tiempo de ejecución de orden lineal.

El problema es que esto parece contradictorio, puesto que hemos reducido el problema de calcular la mediana a ¡calcular la mediana! ¿Cómo salimos ahora de esta situación?

Una posible solución puede encontrarse en [BRA97], y se basa en una técnica bastante general, y por eso hemos querido incluir aquí este tipo de problemas. Basta con encontrar una función que calcule una mediana "aproximada" y a partir de ahí utilizar el algoritmo *Kesimo* con $k = (n+1) \div 2$.

```
PROCEDURE Kesimo3(VAR a:vector;prim,ult,k:CARDINAL):INTEGER;
   VAR i,d:CARDINAL; pm:INTEGER; (* pseudo_mediana *)
BEGIN
   IF prim<ult THEN
      pm:=CasiMediana(a,prim,ult);
      Pivote2(a,pm,prim,ult,i,d);
      IF (prim+k-1)<i THEN RETURN Kesimo3(a,prim,i-1,k) END;
      IF d<=(prim+k-1) THEN RETURN Kesimo3(a,d,ult,k-d+prim) END;
      RETURN a[i]
   ELSE
      RETURN a[ult]
   END
END Kesimo3;
```

La función *CasiMediana* es la que calcula la mediana aproximada, y se basa en otra función auxiliar, *Medianade5*, que determina la mediana de un vector de 5 elementos:

```
PROCEDURE CasiMediana(VAR a:vector; prim,ult:CARDINAL):INTEGER;
(* calcula una mediana aproximada del vector a[prim..ult] *)
   VAR n,i:CARDINAL; b:vector;
BEGIN
   n:=ult-prim+1;
   IF n<=5 THEN
      RETURN Medianade5(a,prim,ult)
   END;
   n:=n DIV 5;
   FOR i:=1 TO n DO
      b[i]:=Medianade5(a,5*i-4+prim-1,5*i+prim-1)
   END;
   RETURN Kesimo3(b,1,n,(n+1)DIV 2)
END CasiMediana;

PROCEDURE Medianade5(VAR a:vector; prim,ult:CARDINAL):INTEGER;
```

```
(* calcula la mediana de un vector de hasta 5 elementos (es
   decir, ult<prim+5) *)
   VAR i,n:CARDINAL; b:vector; (* para no modificar el vector *)
BEGIN
   n:=ult-prim+1; (* numero de elementos *)
   FOR i:=1 TO n DO
      b[i]:=a[prim+i-1]
   END;
   FOR i:=1 TO (n+1) DIV 2 DO
      Intercambia(b,i,PosMinimo(b,i,n))
   END;
   RETURN b[(n+1) DIV 2];
END Medianade5;
```

El procedimiento es capaz de calcular el *k*-ésimo elemento de un vector en tiempo lineal respecto al tamaño de la entrada, aunque sin embargo vuelve a ocurrir aquí lo que comentamos anteriormente. Para conseguir un tiempo lineal hemos tenido que pagar un precio, que en este caso es que la constante multiplicativa del tiempo de ejecución de este método se ha visto duplicada.

La decisión de si merece la pena pagar ese precio sólo para cubrir los dos casos especiales del algoritmo la dejamos al usuario del mismo.

Solución al Problema 2.16. (😐)

a) La opción de ordenar el vector y escoger los *m* primeros elementos es de complejidad O(*n*log*n*) si escogemos uno de los métodos de ordenación de este orden.
b) Si usamos repetidamente el procedimiento de ordenación por Selección conseguimos una mejora en la complejidad para valores pequeños de *m*: el método es de orden O(*mn*).
c) Como el procedimiento que encuentra el *k*-ésimo elemento es de complejidad lineal, invocándolo *m* veces obtenemos también un método de orden O(*mn*).

Estudiemos por tanto otras opciones.

- Pensando en modificar alguno de los métodos de ordenación ya conocidos, podemos pensar en el método de ordenación por montículos. Es fácil modificar este método para conseguir un procedimiento que encuentre los *m* elementos más pequeños en tiempo O((*n*–*m*)log*n*), o bien en tiempo O(*m*log*n*) si utilizamos montículos invertidos, es decir, en donde en la raíz se encuentra el menor elemento.
- Pero es al pensar en una modificación de Quicksort cuando conseguimos un algoritmo basado en su estrategia y con mejor tiempo. Queremos buscar los *m* elementos menores de un vector, y podemos utilizar la función *Pivote* para esto.

Así, nuestro propósito es ir dejando a la izquierda del pivote los elementos menores que él hasta que consigamos que nuestro pivote sea el *m*-ésimo elemento:

```
PROCEDURE Menores(VAR a:vector;prim,ult,m:CARDINAL);
   VAR l:CARDINAL;
BEGIN
   IF prim<ult THEN
      l:=Pivote(a,a[prim],prim,ult);
      IF l>(prim+m-1) THEN Menores(a,prim,l-1,m)
      ELSIF l<(prim+m-1) THEN Menores(a,l+1,ult,m-l+prim-1)
      END
   END
END Menores;
```

Obsérvese que en las *m* primeras posiciones del vector se encuentran todos los elementos pedidos, aunque no necesariamente ordenados, por la forma como trabaja la función *Pivote*. Este procedimiento es, como en el caso de buscar el *k*-ésimo elemento, de complejidad lineal.

Como mencionábamos en el problema anterior, esta solución es de orden lineal menos en dos casos, ambos heredados del método original de Quicksort. Cuando el vector está ordenado de antemano y cuando todos los elementos del vector son iguales. Ante estas dos circunstancias nuestro procedimiento ofrece un comportamiento de complejidad cuadrática. Al igual que entonces, podemos aplicar los mecanismos vistos en el problema anterior que permiten al método tener en cuenta estos dos casos, aunque pagando cierto precio en los demás.

Solución al Problema 2.17. (😐)

Volvemos a encontrarnos aquí con un caso similar a los tratados en los dos problemas anteriores. Las primeras opciones ya son conocidas:

a) La opción de ordenar el vector y escoger los *m* elementos pedidos es de complejidad $n\log n$ si escogemos uno de los métodos de ordenación de este orden.
b) Si usamos repetidamente el procedimiento de ordenación por Selección conseguimos una complejidad de orden $O((m+n/2)n)$, siendo este caso peor que la primera opción (por ser cuadrática).
c) Como el procedimiento que encuentra el *k*-ésimo elemento es de complejidad lineal, invocándolo *m* veces obtenemos un método de orden $O(mn)$.
d) Usando una modificación del método de Montículos obtendríamos un procedimiento de orden $O((m+n/2)\log n)$.

¿Cómo podríamos utilizar la estrategia de Quicksort para conseguir un método mejor?

- La primera idea consiste en acotar, usando un procedimiento análogo al del problema anterior, los elementos $n \div 2$ y $(n \div 2)+m-1$. Esto nos dejaría en medio de ambos los elementos buscados.
- Sin embargo, existe un método mejor, cuya idea consiste en localizar el elemento $n \div 2$ mediante el procedimiento del k-ésimo (problema 2.15), y luego utilizar el procedimiento *Menores* del problema anterior (2.16) sobre el subvector [$n \div 2$..*ult*]:

```
PROCEDURE Medios(VAR a:vector;prim,ult,l,m);
  VAR x:INTEGER;
BEGIN
  x:=Kesimo(a,prim,ult,l);
  Menores(a,l,ult,m)
END Medios;
```

El procedimiento que hemos implementado aquí es un poco más general, puesto que busca los elementos que ocuparían las posiciones l, $l+1$,...,m, con $l < m$. Basta invocarlo con $l = (n \div 2)$ para obtener el método pedido. Y como puede observarse, la complejidad de este método es lineal, por serlo cada uno de los procedimientos que lo componen.

Capítulo 3

DIVIDE Y VENCERÁS

3.1 INTRODUCCIÓN

El término Divide y Vencerás en su acepción más amplia es algo más que una técnica de diseño de algoritmos. De hecho, suele ser considerada una filosofía general para resolver problemas y de aquí que su nombre no sólo forme parte del vocabulario informático, sino que también se utiliza en muchos otros ámbitos.

En nuestro contexto, Divide y Vencerás es una técnica de diseño de algoritmos que consiste en resolver un problema a partir de la solución de subproblemas del mismo tipo, pero de menor tamaño. Si los subproblemas son todavía relativamente grandes se aplicará de nuevo esta técnica hasta alcanzar subproblemas lo suficientemente pequeños para ser solucionados directamente. Ello naturalmente sugiere el uso de la recursión en las implementaciones de estos algoritmos.

La resolución de un problema mediante esta técnica consta fundamentalmente de los siguientes pasos:

1. En primer lugar ha de plantearse el problema de forma que pueda ser descompuesto en k subproblemas del mismo tipo, pero de menor tamaño. Es decir, si el tamaño de la entrada es n, hemos de conseguir dividir el problema en k subproblemas (donde $1 \leq k \leq n$), cada uno con una entrada de tamaño n_k y donde $0 \leq n_k < n$. A esta tarea se le conoce como *división*.
2. En segundo lugar han de resolverse independientemente todos los subproblemas, bien directamente si son elementales o bien de forma recursiva. El hecho de que el tamaño de los subproblemas sea estrictamente menor que el tamaño original del problema nos garantiza la convergencia hacia los casos elementales, también denominados casos *base*.
3. Por último, *combinar* las soluciones obtenidas en el paso anterior para construir la solución del problema original.

El funcionamiento de los algoritmos que siguen la técnica de Divide y Vencerás descrita anteriormente se refleja en el esquema general que presentamos a continuación:

```
PROCEDURE DyV(x:TipoProblema):TipoSolucion;
   VAR i,k,:CARDINAL;
       s:TipoSolucion;
       subproblemas: ARRAY OF TipoProblema;
       subsoluciones:ARRAY OF TipoSolucion;
BEGIN
   IF EsCasobase(x) THEN
      s:=ResuelveCasoBase(x)
   ELSE
      k:=Divide(x,subproblemas);
      FOR i:=1 TO k DO
         subsoluciones[i]:=DyV(subproblemas[i])
      END;
      s:=Combina(subsoluciones)
   END;
   RETURN s
END DyV;
```

Hemos de hacer unas apreciaciones en este esquema sobre el procedimiento *Divide*, sobre el número *k* que representa el número de subproblemas, y sobre el tamaño de los subproblemas, ya que de todo ello va a depender la eficiencia del algoritmo resultante.

En primer lugar, el número *k* debe ser pequeño e independiente de una entrada determinada. En el caso particular de los algoritmos Divide y Vencerás que contienen sólo una llamada recursiva, es decir $k = 1$, hablaremos de algoritmos de *simplificación*. Tal es el caso del algoritmo recursivo que resuelve el cálculo del factorial de un número, que sencillamente reduce el problema a otro subproblema del mismo tipo de tamaño más pequeño. También son algoritmos de simplificación el de búsqueda binaria en un vector o el que resuelve el problema del *k*-ésimo elemento.

La ventaja de los algoritmos de simplificación es que consiguen reducir el tamaño del problema en cada paso, por lo que sus tiempos de ejecución suelen ser muy buenos (normalmente de orden logarítmico o lineal). Además pueden admitir una mejora adicional, puesto que en ellos suele poder eliminarse fácilmente la recursión mediante el uso de un bucle iterativo, lo que conlleva menores tiempos de ejecución y menor complejidad espacial al no utilizar la pila de recursión, aunque por contra, también en detrimento de la legibilidad del código resultante.

Por el hecho de usar un diseño recursivo, los algoritmos diseñados mediante la técnica de Divide y Vencerás van a heredar las ventajas e inconvenientes que la recursión plantea:

a) Por un lado el diseño que se obtiene suele ser simple, claro, robusto y elegante, lo que da lugar a una mayor legibilidad y facilidad de depuración y mantenimiento del código obtenido.
b) Sin embargo, los diseños recursivos conllevan normalmente un mayor tiempo de ejecución que los iterativos, además de la complejidad espacial que puede representar el uso de la pila de recursión.

Desde un punto de vista de la eficiencia de los algoritmos Divide y Vencerás, es muy importante conseguir que los subproblemas sean independientes, es decir, que no exista solapamiento entre ellos. De lo contrario el tiempo de ejecución de estos algoritmos será exponencial. Como ejemplo pensemos en el cálculo de la sucesión de Fibonacci, el cual, a pesar de ajustarse al esquema general y de tener sólo dos llamadas recursivas, tan sólo se puede considerar un algoritmo recursivo pero no clasificarlo como diseño Divide y Vencerás. Esta técnica está concebida para resolver problemas de manera eficiente y evidentemente este algoritmo, con tiempo de ejecución exponencial, no lo es.

En cuanto a la eficiencia hay que tener en también en consideración un factor importante durante el diseño del algoritmo: el número de subproblemas y su tamaño, pues esto influye de forma notable en la complejidad del algoritmo resultante. Veámoslo más detenidamente.

En definitiva, el diseño Divide y Vencerás produce algoritmos recursivos cuyo tiempo de ejecución (según vimos en el primer capítulo) se puede expresar mediante una ecuación en recurrencia del tipo:

$$T(n) = \begin{cases} cn^k & \text{si } 1 \le n < b \\ aT(n/b) + cn^k & \text{si } n \ge b \end{cases}$$

donde a, c y k son números reales, n y b son números naturales, y donde $a>0$, $c>0$, $k\ge 0$ y $b>1$. El valor de a representa el número de subproblemas, n/b es el tamaño de cada uno de ellos, y la expresión cn^k representa el coste de descomponer el problema inicial en los a subproblemas y el de combinar las soluciones para producir la solución del problema original, o bien el de resolver un problema elemental. La solución a esta ecuación, tal y como vimos en el problema 1.4 del primer capítulo, puede alcanzar distintas complejidades. Recordemos que el orden de complejidad de la solución a esta ecuación es:

$$T(n) \in \begin{cases} \Theta(n^k) & \text{si } a < b^k \\ \Theta(n^k \log n) & \text{si } a = b^k \\ \Theta(n^{\log_b a}) & \text{si } a > b^k \end{cases}$$

Las diferencias surgen de los distintos valores que pueden tomar a y b, que en definitiva determinan el número de subproblemas y su tamaño. Lo importante es observar que en todos los casos la complejidad es de orden polinómico o polilogarítmico pero nunca exponencial, frente a los algoritmos recursivos que pueden alcanzar esta complejidad en muchos casos (véase el problema 1.3). Esto se debe normalmente a la repetición de los cálculos que se produce al existir solapamiento en los subproblemas en los que se descompone el problema original.

Para aquellos problemas en los que la solución haya de construirse a partir de las soluciones de subproblemas entre los que se produzca necesariamente solapamiento existe otra técnica de diseño más apropiada, y que permite eliminar el problema de la complejidad exponencial debida a la repetición de cálculos. Estamos hablando de la Programación Dinámica, que discutiremos en el capítulo cinco.

Otra consideración importante a la hora de diseñar algoritmos Divide y Vencerás es el reparto de la carga entre los subproblemas, puesto que es importante que la división en subproblemas se haga de la forma más equilibrada posible. En caso contrario nos podemos encontrar con "anomalías de funcionamiento" como le ocurre al algoritmo de ordenación Quicksort. Éste es un representante claro de los algoritmos Divide y Vencerás, y su caso peor aparece cuando existe un desequilibrio total en los subproblemas al descomponer el vector original en dos subvectores de tamaño 0 y n–1. Como vimos en el capítulo anterior, en este caso su orden es $O(n^2)$, frente a la buena complejidad, $O(n\log n)$, que consigue cuando descompone el vector en dos subvectores de igual tamaño.

También es interesante tener presente la dificultad y es esfuerzo requerido en cada una de estas fases va a depender del planteamiento del algoritmo concreto. Por ejemplo, los métodos de ordenación por Mezcla y Quicksort son dos representantes claros de esta técnica pues ambos están diseñados siguiendo el esquema presentado: dividir y combinar.

En lo que sigue del capítulo vamos a desarrollar una serie de ejemplos que ilustran esta técnica de diseño. Existe una serie de algoritmos considerados como representantes clásicos de este diseño, muy especialmente los de ordenación por Mezcla y Quicksort, que no incluimos en este capítulo por haber sido estudiados anteriormente. Sencillamente señalar la diferencia de esfuerzo que realizan en sus fases de división y combinación. La división de Quicksort es costosa, pero una vez ordenados los dos subvectores la combinación es inmediata. Sin embargo, la división que realiza el método de ordenación por Mezcla consiste simplemente en considerar la mitad de los elementos, mientras que su proceso de combinación es el que lleva asociado todo el esfuerzo.

Por último, y antes de comenzar con los ejemplos escogidos, sólo indicar que en muchos de los problemas aquí presentados haremos uso de vectores unidimensionales cuyo tipo viene dado por:

```
CONST n =...; (* numero maximo de elementos del vector *)
TYPE vector = ARRAY [1..n] OF INTEGER;
```

3.2 BÚSQUEDA BINARIA

El algoritmo de búsqueda binaria es un ejemplo claro de la técnica Divide y Vencerás. El problema de partida es decidir si existe un elemento dado x en un vector de enteros ordenado. El hecho de que esté ordenado va a permitir utilizar esta técnica, pues podemos plantear un algoritmo con la siguiente estrategia: compárese el elemento dado x con el que ocupa la posición central del vector. En caso de que coincida con él, hemos solucionado el problema. Pero si son distintos, pueden darse dos situaciones: que x sea mayor que el elemento en posición central, o que sea menor. En cualquiera de los dos casos podemos descartar una de las dos mitades del vector, puesto que si x es mayor que el elemento en posición central, también será mayor que todos los elementos en posiciones anteriores, y al revés. Ahora se procede de forma recursiva sobre la mitad que no hemos descartado.

En este ejemplo la división del problema es fácil, puesto que en cada paso se divide el vector en dos mitades tomando como referencia su posición central. El problema queda reducido a uno de menor tamaño y por ello hablamos de

"simplificación". Por supuesto, aquí no es necesario un proceso de combinación de resultados.

Su caso base se produce cuando el vector tiene sólo un elemento. En esta situación la solución del problema se basa en comparar dicho elemento con x. Como el tamaño de la entrada (en este caso el número de elementos del vector a tratar) se va dividiendo en cada paso por dos, tenemos asegurada la convergencia al caso base.

La función que implementa tal algoritmo ya ha sido expuesta en el primer capítulo como uno de los ejemplos de cálculo de complejidades (problema 1.16), con lo cual no reincidiremos más en ella.

3.3 BÚSQUEDA BINARIA NO CENTRADA

Una de las cuestiones a considerar cuando se diseña un algoritmo mediante la técnica de Divide y Vencerás es la partición y el reparto equilibrado de los subproblemas. Más concretamente, en el problema de la búsqueda binaria nos podemos plantear la siguiente cuestión: supongamos que en vez de dividir el vector de elementos en dos mitades del mismo tamaño, las dividimos en dos partes de tamaños 1/3 y 2/3. ¿Conseguiremos de esta forma un algoritmo mejor que el original?

Solución (☺)

Tal algoritmo puede ser implementado como sigue:

```
PROCEDURE BuscBin2(Var a:vector;
          prim,ult:CARDINAL;x:INTEGER):BOOLEAN;
   VAR tercio:CARDINAL; (* posicion del elemento n/3 *)
BEGIN
   IF (prim>=ult) THEN RETURN a[ult]=x
   ELSE
      tercio:=prim+((ult-prim+1)DIV 3);
      IF x=a[tercio] THEN RETURN TRUE
      ELSIF (x<a[tercio]) THEN RETURN BuscBin2(a,prim,tercio,x)
      ELSE RETURN BuscBin2(a,tercio+1,ult,x)
      END
   END
END BuscBin2;
```

El cálculo del número de operaciones elementales que se realiza en el peor caso de una invocación a esta función puede hacerse de manera análoga a como se hizo en el problema 1.16 cuando se estudió la búsqueda binaria. En este caso obtenemos la ecuación en recurrencia $T(n) = 11 + T(2n/3)$, con la condición inicial $T(1) = 4$. Para resolverla, haciendo el cambio $t_k = T((3/2)^k)$ obtenemos

$$t_k - t_{k-1} = 11,$$

ecuación no homogénea con ecuación característica $(x–1)^2 = 0$. Por tanto,

$$t_k = c_1 k + c_2$$

y, deshaciendo el cambio:

$$T(n) = c_1 \log_{3/2} n + c_2.$$

Para calcular las constantes, nos basaremos en la condición inicial $T(1) = 4$, junto con el valor de $T(2)$, que puede ser calculado apoyándonos en la expresión de la ecuación en recurrencia, es decir, $T(2) = 11 + 4 = 15$. De esta forma obtenemos que:

$$T(n) = 11\log_{3/2} n + 4 \in \Theta(\log n).$$

A pesar de ser del mismo orden de complejidad que la búsqueda binaria clásica, como $3/2 < 2$ se tiene que $\log_{3/2} n > \log n$, es decir, este algoritmo es más lento en el caso peor que el algoritmo original presentado en el problema 1.16 del primer capítulo.

Este hecho puede ser generalizado fácilmente para demostrar que, dividiendo el vector en dos partes de tamaños k y $n–k$, el tiempo de ejecución del algoritmo resultante en el peor caso es:

$$T_k(n) = 11\log_{n/\max\{k,n-k\}} n + 4 \in \Theta(\log n).$$

Ahora bien, para $1 \leq k < n$ sabemos que la función $n/\max\{k,n–k\}$ se mantiene por debajo de 2, y sólo alcanza este valor para $k = n/2$, por lo que

$$\log_{n/\max\{k,n-k\}} n \geq \log n$$

para todo k entre 1 y n. Esto nos indica que la mejor forma de partir el vector para realizar la búsqueda binaria es por la mitad, es decir, tratando de equilibrar los subproblemas en los que realizamos la división tal como comentábamos en la introducción de este capítulo.

3.4 BÚSQUEDA TERNARIA

Podemos plantearnos también diseñar un algoritmo de búsqueda "ternaria", que primero compara con el elemento en posición $n/3$ del vector, si éste es menor que el elemento x a buscar entonces compara con el elemento en posición $2n/3$, y si no coincide con x busca recursivamente en el correspondiente subvector de tamaño 1/3 del original. ¿Conseguimos así un algoritmo mejor que el de búsqueda binaria?

Solución (☺)

Podemos implementar el algoritmo pedido, también de simplificación, de una forma similar a la anterior. La única diferencia es la interpretación de la variable *nterc*, que no indica una posición dentro del vector (como le ocurría a la variable *tercio* del ejercicio previo), sino el número de elementos a tratar ($n/3$). Es por eso por lo que se lo sumamos y restamos a los valores de *prim* y *ult* para obtener las posiciones adecuadas. El algoritmo resultante es:

```
PROCEDURE BuscBin3(VAR a:vector;prim,ult:CARDINAL;x:INTEGER):BOOLEAN;
   VAR nterc:CARDINAL; (* 1/3 del numero de elementos *)
BEGIN
   IF (prim>=ult) THEN RETURN a[ult]=x END;          (* 1 *)
   nterc:=(ult-prim+1)DIV 3;                         (* 2 *)
   IF x=a[prim+nterc] THEN RETURN TRUE         (* 3 *)
   ELSIF x<a[prim+nterc] THEN                        (* 4 *)
      RETURN BuscBin3(a,prim,prim+nterc-1,x)   (* 5 *)
   ELSIF x=a[ult-nterc] THEN RETURN TRUE             (* 6 *)
   ELSIF x<a[ult-nterc] THEN                         (* 7 *)
      RETURN BuscBin3(a,prim+nterc+1,ult-nterc-1,x) (* 8 *)
   ELSE                                              (* 9 *)
      RETURN BuscBin3(a,ult-nterc+1,ult,x)           (* 10 *)
   END                                               (* 11 *)
   END                                               (* 12 *)
END BuscBin3;
```

Para estudiar su complejidad calcularemos el número de operaciones elementales que se realizan:

- En la línea (1) se ejecutan la comparación del *IF* (1 OE), y un acceso a un vector (1 OE), una comparación (1 OE) y un *RETURN* (1 OE) si la condición es verdadera.
- En la línea (2) se realizan 4 OE (resta, suma, división y asignación).
- En la línea (3) hay una suma (1 OE), un acceso a un vector (1 OE) y una comparación (1 OE), más 1 OE si la condición del *IF* es verdadera.
- En la línea (4) se realiza una suma (1 OE), un acceso a un vector (1 OE) y una comparación (1 OE).
- En la línea (5) se efectúan 2 operaciones aritméticas (2 OE), una llamada a la función *BuscBin3* (lo que supone 1 OE), más lo que tarde en ejecutarse la función con un tercio de los elementos y un *RETURN* (1 OE).
- Las líneas (6) y (7) suponen las mismas OE que las líneas (3) y (4).
- Por último, las líneas (8) y (10) efectúan $6+T(n/3)$ y $4+T(n/3)$ cada una: 4 y 2 operaciones aritméticas respectivamente, una llamada a la función *BuscBin3* (lo que supone 1 OE), más lo que tarde en ejecutarse la función con un tercio de los elementos y un *RETURN* (1 OE).

Por tanto, en el peor caso obtenemos la ecuación $T(n) = 23 + T(n/3)$, con la condición inicial $T(1) = 4$. Para resolverla, haciendo el cambio $t_k=T(3^k)$ obtenemos

$$t_k - t_{k-1} = 23,$$

ecuación no homogénea cuya ecuación característica es $(x-1)^2 = 0$. Por tanto, $t_k = c_1k + c_2$ y, deshaciendo el cambio,

$$T(n) = c_1\log_3 n + c_2.$$

Para calcular las constantes, nos basaremos en la condición inicial $T(1) = 4$, junto con el valor de $T(3)$, que puede ser calculado apoyándonos en la expresión de la ecuación en recurrencia: $T(3) = 23 + 4 = 27$. De esta forma obtenemos:

$$T(n) = 23\log_3 n + 4 \in \Theta(\log n).$$

Como $23\log_3 n = 23\log n/\log 3 = 14.51\log n > 11\log n$, el tiempo de ejecución de la búsqueda ternaria es mayor al de la búsqueda binaria, por lo que no consigue ninguna mejora con este algoritmo.

3.5 MULTIPLICACIÓN DE ENTEROS

Sean u y v dos números naturales de n bits donde, por simplicidad, n es una potencia de 2. El algoritmo tradicional para multiplicarlos es de complejidad $O(n^2)$. Ahora bien, un algoritmo basado en la técnica de Divide y Vencerás expuesto en [AHO87] divide los números en dos partes

$$u = a2^{n/2} + b$$
$$v = c2^{n/2} + d$$

siendo a, b, c y d números naturales de $n/2$ bits, y calcula su producto como sigue:

$$uv = (a2^{n/2} + b)(c2^{n/2} + d) = ac2^n + (ad + bc)2^{n/2} + bd$$

Las multiplicaciones ac, ad, bc y bd se realizan usando este algoritmo recursivamente. En primer lugar nos gustaría estudiar la complejidad de este algoritmo para ver si ofrece alguna mejora frente al tradicional.

Por otro lado podríamos pensar en otro algoritmo Divide y Vencerás en el que la expresión $ad+bc$ la sustituimos por la expresión equivalente $(a–b)(d–c)+ac+bd$. Nos cuestionamos si se consigue así un algoritmo mejor que el anterior.

Solución (☺)

Necesitamos en primer lugar determinar su caso base, que en este caso ocurre para $n = 1$, es decir, cuando los dos números son de 1 bit, en cuyo caso uv vale 1 si $u = v = 1$, o bien 0 en otro caso.

Para compararlo con otros algoritmos es necesario determinar su tiempo de ejecución y complejidad. Para ello hemos de observar que para calcular una multiplicación de dos números de n bits $–T(n)–$ es necesario realizar cuatro multiplicaciones de $n/2$ bits (las de ac, ad, bc y bd), dos desplazamientos (las multiplicaciones por 2^n y $2^{n/2}$) y tres sumas de números de a lo más $2n$ bits (en el peor caso, ése es el tamaño del mayor de los tres números, pues ac puede alcanzar n bits). Como las sumas y los desplazamientos son operaciones de orden n, el orden de complejidad del algoritmo viene dada por la expresión:

$$T(n) = 4T(n/2) + An,$$

siendo A una constante. Además, podemos tomar $T(1) = 1$. Para resolver la ecuación hacemos el cambio $t_k = T(2^k)$, obteniendo

$$t_k = 4t_{k-1} + A2^k,$$

ecuación en recurrencia no homogénea con ecuación característica $(x–4)(x–2) = 0$. Aplicando los métodos utilizados en el capítulo 1, la expresión de t_k viene dada por

$$t_k = c_1 4^k + c_2 2^k,$$

y de aquí, deshaciendo el cambio $n = 2^k$ (o lo que es igual, $k = \log n$), obtenemos

$$T(n) = c_1 4^{\log n} + c_2 2^{\log n} = c_1 n^2 + c_2 n \in \mathrm{O}(n^2).$$

Por tanto, este método no mejora el tradicional, también de orden n^2. En cuanto a la modificación sugerida, expresando $ad+bc$ como $(a–b)(d–c)+ac+bd$ obtenemos la siguiente expresión para uv:

$$uv = ac2^n + ((a–b)(d–c)+ac+bd)2^{n/2} + bd.$$

Aunque aparentemente es más complicada, su cálculo precisa tan sólo de tres multiplicaciones de números de $n/2$ bits (ac, bd y $(a–b)(d–c)$) dos desplazamientos de números de n y $n/2$ bits, y seis sumas de números de a lo más $2n$ bits. Tanto las sumas como los desplazamientos son de orden n, y en consecuencia el tiempo de ejecución del algoritmo viene dado por la expresión

$$T(n) = 3T(n/2) + Bn,$$

siendo B una constante. Nuestro caso base sigue siendo el mismo, por lo que podemos volver a tomar $T(1) = 1$. Para resolver la ecuación hacemos el cambio $t_k = T(2^k)$ y obtenemos:

$$t_k = 3t_{k-1} + B2^k,$$

ecuación en recurrencia no homogénea con ecuación característica $(x–3)(x–2) = 0$. Aplicando de nuevo los métodos utilizados en el capítulo 1, la expresión de t_k viene dada por

$$t_k = c_1 3^k + c_2 2^k,$$

y de aquí, deshaciendo el cambio $n = 2^k$ (o lo que es igual, $k = \log n$), obtenemos:

$$T(n) = c_1 3^{\log n} + c_2 2^{\log n} = c_1 n^{\log 3} + c_2 n \in \mathrm{O}(n^{1.59}).$$

Por tanto, este método es de un orden de complejidad menor que el tradicional. ¿Por qué no se enseña entonces en las escuelas y se usa normalmente? Existen fundamentalmente dos razones para ello, una de las cuales es que, aunque más eficiente, es mucho menos intuitivo que el método clásico. La segunda es que las constantes de proporcionalidad que se obtienen en esta caso hacen que el nuevo método sea más eficiente que el tradicional a partir de 500 bits (cf. [AHO87]), y los números que normalmente multiplicamos a mano son, afortunadamente, menores de ese tamaño.

3.6 PRODUCTO DE MATRICES CUADRADAS (1)

Supongamos que necesitamos calcular el producto de matrices cuadradas de orden n, donde n es una potencia de 3. Usando la técnica de Divide y Vencerás, el problema puede ser reducido a una multiplicación de matrices cuadradas de orden 3. El método tradicional para multiplicar estas matrices requiere 27 multiplicaciones. ¿Cuántas multiplicaciones hemos de ser capaces de realizar para multiplicar dos matrices cuadradas de orden 3 para obtener un tiempo total del algoritmo menor que $O(n^{2.81})$? De forma análoga podemos plantearnos la misma pregunta para el caso de matrices cuadradas de orden n, con n una potencia de 4.

Solución (☺)

Utilizando el método tradicional para multiplicar matrices cuadradas de orden tres necesitamos 27 multiplicaciones escalares. Por tanto, basándonos en él para multiplicar matrices cuadradas de orden $n = 3^k$ (multiplicando por bloques), obtenemos que el número de multiplicaciones escalares requerido para este caso (despreciando las adiciones) viene dado por la ecuación en recurrencia

$$T(3^k) = 27T(3^{k-1})$$

con la condición inicial $T(3) = 27$. Resolviendo esta ecuación homogénea, obtenemos que $T(n) = n^3$, resultado clásico ya conocido.

Sea ahora M el número pedido, que indica el número de multiplicaciones escalares necesario para multiplicar dos matrices cuadradas de orden 3. Entonces el número total de multiplicaciones necesario para multiplicar dos matrices cuadradas de orden $n = 3^k$ (multiplicando por bloques) vendrá dado por la ecuación:

$$T(3^k) = M \cdot T(3^{k-1})$$

con la condición inicial $T(3) = M$. Resolviendo esta ecuación homogénea,

$$T(n) = n^{\log_3 M}.$$

Para que la complejidad de $T(n)$ sea menor que $O(n^{2.81})$ se ha de cumplir que $\log_3 M < 2.81$. Por tanto, M ha de verificar que

$$M < 3^{2.81} \approx 22.$$

Es decir, necesitamos encontrar un método para multiplicar matrices de orden 3 con 21 o menos multiplicaciones escalares, en vez de las 27 usuales.

Pasemos ahora al caso de matrices cuadradas de orden $n = 4^k$, y sea N el número de multiplicaciones escalares necesario para multiplicar dos matrices cuadradas de orden 4. En este caso obtenemos la ecuación en recurrencia:

$$T(4^k) = N \cdot T(4^{k-1})$$

con la condición inicial $T(4) = N$. Resolviendo esta ecuación homogénea, obtenemos que

$$T(n) = n^{\log_4 N}.$$

Para que la complejidad de $T(n)$ sea menor que $O(n^{2.81})$ se ha de cumplir que $\log_4 N < 2.81$. Por tanto, N ha de verificar que

$$N < 4^{2.81} \approx 49.$$

Es decir, necesitamos encontrar un método para multiplicar matrices de orden 4 con 48 o menos multiplicaciones escalares, en vez de las 64 ($=4^3$) usuales.

Este tipo de problemas tiene su origen en el descubrimiento de Strassen (1968), que diseñó un método para multiplicar matrices cuadradas de orden 2 usando sólo siete multiplicaciones escalares, en vez de las ocho necesarias en el método clásico (despreciando las adiciones frente a las multiplicaciones). Así se consigue un algoritmo de multiplicación de matrices cuadradas nxn del orden de $n^{\log 7} = n^{2.81}$ en vez de los clásicos $n^{\log 8} = n^3$.

Dadas dos matrices cuadradas de orden 2, A y B, tal algoritmo se basa en obtener la matriz producto C mediante las siguientes fórmulas:

$$\begin{aligned} c_{11} &= m_1 + m_2 - m_4 + m_6 \\ c_{12} &= m_4 + m_5 \\ c_{21} &= m_6 + m_7 \\ c_{22} &= m_2 - m_3 + m_5 - m_7 \end{aligned}$$

donde los valores de $m_1, m_2, ..., m_7$ vienen dados por:

$$\begin{aligned} m_1 &= (a_{12} - a_{22})(b_{21} + b_{22}) \\ m_2 &= (a_{11} + a_{22})(b_{11} + b_{22}) \\ m_3 &= (a_{11} - a_{21})(b_{11} + b_{12}) \\ m_4 &= (a_{11} + a_{12})b_{22} \\ m_5 &= a_{11}(b_{12} - b_{22}) \\ m_6 &= a_{22}(b_{21} - b_{11}) \\ m_7 &= (a_{21} + a_{22})b_{11} \end{aligned}$$

Aunque el número de sumas se ha visto incrementado, el número de multiplicaciones escalares se ha reducido a siete. Utilizando este método para multiplicar por bloques dos matrices cuadradas de orden n, con n potencia de 2, conseguimos un algoritmo cuyo tiempo de ejecución viene dado por la ecuación en recurrencia $T(2^k) = 7T(2^{k-1})$, con la condición inicial $T(2) = 7$. Resolviendo esta ecuación homogénea, obtenemos

$$T(n) = n^{\log 7} \approx n^{2.81}.$$

Para intentar mejorar el algoritmo de Strassen, una primera idea es la de conseguir multiplicar matrices cuadradas de orden 2 con sólo seis multiplicaciones escalares, lo que llevaría a un método de orden $n^{\log 6} < n^{2.81}$. Sin embargo, Hopcroft y Kerr probaron en 1971 que esto es imposible.

Lo siguiente es pensar en matrices cuadradas de orden 3. Según acabamos de ver, si consiguiésemos un método para multiplicar dos matrices de orden 3 con 21 o menos multiplicaciones escalares conseguiríamos un método mejor que el de Strassen. Sin embargo, esto también se ha demostrado que es imposible.

¿Y para matrices de otros órdenes? En general, queremos encontrar un método para multiplicar matrices cuadradas de orden k utilizando menos de $k^{2.81}$

multiplicaciones escalares, en vez de las k^3 requeridas por el método clásico. El primer k que se descubrió fue $k = 70$, y se han llegado a conseguir métodos hasta de orden de $n^{2.376}$, al menos en teoría.

Sin embargo todos estos métodos no tienen ninguna utilidad práctica, debido a las constantes multiplicativas que poseen. Sólo el de Strassen es quizá el único útil, aún teniendo en cuenta que incluso él no muestra su bondad hasta valores de n muy grandes, pues en la práctica no podemos despreciar las adiciones. Incluso para tales matrices la mejora real obtenida es sólo de $n^{1.5}$ frente a $n^{1.41}$, lo que hace de este algoritmo una contribución más teórica que práctica por la dificultad en su codificación y mantenimiento.

3.7 PRODUCTO DE MATRICES CUADRADAS (2)

Sean $n = 2p$, $V = (v_1,v_2,...,v_n)$ y $W = (w_1,w_2,...,w_n)$. Para calcular el producto escalar de ambos vectores podemos usar la fórmula:

$$V \cdot W = \sum_{i=1}^{p} (v_{2i-1} + w_{2i})(v_{2i} + w_{2i-1}) - \sum_{i=1}^{p} v_{2i-1}v_{2i} - \sum_{i=1}^{p} w_{2i-1}w_{2i}$$

que requiere $3n/2$ multiplicaciones. ¿Podemos utilizar esta fórmula para la multiplicación de matrices cuadradas de orden n dando lugar a un método que requiera del orden de $n^3/2 + n^2$ multiplicaciones, en vez de las usuales n^3?

Solución (😐)

La multiplicación de dos matrices cuadradas de orden n puede realizarse utilizando el método clásico, que consiste en realizar n^2 multiplicaciones escalares de vectores: supongamos que queremos calcular el producto $A = B \cdot C$, siendo A, B y C tres matrices cuadradas de orden n.

Llamando B_i a los vectores fila de B (i=1,...,n), y C^j a los vectores columna de C (j=1,...,n),

$$B = \begin{pmatrix} B_1 \\ B_2 \\ \vdots \\ B_n \end{pmatrix} \qquad C = \begin{pmatrix} C^1 & C^2 & \dots & C^n \end{pmatrix}$$

obtenemos que $A[i,j] = B_i \cdot C^j$ ($1 \le i \le n$, $1 \le j \le n$), es decir, cada elemento de A puede calcularse como la multiplicación escalar de dos vectores. Por tanto, son necesarias n^2 multiplicaciones de vectores (una para cada elemento de A).

El método usual de multiplicar escalarmente dos vectores V y W da lugar a n multiplicaciones de elementos, si es que se utiliza la fórmula:

$$V \cdot W = \sum_{i=1}^{n} v_i w_i \,.$$

Ahora bien, estudiando la fórmula propuesta para el caso en que n es par ($n = 2p$):

$$V \cdot W = \sum_{i=1}^{p} (v_{2i-1} + w_{2i})(v_{2i} + w_{2i-1}) - \sum_{i=1}^{p} v_{2i-1} v_{2i} - \sum_{i=1}^{p} w_{2i-1} w_{2i}$$

podemos observar que pueden reutilizarse muchos cálculos, puesto que los dos últimos sumandos de esta ecuación sólo dependen de los vectores V y W. Entonces el método pedido puede implementarse como sigue:

- Primero se calculan las sumas

$$\sum_{i=1}^{p} v_{2i-1} v_{2i}$$

  para cada uno de los vectores fila de B y columna de C. Hay que realizar $2n$ de estas operaciones, y cada una requiere $n/2$ multiplicaciones, lo que implica n^2 multiplicaciones de elementos en esta fase.
- Después se calcula cada uno de los elementos de A como $A[i,j] = B_i \cdot C^j$ utilizando la fórmula anterior, pero en donde ya hemos calculado (en el paso previo) los dos últimos términos de los tres que componen la expresión. Por tanto, para cada elemento de A sólo es necesario realizar ahora $n/2$ multiplicaciones de elementos. Como hay que calcular n^2 elementos en total, realizaremos en esta fase $n^3/2$ multiplicaciones.

Sumando el número de multiplicaciones de ambas fases, hemos conseguido un método con el número de operaciones pedido.

3.8 MEDIANA DE DOS VECTORES

Sean X e Y dos vectores de tamaño n, ordenados de forma no decreciente. Necesitamos implementar un algoritmo para calcular la *mediana* de los $2n$ elementos que contienen X e Y. Recordemos que la mediana de un vector de k elementos es aquel elemento que ocupa la posición $(k+1)\div 2$ una vez el vector está ordenado de forma creciente. Dicho de otra forma, la mediada es aquel elemento que, una vez ordenado el vector, deja la mitad de los elementos a cada uno de sus lados. Como en nuestro caso $k = 2n$ (y por tanto par) buscamos el elemento en posición n de la unión ordenada de X e Y.

Solución (😐)

Para resolver el problema utilizaremos una idea basada en el método de búsqueda binaria. Comenzaremos estudiando el caso base, que ocurre cuando tenemos dos vectores de un elemento cada uno ($n = 1$). En este caso la mediana será el mínimo de ambos números, pues obedeciendo a la definición sería el elemento que ocupa la primera posición ($n = 1$) si ordenásemos ambos vectores.

Respecto al caso general, existe una forma de dividir el problema en subproblemas más pequeños. Sea Z el vector resultante de mezclar ordenadamente los vectores X e Y, y sea m_Z la mediana de Z. Apoyándonos en el hecho de que X e Y se encuentran ordenados, es fácil calcular sus medianas (son los elementos que ocupan las posiciones centrales de ambos vectores), y que llamaremos m_X y m_Y.

Ahora bien, si $m_X = m_Y$ entonces la mediana m_Z va a coincidir también con ellas, pues al mezclar ambos vectores las medianas se situarán en el centro del vector.

Si tenemos que $m_X < m_Y$, podemos afirmar que la mediana m_Z va a ser mayor que m_X pero menor que m_Y, y por tanto m_Z va a encontrarse en algún lugar de la segunda mitad del vector X o en algún lugar de la primera mitad de Y (por estar ambos vectores ordenados).

Análogamente, si $m_X > m_Y$ la mediana m_Z va a ser mayor que m_Y pero menor que m_X, y por tanto m_Z va a encontrarse en algún lugar de la primera mitad del vector X o en algún lugar de la segunda mitad de Y (por estar ambos vectores ordenados). Esta idea nos lleva a la siguiente versión de nuestro algoritmo:

```
PROCEDURE Mediana(VAR X,Y:vector;primX,ultX,primY,ultY:CARDINAL)
                                                              :INTEGER;
   VAR posX,posY:CARDINAL; nitems:CARDINAL;
BEGIN
   IF (primX>=ultX) AND (primY>=ultY) THEN (* caso base *)
      RETURN Min2(X[ultX],Y[ultY])
   END;
   nitems:=ultX-primX+1;
   IF nitems=2 THEN      (* 2 vectores de 2 elementos cada uno *)
      IF X[ultX]<Y[primY] THEN RETURN X[ultX]
      ELSIF Y[ultY]<X[primX] THEN RETURN Y[ultY]
      ELSE RETURN Max2(X[primX],Y[primY])
      END
   END;
   nitems:=(nitems-1) DIV 2;  (* caso general *)
   posX:=primX+nitems;
   posY:=primY+nitems;
   IF X[posX]=Y[posY] THEN RETURN X[posX]
   ELSIF X[posX]<Y[posY] THEN
      RETURN Mediana(X,Y,ultX-nitems,ultX,primY,primY+nitems)
   ELSE
      RETURN Mediana(X,Y,primX,primX+nitems,ultY-nitems,ultY)
   END;
END Mediana;
```

que calcula la solución pedida cuando lo invocamos como *Mediana*(X,Y,1,*n*,1,*n*). Es conveniente observar que uno de los invariantes del algoritmo es que el número de elementos de los subvectores X e Y coincide en cada uno de los pasos (es decir, se verifica que *ultX–primX+1=ultY–primY+1=n*) y que en cada invocación se reduce a la mitad, y por ello se trata de un algoritmo de simplificación. Este hecho de que el tamaño de los dos vectores es siempre igual en cada invocación es lo que nos permite garantizar que la media que se calcula recursivamente en los trozos coincide con la mediana buscada antes de realizar los descartes de elementos. En particular, basta con observar que los trozos descartados eliminan el mismo número de elementos.

En otro orden de cosas, las funciones *Min2* y *Max2* que utiliza este algoritmo son las que calculan respectivamente el mínimo y el máximo de dos números enteros.

Para el estudio de su complejidad, expresamos su tiempo de ejecución como

$$T(2n) = T(n) + A,$$

siendo A una constante. Entonces hacemos el cambio $t_k = T(2^k)$, y entonces

$$t_{k+1} = t_k + A,$$

ecuación en recurrencia no homogénea con ecuación característica $(x-1)^2 = 0$. Aplicando los métodos utilizados en el capítulo 1, la expresión de t_k viene dada por

$$t_k = c_1 k + c_2,$$

y de aquí, deshaciendo el cambio $n = 2^k$ (o lo que es igual, $k = \log n$), obtenemos:

$$T(n) = c_1 \log n + c_2 \in O(\log n).$$

3.9 EL ELEMENTO EN SU POSICIÓN

Sea $a[1..n]$ un vector ordenado de enteros todos distintos. Nuestro problema es implementar un algoritmo de complejidad $O(\log n)$ en el peor caso capaz de encontrar un índice i tal que $1 \le i \le n$ y $a[i] = i$, suponiendo que tal índice exista.

Solución (☺)

Podemos implementar el algoritmo pedido apoyándonos en el hecho de que el vector está originalmente ordenado. Por tanto, podemos usar un método basado en la idea de la búsqueda binaria, en donde examinamos el elemento en mitad del vector (su mediana). Si $a[(n+1)\div 2] = (n+1)\div 2$, ésa es la posición pedida. Si $a[(n+1)\div 2]$ fuera mayor que $(n+1)\div 2$, la posición pedida ha de encontrarse antes de la mitad, y en caso contrario detrás de ella.

Esto da lugar al siguiente algoritmo:

```
PROCEDURE Localiza(VAR a:vector;prim,ult:CARDINAL):CARDINAL;
   VAR i:CARDINAL;
BEGIN
   IF prim>ult THEN RETURN 0 END; (* no existe tal indice *)
   i:=(prim+ult+1)DIV 2;
   IF a[i]=INTEGER(i) THEN RETURN i
   ELSIF a[i]>INTEGER(i) THEN RETURN Localiza(a,prim,i-1)
   ELSE RETURN Localiza(a,i+1,ult)
   END;
END Localiza;
```

Tal método sigue la técnica Divide y Vencerás puesto que en cada invocación reduce el problema a uno más pequeño, buscando la posición pedida en un subvector con la mitad de elementos. Este hecho hace que su complejidad sea de orden logarítmico, puesto que su tiempo de ejecución viene dado por la expresión:

$$T(n) = T(n/2) + A$$

siendo A una constante. Nuestro caso base sigue siendo el mismo, por lo que podemos volver a tomar $T(1) = 1$. Para resolver la ecuación hacemos el cambio $t_k = T(2^k)$, por lo que

$$t_k = t_{k-1} + A,$$

ecuación en recurrencia no homogénea con ecuación característica $(x-1)^2 = 0$. Aplicando los métodos utilizados en el capítulo 1, la expresión de t_k viene dada por

$$t_k = c_1 k + c_2,$$

y de aquí, deshaciendo el cambio $n = 2^k$ (o lo que es igual, $k = \log n$), obtenemos:

$$T(n) = c_1 \log n + c_2 \in O(\log n).$$

3.10 REPETICIÓN DE CÁLCULOS EN FIBONACCI

En el cálculo recursivo del n-ésimo número de Fibonacci, $fib(n)$, necesitamos determinar para cada $0 \le k < n$ el número de veces que se calcula $fib(k)$.

Solución (☺)

Para el cálculo recursivo de $fib(n)$ podemos utilizar la ecuación en recurrencia:

$$fib(n) = fib(n-1) + fib(n-2) \qquad (n > 1)$$

con las condiciones iniciales $fib(1) = fib(0) = 1$. Por tanto, el número de veces que se va a calcular un número $fib(k)$ en el cómputo de $fib(n)$ coincidirá con el número de veces que se calcule en el cómputo de $fib(n-1)$ más el número de veces que se calcule en el cómputo de $fib(n-2)$. En consecuencia, llamando $N_k(n)$ al número de veces que se calcula $fib(k)$ en el cómputo de $fib(n)$, obtenemos la ecuación en recurrencia:

$$N_k(n) = N_k(n-1) + N_k(n-2) \qquad (1 \le k \le n),$$

que es a su vez una ecuación de Fibonacci. Sin embargo sus condiciones iniciales son diferentes, pues para $k = n$ y $k = n-1$ los números $fib(n)$ y $fib(n-1)$ sólo se calculan una vez, con lo cual obtenemos que $N_n(n) = N_{n-1}(n) = 1$ para todo n. Además, $N_k(n) = 0$ si $k > n$. Así, vamos obteniendo:

$N_n(n) = N_{n-1}(n) = 1$	para todo $n>1$.
$N_{n-2}(n) = N_{n-2}(n-1) + N_{n-2}(n-2) = 1 + 1 = 2$	por la ecuación anterior.
$N_{n-3}(n) = N_{n-3}(n-1) + N_{n-3}(n-2) = 2 + 1 = 3$	por las ecuaciones anteriores.
$N_{n-4}(n) = N_{n-4}(n-1) + N_{n-4}(n-2) = 3 + 2 = 5$	por las ecuaciones anteriores.

De esta forma llegamos a la expresión de $N_k(n)$:

$$N_k(n) = fib(n-k), (1 \le k \le n).$$

Respecto al caso especial $N_0(n)$, su valor se calculará tantas veces como se calcule $fib(2)$ en el cómputo de $fib(n)$, puesto que no hará falta calcularlo en el cómputo de $fib(1)$. Por tanto,

$$N_0(n) = N_2(n) = fib(n-2).$$

Es importante señalar en este punto que los resultados obtenidos en este apartado muestran la ineficiencia del algoritmo puramente recursivo para el cálculo de los números de Fibonacci, no sólo por el mero hecho del uso de la pila de ejecución por ser recursivo, sino por la enorme repetición de los cálculos que se realizan. Evitar esta repetición para conseguir tiempos de ejecución polinómicos en vez de exponenciales es una de las ideas claves de la técnica de Programación Dinámica, que será discutida en el capítulo 5.

3.11 EL ELEMENTO MAYORITARIO

Sea $a[1..n]$ un vector de enteros. Un elemento x se denomina *elemento mayoritario* de a si x aparece en el vector más de $n/2$ veces, es decir, $Card\{i \mid a[i]=x\} > n/2$. Necesitamos implementar un algoritmo capaz de decidir si un vector dado contiene un elemento mayoritario (no puede haber más de uno) y calcular su tiempo de ejecución.

Solución (𝒢)

Al pensar en una posible solución a este ejercicio podemos considerar primero lo que ocurre cuando el vector está ordenado. En ese caso la solución es trivial pues los elementos iguales aparecen juntos. Basta por tanto recorrer el vector buscando un rellano de longitud mayor que $n/2$. Utilizamos el término "rellano" en el mismo sentido que lo hace Gries en su problema *El rellano más largo*: aquel subvector cuyos elementos son todos iguales [GRI81]. El algoritmo puede ser implementado en este caso como sigue:

```
PROCEDURE Mayoritario(VAR a:vector;prim,ult:CARDINAL):BOOLEAN;
(* supone que el vector esta ordenado *)
   VAR mitad,i:CARDINAL;
BEGIN
   IF prim=ult THEN RETURN TRUE END;
   mitad:=(prim+ult+1)DIV 2;
   FOR i:=mitad TO ult DO
      IF a[i]=a[i-mitad+prim] THEN RETURN TRUE END;
   END;
   RETURN FALSE;
END Mayoritario;
```

Este procedimiento comprueba si el vector $a[prim..ult]$ contiene un elemento mayoritario o no, y supone para ello que dicho vector está ordenado en forma creciente.

Por tanto, una primera solución al problema consiste en ordenar el vector y después ejecutar la función anterior. La complejidad de esta solución es de orden

O($n$log$n$), pues de este orden son los procedimientos que ordenan un vector. La complejidad del procedimiento *Mayoritario* no influye frente a ella por ser de orden lineal.

En general, llamaremos algoritmos "en línea" (del término inglés *scanning*) a los algoritmos para vectores que resuelven el problema recorriéndolo una sola vez, sin utilizar otros vectores auxiliares y que permiten en cualquier momento dar una respuesta al problema para la subsecuencia leída hasta entonces. El anterior es un claro ejemplo de este tipo de algoritmos.

Otro algoritmo también muy intuitivo cuando el vector está ordenado es el siguiente: en caso de haber elemento mayoritario, éste ha de encontrarse en la posición (n+1)÷2. Basta entonces con recorrer el vector desde ese elemento hacia atrás y hacia delante, contando el número de veces que se repite. Si este número es mayor que n/2, el vector tiene elemento mayoritario. Sin embargo, la complejidad de este algoritmo coincide con la del anterior por tener que ordenar primero el vector, por lo que no representa ninguna mejora.

Sin suponer que el vector se encuentra ordenado, una forma de plantear la solución a este problema aparece indicada en [WEI95]. La idea consiste en encontrar primero un posible candidato, es decir, el único elemento que podría ser mayoritario, y luego comprobar si realmente lo es. De no serlo, el vector no admitiría elemento mayoritario, como le ocurre al vector [1,1,2,2,3,3,3,3,4].

Para encontrar el candidato, suponiendo que el número de elementos del vector es par, vamos a ir comparándolos por pares (a[1] con a[2], a[3] con a[4], etc.). Si para algún k = 1,3,5,7,... se tiene que $a[k] = a[k+1]$ entonces copiaremos $a[k]$ en un segundo vector auxiliar b. Una vez recorrido todo el vector a, buscaremos recursivamente un candidato para el vector b, y así sucesivamente. Este método obecede a la técnica de Divide y Vencerás pues en cada paso el número de elementos se reduce a menos de la mitad.

Vamos a considerar tres cuestiones para implementar un algoritmo basado en esta idea: (*i*) su caso base, (*ii*) lo que ocurre cuando el número de elementos es impar, y (*iii*) cómo eliminar el uso recursivo del vector auxiliar b para no aumentar excesivamente la complejidad espacial del método.

i) En primer lugar, el caso base de la recursión ocurre cuando disponemos de un vector con uno o dos elementos. En este caso existe elemento mayoritario si los elementos del vector son iguales.

ii) Si el número de elementos n del vector es impar y mayor que 2, aplicaremos la idea anterior para el subvector compuesto por sus primeros n–1 elementos. Como resultado puede que obtengamos que dicho subvector contiene un candidato a elemento mayoritario, con lo cual éste lo será también para el vector completo. Pero si la búsqueda de candidato para el subvector de n–1 elementos no encuentra ninguno, escogeremos como candidato el n-ésimo elemento.

iii) Respecto a cómo eliminar el vector auxiliar b, podemos pensar en utilizar el propio vector a para ir almacenando los elementos que vayan quedando tras cada una de las pasadas.

Esto da lugar al siguiente algoritmo:

```
PROCEDURE Mayoritario2(VAR a:vector;prim,ult:CARDINAL):BOOLEAN;
```

```
(* comprueba si a[prim..ult] contiene un elemento mayoritario *)
   VAR suma,i:CARDINAL; candidato: INTEGER;
BEGIN
   suma:=0;
   IF BuscaCandidato(a,prim,ult,candidato) THEN
      (* comprobacion de si el candidato es o no mayoritario *)
      FOR i:=prim TO ult DO
         IF a[i]=candidato THEN INC(suma) END;
      END
   END;
   RETURN suma>((ult-prim+1)DIV 2);
END Mayoritario2;
```

La función *BuscaCandidato* intenta encontrar un elemento mayoritario:

```
PROCEDURE BuscaCandidato(VAR a:vector;prim,ult:CARDINAL;
                         VAR candidato:INTEGER):BOOLEAN;
   VAR i,j:CARDINAL;
BEGIN
   candidato:=a[prim];
   IF ult<prim THEN RETURN FALSE END;  (* casos base *)
   IF ult=prim THEN RETURN TRUE END;
   IF prim+1=ult THEN
      candidato:=a[ult];
      RETURN (a[prim]=a[ult])
   END;
   j:=prim; (* caso general *)
   IF ((ult-prim+1)MOD 2)=0 THEN (* n par *)
      FOR i:=prim+1 TO ult BY 2 DO
         IF a[i-1]=a[i] THEN
            a[j]:=a[i]; INC(j)
         END
      END;
      RETURN BuscaCandidato(a,prim,j-1,candidato);
   ELSE (* n impar *)
      FOR i:=prim TO ult-1 BY 2 DO
         IF a[i]=a[i+1] THEN a[j]:=a[i]; INC(j) END
      END;

      IF NOT BuscaCandidato(a,prim,j-1,candidato) THEN
         candidato:=a[ult]
      END;
      RETURN TRUE
   END;
END BuscaCandidato;
```

La complejidad del algoritmo *BuscaCandidato* es de orden O(n), pues en cada iteración del procedimiento general se efectúa un ciclo de orden n, junto con una llamada recursiva a la función, pero a lo sumo con $n/2$ elementos. Esto permite expresar su tiempo de ejecución $T(n)$ mediante la ecuación

$$T(n) = T(n/2) + An + B,$$

siendo A y B constantes. Para resolver la ecuación hacemos el cambio $t_k = T(2^k)$, por lo que

$$t_k = t_{k-1} + A2^k + B,$$

ecuación en recurrencia no homogénea con ecuación característica $(x-1)^2(x-2)=0$. Aplicando los métodos utilizados en el capítulo 1, la expresión de t_k viene dada por

$$t_k = c_1k + c_2 + c_32^k,$$

y de aquí, deshaciendo el cambio $n = 2^k$ (o lo que es igual, $k = \log n$), obtenemos:

$$T(n) = c_1\log n + c_2 + c_3n \in \mathrm{O}(n).$$

Este algoritmo es, por tanto, mejor que el que ordena primero el vector.

3.12 LA MODA DE UN VECTOR

Deseamos implementar un algoritmo Divide y Vencerás para encontrar la *moda* de un vector, es decir, aquel elemento que se repite más veces.

Solución ()

La primera solución que puede plantearse para resolver este problema es a partir de la propia definición de moda. Se calcula la frecuencia con la que aparece cada uno de los elementos del vector y se escoge aquel que se repite más veces. Esto da lugar a la siguiente función:

```
PROCEDURE Moda(VAR a:vector;prim,ult:CARDINAL):INTEGER;
   VAR i,frec,maxfrec:CARDINAL;moda:INTEGER;
BEGIN
   IF prim=ult THEN RETURN a[prim] END;
   moda:=a[prim];
   maxfrec:=Frecuencia(a,a[prim],prim,ult);

   FOR i:=prim+1 TO ult DO
      frec:=Frecuencia(a,a[i],i,ult);
      IF frec>maxfrec THEN
         maxfrec:=frec;
         moda:=a[i]
      END;
   END;
```

```
   RETURN moda
END Moda;
```

La función *Frecuencia* es la que calcula el número de veces que se repite un elemento dado:

```
PROCEDURE Frecuencia(VAR
a:vector;p:INTEGER;prim,ult:CARDINAL):CARDINAL;
   VAR i,suma:CARDINAL;
BEGIN
   IF prim>ult THEN RETURN 0 END;
   suma:=0;
   FOR i:=prim TO ult DO
      IF a[i]=p THEN
         INC(suma)
      END;
   END;
   RETURN suma;
END Frecuencia;
```

La complejidad de la función *Frecuencia* es O(n), lo que hace que la complejidad del algoritmo presentado para calcular la moda sea de orden O(n^2).

Ahora bien, en el caso particular en el que el vector esté ordenado existe una forma mucho más eficiente para calcular su moda, recorriendo el vector una sola vez. El algoritmo que presentamos a continuación es del tipo "en línea" y está basado en el algoritmo para calcular el rellano más largo de un vector (ver [GRI81] y el problema anterior), y da lugar a la siguiente función:

```
PROCEDURE Moda2(VAR a:vector;prim,ult:CARDINAL):INTEGER;
(* supone que el vector a[prim..ult] esta ordenado *)
   VAR i,p:CARDINAL;moda:INTEGER;
BEGIN
   i:=prim+1; p:=1; moda:=a[prim];
   WHILE i<=ult DO
      IF a[i-p]=a[i] THEN INC(p); moda:=a[i] END;
      INC(i);
   END;
   RETURN moda
END Moda2;
```

La complejidad de este algoritmo es O(n). Sin embargo, como es preciso ordenar primero el vector antes de invocar a esta función, la complejidad del algoritmo resultante sería de orden O($n$log$n$).

Existe sin embargo una solución aplicando la técnica de Divide y Vencerás, indicada en [GON91] capaz de ofrecer una complejidad mejor que O($n$log$n$).

El algoritmo se basa en la utilización de dos conjuntos, *homog* y *heterog*, que van a contener en cada paso subvectores del vector original *a*[*prim..ult*]. El primero

de ellos contiene sólo aquellos subvectores que tienen todos sus elementos iguales, y el segundo aquellos que tienen sus elementos distintos.

Inicialmente *homog* es vacío y *heterog* contiene al vector completo. En cada paso vamos a extraer el subvector *p* de *heterog* de mayor longitud, calcularemos su mediana y lo dividiremos en tres subvectores: *p*1, que contiene los elementos de *p* menores a su mediana, *p*2, con los elementos de *p* iguales a su mediana, y *p*3, con los elementos de *p* mayores a su mediana. Entonces actualizaremos los conjuntos *homog* y *heterog*, pues en el primero introduciremos *p*2 y en el segundo *p*1 y *p*3.

Este proceso lo repetiremos mientras que la longitud del subvector más largo de *heterog* sea mayor que la del más largo de *homog*. Una vez llegado a este punto, el subvector más largo de *homog* contendrá la moda del vector original.

Para implementar tal esquema haremos uso de un tipo abstracto de datos que representa a los conjuntos de subvectores (*CJTS*), que aporta las operaciones sobre los elementos de tal tipo, que supondremos implementado. Los subvectores van a ser representados como ternas en donde el primer elemento es un vector, y los otros dos indican las posiciones de comienzo y fin de sus elementos.

El algoritmo que desarrolla esta idea para encontrar la moda de un vector es el siguiente:

```
PROCEDURE Moda3(VAR a:vector;prim,ult:CARDINAL):INTEGER;
   VAR  p,p1,p2,p3:CJTS.subvector;
        homog,heterog:CJTS.conjunto;
        mediana:INTEGER;
        izq,der:CARDINAL;
BEGIN
   CJTS.Crear(homog);
   CJTS.Crear(heterog);
   (* insertamos a[prim..ult] en heterog: *)
   p.a:=a;
   p.prim:=prim;
   p.ult:=ult;
   CJTS.Insertar(heterog,p);
   WHILE CJTS.Long_Mayor(heterog)> CJTS.Long_Mayor(homog) DO
      p:=CJTS.Mayor(heterog); (* esto extrae p del conjunto *)
      (* calculamos la mediana de p *)
      mediana:=Kesimo(p.a,p.prim,p.ult,(p.ult-p.prim+2)DIV 2);
      (* y dividimos p en 3 subvectores *)
      Pivote2(p.a,mediana,p.prim,p.ult,izq,der);
      p1.a:=p.a; p1.prim:=p.prim; p1.ult:=izq-1;
      p2.a:=p.a; p2.prim:=izq; p2.ult:=der-1;
      p3.a:=p.a; p3.prim:=der; p3.ult:=p.ult;
      (* ahora modificamos los conjuntos heterog y homog *)
      IF p1.prim<p1.ult THEN CJTS.Insertar(heterog,p1) END;
      IF p3.prim<p3.ult THEN CJTS.Insertar(heterog,p3) END;
      IF p2.prim<p2.ult THEN CJTS.Insertar(homog,p2) END
   END; (* WHILE *)
   IF CJTS.Esvacio(homog) THEN
```

```
        RETURN a[prim]
      END;
      p:=CJTS.Mayor(homog);
      CJTS.Destruir(homog);
      CJTS.Destruir(heterog);
      RETURN p.a[p.prim]
  END Moda3;
```

Las funciones *Kesimo* y *Pivote2* fueron definidas e implementadas en el capítulo anterior, y son utilizadas aquí para calcular la mediana del vector y dividirlo en tres partes, de acuerdo al esquema general presentado.

El estudio de la complejidad de este algoritmo no es fácil. Sólo mencionaremos que su complejidad, como se muestra en [GON91], es de orden O($n$log(n/m)), siendo m la multiplicidad de la moda, y que por las constantes multiplicativas que posee, resulta ser mejor que el algoritmo *Moda2* presentado anteriormente. Sin embargo, la dificultad de su diseño e implementación han de tenerse también en cuenta, pues complican notablemente su codificación y mantenimiento.

3.13 EL TORNEO DE TENIS

Necesitamos organizar un torneo de tenis con n jugadores en donde cada jugador ha de jugar exactamente una vez contra cada uno de sus posibles n–1 competidores, y además ha de jugar un partido cada día, teniendo a lo sumo un día de descanso en todo el torneo. Por ejemplo, las siguientes tablas son posibles cuadrantes resultado para torneos con 5 y 6 jugadores:

Jug	d1	d2	d3	d4	d5
1	2	3	4	5	–
2	1	5	3	–	4
3	–	1	2	4	5
4	5	–	1	3	2
5	4	2	–	1	3

Jug	d1	d2	d3	d4	d5
1	2	3	4	5	6
2	1	5	3	6	4
3	6	1	2	4	5
4	5	6	1	3	2
5	4	2	6	1	3
6	3	4	5	2	1

Solución (😐)

Para resolver este problema procederemos por partes, considerando los siguientes casos:

a) Si n es potencia de 2, implementaremos un algoritmo para construir un cuadrante de partidas del torneo que permita terminarlo en n–1 días.

b) Dado cualquier n>1, implementaremos un algoritmo para construir un cuadrante de partidas del torneo que permita terminarlo en n–1 días si n es par, o en n días si n es impar.

En el primer caso suponemos que n es una potencia de 2. El caso más simple se produce cuando sólo tenemos dos jugadores, cuya solución es fácil pues basta enfrentar uno contra el otro.

Si n>2, aplicaremos la técnica de Divide y Vencerás para construir la tabla pedida suponiendo que tenemos calculada ya una solución para la mitad de los jugadores, esto es, que tenemos relleno el cuadrante superior izquierdo de la tabla. En este caso los otros tres cuadrantes no son difíciles de rellenar, como puede observarse en la siguiente figura, y en donde se han tenido en cuenta la siguientes consideraciones para su construcción:

1. El cuadrante inferior izquierdo debe enfrentar a los jugadores de número superior entre ellos, por lo que se obtiene sumando n/2 a los valores del cuadrante superior izquierdo.
2. El cuadrante superior derecho enfrenta a los jugadores con menores y mayores números, y se puede obtener enfrentando a los jugadores numerados 1 a n/2 contra (n/2)+1 a n respectivamente en el día n/2, y después rotando los valores (n/2)+1 a n cada día.
3. Análogamente, el cuadrante inferior derecho enfrenta a los jugadores de mayor número contra los de menor número, y se puede obtener enfrentando a los jugadores (n/2)+1 a n contra 1 a n/2 respectivamente en el día n/2, y después rotando los valores 1 a n cada día, pero en sentido contrario a como lo hemos hecho para el cuadrante superior derecho.

	d1
J1	2
J2	1

	d1	d2	d3
J1	2	3	4
J2	1	4	3
J3	4	1	2
J4	3	2	1

	d1	d2	d3	d4	d5	d6	d7
J1	2	3	4	5	6	7	8
J2	1	4	3	6	7	8	5
J3	4	1	2	7	8	5	6
J4	3	2	1	8	5	6	7
J5	6	7	8	1	4	3	2
J6	5	8	7	2	1	4	3
J7	8	5	6	3	2	1	4
J8	7	6	5	4	3	2	1

El algoritmo que implementa tal estrategia es:

```
CONST MAXJUG =...; (* numero maximo de jugadores *)
TYPE cuadrante = ARRAY [1..MAXJUG],[1..MAXJUG] OF CARDINAL;

PROCEDURE Torneo(n:CARDINAL;VAR tabla:cuadrante);
(* n es el numero de jugadores, que suponemos potencia de 2 *)
(* en tabla devuelve el cuadrante de partidos relleno *)
   VAR jug,dia:CARDINAL;
BEGIN
   IF n=2 THEN   (* caso base *)
```

```
      tabla[1,1]:=2;
      tabla[2,1]:=1
   ELSE
      (* primero se rellena el cuadrante superior izquierdo *)
      Torneo(n DIV 2, tabla);  (* llamada recursiva *)
      (* despues el cuadrante inferior izquierdo *)
      FOR jug:=(n DIV 2)+1 TO n DO
         FOR dia:=1 TO (n DIV 2)-1 DO
            tabla[jug,dia]:=tabla[jug-(n DIV 2),dia]+(n DIV 2)
         END
      END;
      (* luego el cuadrante superior derecho *)
      FOR jug:=1 TO (n DIV 2) DO
         FOR dia:=(n DIV 2) TO n-1 DO
            IF (jug+dia)<=n THEN tabla[jug,dia]:=jug+dia
            ELSE tabla[jug,dia]:=jug+dia-(n DIV 2)
            END
         END
      END;
      (* y finalmente el cuadrante inferior derecho *)
      FOR jug:=(n DIV 2)+1 TO n DO
         FOR dia:=(n DIV 2) TO n-1 DO
            IF jug>dia THEN tabla[jug,dia]:=jug-dia
            ELSE tabla[jug,dia]:=(jug+(n DIV 2))-dia
            END
         END
      END
   END (* IF *)
END Torneo;
```

Supongamos ahora que el número de jugadores n es impar y que sabemos resolver el problema para un número par de jugadores. En este caso existe una solución al problema en n días, que se construye a partir de la solución al problema para n+1 jugadores. Si n es impar entonces n+1 es par, y sea $S[1..n+1][1..n]$ el cuadrante solución para n+1 jugadores. Entonces podemos obtener el cuadrante solución para n jugadores $T[1..n][1..n]$ como:

$$T[jug, dia] = \begin{cases} S[jug, dia] & \text{si } S[jug, dia] \neq n+1 \\ 0 & \text{si } S[jug, dia] = n+1 \end{cases}$$

Es decir, utilizamos la convención de que un 0 en la posición $[i,j]$ de la tabla indica que el jugador i descansa (no se enfrenta a nadie) el día j, y aprovechamos este hecho para construir el cuadrante solución pedido. Por ejemplo, para el caso de n = 3 nos apoyamos en la tabla construida para 4 jugadores, eliminando la última fila y sustituyendo las apariciones del jugador número 4 por ceros:

	d1	d2	d3
J1	2	3	0
J2	1	0	3
J3	0	1	2

Sólo nos queda resolver el caso en que n es par, y para ello llamaremos m al número $n \div 2$. Utilizando la técnica de Divide y Vencerás vamos a encontrar una forma de resolver el problema para n jugadores suponiendo que lo tenemos resuelto para m. Distinguiremos dos casos:

Si m es par, sabemos que existe una solución para enfrentar a esos m jugadores entre sí en $m-1$ días. Éste va a constituir el cuadrante superior izquierdo de la solución para n. Los otros tres cuadrantes se van a construir de igual forma al caso anterior cuando n es potencia de 2.

Si m es impar, su solución necesita m días. Esto va a volver a constituir el cuadrante superior izquierdo de la solución para el caso de n jugadores, pero ahora nos encontramos con que tiene algunos ceros, que necesitaremos rellenar convenientemente.

El cuadrante inferior izquierdo va a construirse como anteriormente, es decir, va a enfrentar a los jugadores de número superior entre ellos, por lo que se obtiene sumando $n/2$ a los valores del cuadrante superior que no sean cero.

El cuadrante superior derecho enfrenta a los jugadores con menores y mayores números y se va a obtener de forma similar a como lo construíamos antes, solo que no va a enfrentar a los jugadores 1 a $n/2$ contra $(n/2)+1$ a n en el día $n/2$, sino en cada una de las posiciones vacías del cuadrante superior izquierda. Los demás días del cuadrante superior derecho sí se van a obtener rotando esos valores cada día.

Análogamente, el cuadrante inferior derecho enfrenta a los jugadores de mayor número contra los de menor número, y se va a obtener enfrentando a los jugadores $(n/2)+1$ a n contra 1 a $n/2$ respectivamente, pero no en el día $n/2$, sino ocupando las posiciones vacías del cuadrante inferior izquierdo. Los valores restantes sí se obtendrán como antes, rotando los valores 1 a n cada día, pero en sentido contrario a como lo hemos hecho para el cuadrante superior.

Este proceso puede apreciarse en la siguiente figura para $n = 6$:

	d1	d2	d3
J1	2	3	0
J2	1	0	3
J3	0	1	2

$m = 3$

	d1	d2	d3	d4	d5
J1	2	3	0		
J2	1	0	3		
J3	0	1	2		
J4	5	6	0		
J5	4	0	6		
J6	0	4	5		

cuadrantes 1º y 2º.

	d1	d2	d3	d4	d5
J1	2	3	4	5	6
J2	1	5	3	6	4
J3	6	1	2	4	5
J4	5	6	1	3	2
J5	4	2	6	1	3
J6	3	4	5	2	1

cuadrantes 3º y 4º.

y el algoritmo que lleva a cabo tal estrategia puede ser implementado como sigue:

```
PROCEDURE Torneo(n:CARDINAL; VAR tabla:cuadrante);
(* n es el num. jugadores y en tabla se devuelve la solucion *)
   VAR jug,dia,m:CARDINAL;
BEGIN
```

```
IF n=2 THEN   (* caso base *)
   tabla[1,1]:=2; tabla[2,1]:=1
ELSIF (n MOD 2)<>0 THEN  (* n impar *)
   Torneo(n+1,tabla);    (* llamada recursiva *)
   FOR jug:=1 TO n DO    (* eliminamos el jugador n+1 *)
      FOR dia:=1 TO n DO
         IF tabla[jug,dia]=n+1 THEN tabla[jug,dia]:=0 END
      END
   END
ELSE (* n par *)
   m:=n DIV 2;
   Torneo(m, tabla); (* primero el cuadrante sup. izq. *)
   IF (m MOD 2 )=0 THEN  (* m par *)
      FOR jug:=m+1 TO n DO  (* cuadrante inferior izquierdo *)
         FOR dia:=1 TO m-1 DO
            tabla[jug,dia]:=tabla[jug-m,dia]+m
         END
      END;
      FOR jug:=1 TO m DO (* cuadrante superior derecho *)
         FOR dia:=m TO n-1 DO
            IF (jug+dia)<=n THEN tabla[jug,dia]:=jug+dia
            ELSE tabla[jug,dia]:=jug+dia-m
            END
         END
      END;
      FOR jug:=m+1 TO n DO  (* y cuadrante inferior derecho *)
         FOR dia:=m TO n-1 DO
            IF jug>dia THEN tabla[jug,dia]:=jug-dia
            ELSE tabla[jug,dia]:=(jug+m)-dia
            END
         END
      END
   ELSE (* m impar *)
      FOR jug:=m+1 TO n DO  (* cuadrante inferior izquierdo *)
         FOR dia:=1 TO m DO
            IF tabla[jug-m,dia]=0 THEN tabla[jug,dia]:=0
            ELSE tabla[jug,dia]:=tabla[jug-m,dia]+m
            END
         END
      END;
      FOR jug:=1 TO m DO (* ceros de los cuadrantes izq *)
         FOR dia:=1 TO m DO
            IF tabla[jug,dia]=0 THEN
               tabla[jug,dia]:=jug+m;
               tabla[jug+m,dia]:=jug
            END
```

```
            END
         END;
         FOR jug:=1 TO m DO (* cuadrante superior derecho *)
            FOR dia:=m+1 TO n-1 DO
               IF (jug+dia)<=n THEN tabla[jug,dia]:=jug+dia
               ELSE tabla[jug,dia]:=jug+dia-m
               END
            END
         END;
         FOR jug:=m+1 TO n DO  (* ultimo, cuadrante inf. der. *)
            FOR dia:=m+1 TO n-1 DO
               IF jug>dia THEN tabla[jug,dia]:=jug-dia
               ELSE tabla[jug,dia]:=(jug+m)-dia
               END
            END
         END
      END (* IF m impar *)
   END (* IF n par *)
END Torneo;
```

Este algoritmo puede reducirse en extensión pero a costa de perder claridad en los casos tratados y en el manejo de los índices que rellenan la tabla solución. Hemos preferido mantener la presente versión para una mejor legibilidad del algoritmo.

Por otro lado, este es un ejemplo de algoritmo Divide y Vencerás de simplificación, esto es, en donde el problema se reduce en cada paso a un solo problema de tamaño más pequeño, cuyo proceso de combinación no es trivial.

3.14 DIVIDE Y VENCERÁS MULTIDIMENSIONAL

Una generalización de la técnica que estamos estudiando en este capítulo es el Divide y Vencerás multidimensional, la cual trata de resolver un problema de tamaño n en un espacio k-dimensional mediante la solución de dos subproblemas de tamaño $n/2$ en un espacio k-dimensional y un problema de tamaño n en un espacio $(k–1)$-dimensional. Para ilustrar esta técnica vamos a considerar el siguiente problema:

En un espacio discreto bidimensional $[1..M]x[1..M]$ tenemos un conjunto S de n puntos. Diremos que un punto $P=(p_1,p_2)$ *domina* a otro punto $Q=(q_1,q_2)$ si $p_1>q_1$ y $p_2>q_2$. El *rango* de un punto P de S es el número de puntos que domina. Implementar un algoritmo que calcule el rango de todos los puntos del conjunto S.

Solución ()

Una primera solución al problema consiste en calcular el rango para cada punto comparándolo con los $n–1$ restantes, y da lugar al siguiente algoritmo:

```
CONST M =...; (* dimension del espacio *)
```

```
      n =...; (* numero de puntos *)
TYPE  punto = RECORD x,y:[1..M] END;
      cjt_puntos = ARRAY[1..n] OF punto;
      rango = ARRAY[1..n] OF CARDINAL;

PROCEDURE CalculaRango(s:cjt_puntos;prim,ult:CARDINAL;VAR r:rango);
(* calcula en r el rango de los puntos en el conj. s[prim..ult] *)
   VAR i,j:CARDINAL;
BEGIN
   FOR i:=prim TO ult DO
      r[i]:=0;
      FOR j:=prim TO ult DO
         IF Domina(s[i],s[j]) THEN INC(r[i]) END;
      END;
   END;
END CalculaRango;
```

Este procedimiento usa una función que decide cuándo un punto domina a otro:

```
PROCEDURE Domina(p,q:punto):BOOLEAN;
BEGIN
   RETURN (p.x>q.x) AND (p.y>q.y)
END Domina;
```

La complejidad de la función *CalculaRango* es claramente de orden $O(n^2)$ por tratarse de dos bucles anidados en donde sólo se realizan operaciones de orden constante. Sin embargo, existe un método de menor complejidad utilizando Divide y Vencerás multidimensional, originalmente expuesto en [BEN80].

En primer lugar se escoge una línea vertical L que divide al conjunto de puntos S en dos subconjuntos A y B, cada uno con la mitad de los puntos (la ecuación de esta recta no es sino $x = med$, siendo *med* la mediana del conjunto de abcisas de los puntos de S).

El segundo paso del método calcula recursivamente el rango de los dos subconjuntos.

Por último, el tercer paso combina los resultados obtenidos para componer la solución del problema. Para esto hemos de fijarnos en dos hechos:

a) Primero, que ningún punto de A domina a uno de B (pues la abcisa de un punto de A nunca es mayor que la de cualquier punto de B).
b) Segundo, que un punto P de B va a dominar a otro Q de A si y sólo si la ordenada de P es mayor que la de Q.

Por el primero de ellos, el rango de los puntos de A coincide con el rango que van a tener cuando los consideremos puntos de S. Ahora bien, para calcular el rango final de los puntos de B necesitamos añadir al rango calculado para cada uno de ellos el número de puntos de A que cada uno domina. Para encontrar tal número lo que haremos es "proyectar" los puntos de S sobre la línea L. Una vez hecho esto,

basta recorrer tal línea de abajo arriba e ir acumulando el número de puntos de A que vayamos encontrando. Cuando se encuentre un punto de B, el número de puntos de A acumulado hasta ese momento será el número pedido.

Obsérvese cómo este método sigue la estrategia de Divide y Vencerás multidimensional. Para resolver un problema de tamano n en el plano resolvemos dos problemas de tamaño $n/2$ en el plano, y uno de tamaño n sobre una recta. Para el caso de la recta (dimensión 1), el cálculo del rango de cada uno de los puntos es fácil: basta con ordenarlos y el rango de un punto va a ser el número de puntos que le preceden.

Para implementar este algoritmo vamos a hacer dos suposiciones que no van a restar ninguna generalidad a la solución, pero que permiten simplificar el código. Lo primero que supondremos es que las abcisas de los puntos son todas distintas, y que están numeradas consecutivamente de 1 a n. La segunda es que el conjunto S está ordenado por los valores de las abcisas de sus puntos. Ninguna de ellas resta generalidad. La primera, porque podemos utilizar claves distintas para referenciar las abcisas de los puntos. Respecto a la segunda, podemos ordenar el conjunto S antes de invocar a este algoritmo, lo que únicamente conlleva una complejidad adicional de orden O($n$log$n$).

Tales suposiciones nos van a permitir encontrar la línea L fácilmente (la mediana es el elemento en posición $(n+1)\div 2$ del vector), y recorrerla posteriormente de forma creciente sin problemas. Este esquema da lugar al siguiente procedimiento:

```
PROCEDURE CalculaRango2(s:cjt_puntos;prim,ult:CARDINAL;
                        VAR r:rango);
(* calcula en r el rango de los puntos en s[prim..ult] *)
(* supone que los puntos estan ordenados respecto a sus abcisas,
   y que estas coinciden con prim,prim+1,...,ult. *)
   VAR i,j,mitad,suma_A:CARDINAL;
       s_y:cjt_puntos;
BEGIN
   IF prim=ult THEN (* caso base *)
      r[prim]:=0; RETURN
   END;
(* paso 2: resolver el problema para los subconjuntos A y B *)
   mitad:=(prim+ult) DIV 2;
   CalculaRango2(s,prim,mitad,r);
   CalculaRango2(s,mitad+1,ult,r);
   (* paso 3: ordenamos s respecto a su ordenada *)
   s_y:=s; (* utilizamos una copia de s para esto *)
   Ordenar_Y(s_y,prim,ult);
   (* paso 4: y ahora recorremos la linea imaginaria L *)
   suma_A:=0;
   FOR i:=prim TO ult DO
      IF s_y[i].x<=mitad THEN INC(suma_A) (* el punto es de A *)
      ELSE INC(r[s_y[i].x],suma_A);       (* el punto es de B *)
      END;
```

```
    END;
END CalculaRango2;
```

El procedimiento *Ordenar_Y(VAR a:cjt_puntos; prim,ult:CARDINAL)* ordena el conjunto de puntos *a*[*prim..ult*] respecto a su ordenada.

Para analizar el tiempo de ejecución $T(n)$ del procedimiento *CalculaRango2*, calcularemos la complejidad de cada uno de sus pasos:

- El caso base se resuelve con 4 operaciones elementales; es por tanto O(1).
- El paso 2 tiene un tiempo de ejecución $2T(n/2)+O(1)$.
- El paso 3 conlleva una copia del vector y una ordenación, es decir: $O(n)+O(n\log n)$
- Por último, el orden de complejidad del paso 4 es $O(n)$.

Por consiguiente, el tiempo de ejecución viene dado por la ecuación en recurrencia

$$T(n) = O(1)+2T(n/2)+O(1)+O(n)+O(n\log n)+O(n) = 2T(n/2)+O(n\log n),$$

cuya solución es de un orden de complejidad $O(n\log^2 n)$. Para ver esto, podemos expresar $T(n)$ como:

$$T(n) = 2T(n/2) + An\log n,$$

siendo A una constante. Llamando $n = 2^k$ (o lo que es igual, $k = \log n$) y haciendo el cambio $t_k = T(2^k)$, obtenemos

$$t_k = 2t_{k-1} + Ak2^k$$

ecuación en recurrencia no homogénea cuya ecuación característica es $(x-2)^3 = 0$, lo que implica que t_k viene dado por la expresión:

$$t_k = c_1 2^k + c_2\, k2^k + c_3 k^2 2^k.$$

Deshaciendo los cambios realizados con anterioridad obtenemos finalmente:

$$T(n) = c_1 n + c_2 n\log n + c_3 n\log^2 n \in O(n\log^2 n).$$

Este algoritmo mejora notablemente el presentado al principio de este apartado. Sin embargo, tras un estudio de *CalculaRango2* podemos observar que su complejidad esta dominada por la complejidad de la ordenación que se realiza en su tercer paso. ¿Existe alguna forma de evitar esta ordenación?

La respuesta es afirmativa, y además da lugar a una mejora usual de este tipo de algoritmos. Se basa en ordenar sólo una vez (al principio) el conjunto S respecto a las ordenadas de sus puntos, y mantener esta ordenación cuando se divida el conjunto inicial en dos. Esto permite simplificar el paso 3, lo que hace que el tiempo de ejecución del algoritmo sea ahora de $T(n) = 2T(n/2) + O(n)$. Esta ecuación es de una complejidad $O(n\log n)$, como hemos calculado en varios de los problemas de este capítulo.

Cara a implementar esta solución, lo que vamos a necesitar es una estructura auxiliar que nos permita decidir en cualquier momento a qué conjunto (A o B) pertenece un punto de *S*. Esto nos lleva al siguiente algoritmo:

```
PROCEDURE CalculaRango3(sX,sY:cjt_puntos;prim,ult:CARDINAL;
                        VAR r:rango);
(* calcula el rango de los puntos en el conjunto sX[prim..ult] *)
(* supone que los puntos de sX estan ordenados respecto a sus
   abcisas, que estas coinciden con prim,prim+1,...,ult, y que los
   puntos de sY estan ordenados respecto a sus ordenadas *)

   VAR i,j,mitad,suma_A:CARDINAL;s:cjt_puntos;
BEGIN
   IF prim=ult THEN (* caso base *)
      r[prim]:=0; RETURN
   END;
   (* paso 2: resolvemos el problema para los subconjuntos A y B *)
   mitad:=(prim+ult) DIV 2;
   CalculaRango3(sX,sY,prim,mitad,r);
   CalculaRango3(sX,sY,mitad+1,ult,r);
   (* en el paso 3 seleccionamos los elementos adecuados de sY *)
   i:=1;j:=prim;
   WHILE (j<=ult) DO
      IF (prim<=sY[i].x) AND (sY[i].x<=ult) THEN
         s[j]:=sY[i]; INC(j)
      END;
      INC(i)
   END;
   (* paso 4: y ahora recorremos la linea imaginaria L *)
   suma_A:=0;
   FOR i:=prim TO ult DO
      IF s[i].x<=mitad THEN INC(suma_A)     (* el punto es de A *)
      ELSE INC(r[s[i].x],suma_A)          (* el punto es de B *)
      END;
   END;
END CalculaRango3;
```

3.15 LA SUBSECUENCIA DE SUMA MÁXIMA

Dados n enteros cualesquiera $a_1,a_2,...,a_n$, necesitamos encontrar el valor de la expresión:

$$\max_{1\le i\le j\le n}\left\{\sum_{k=i}^{j} a_k\right\},$$

que calcula el máximo de las sumas parciales de elementos consecutivos. Como ejemplo, dados los 6 enteros (–2,11,–4,13,–5,–2) la solución al problema es 20 (suma de a_2 hasta a_4).

Deseamos implementar un algoritmo Divide y Vencerás de complejidad $n\log n$ que resuelva el problema. ¿Existe algún otro algoritmo que lo resuelva en menor tiempo?

Solución (✍)

Existe una solución trivial a este problema, basada en calcular todas las posibles sumas y escoger la de valor máximo (esto es, mediante un algoritmo de "fuerza bruta") cuyo orden de complejidad es $O(n^3)$. Esto lo hace bastante ineficiente para valores grandes de n:

```
PROCEDURE Sumamax(VAR a:vector;prim,ult:CARDINAL):CARDINAL;
   VAR izq,der,i:CARDINAL; max_aux,suma:INTEGER;
BEGIN
   max_aux:=0;
   FOR izq:=prim TO ult DO
      FOR der:=izq TO ult DO
         suma:=0;
         FOR i:=izq TO der DO
            suma:=suma+a[i]
         END;
         IF suma>max_aux THEN
            max_aux:=suma
         END
      END
   END;
   RETURN max_aux
END Sumamax;
```

Una mejora inmediata para el algoritmo es la de evitar calcular la suma para cada posible subsecuencia, aprovechando el valor ya calculado de la suma de los valores de $a[izq..der–1]$ para calcular la suma de los valores de $a[izq..der]$. Esto da lugar a la siguiente función

```
PROCEDURE Sumamax2(VAR a:vector;prim,ult:CARDINAL):CARDINAL;
   VAR izq,der:CARDINAL; max_aux,suma:INTEGER;
BEGIN
   max_aux:=0;
   FOR izq:=prim TO ult DO
      suma:=0;
      FOR der:=izq TO ult DO
         suma:=suma+a[der];
```

```
        (* suma contiene la suma de a[izq..der] *)
        IF suma>max_aux THEN max_aux:=suma END
      END
    END;
    RETURN max_aux
END Sumamax2;
```

cuya complejidad es de orden O(n^2), lo cual mejora sustancialmente al anterior, pero que aún no consigue la complejidad pedida.

Utilizaremos ahora la técnica de Divide y Vencerás para intentar mejorar la eficiencia de los algoritmos anteriores, y lo haremos siguiendo una idea de [BEN89]. Para ello, dividiremos el problema en tres subproblemas más pequeños, sobre cuyas soluciones construiremos la solución total.

En este caso la subsecuencia de suma máxima puede encontrarse en uno de tres lugares. O está en la primera mitad del vector, o en la segunda, o bien contiene al punto medio del vector y se encuentra en ambas mitades. Las tres soluciones se combinan mediante el cálculo de su máximo para obtener la suma pedida.

Los dos primeros casos pueden resolverse recursivamente. Respecto al tercero, podemos calcular la subsecuencia de suma máxima de la primera mitad que contenga al último elemento de esa primera mitad, y la subsecuencia de suma máxima de la segunda mitad que contenga al primer elemento de esa segunda mitad. Estas dos secuencias pueden concatenarse para construir la subsecuencia de suma máxima que contiene al elemento central de vector. Esto da lugar al siguiente algoritmo:

```
PROCEDURE Sumamax3(VAR a:vector;prim,ult:CARDINAL):CARDINAL;
   VAR mitad,i:CARDINAL;
       max_izq,max_der,suma,max_aux:INTEGER;
BEGIN
   (* casos base *)
   IF prim>ult THEN RETURN 0 END;
   IF prim=ult THEN RETURN Max2(0,a[prim]) END;
   mitad:=(prim+ult)DIV 2;
   (* casos 1 y 2 *)
   max_aux:=Max2(Sumamax3(a,prim,mitad),Sumamax3(a,mitad+1,ult));
   (* caso 3: parte izquierda *)
   max_izq:=0;
   suma:=0;
   FOR i:=mitad TO prim BY -1 DO
      suma:=suma+a[i];
      max_izq:=Max2(max_izq,suma)
   END;
   (* caso 3: parte derecha *)
   max_der:=0;
   suma:=0;
   FOR i:=mitad+1 TO ult DO
```

```
      suma:=suma+a[i];
      max_der:=Max2(max_der,suma)
   END;
   (* combinacion de resultados *)
   RETURN Max2(max_der+max_izq,max_aux)
END Sumamax3;
```

donde la función *Max2* utilizada es la que calcula el máximo de dos números enteros.

El procedimiento *Sumamax3* es de complejidad O($n$log$n$), puesto que su tiempo de ejecución $T(n)$ viene dado por la ecuación en recurrencia

$$T(n) = 2T(n/2) + An$$

con la condición inicial $T(1) = 7$, siendo A una constante.

Obsérvese además que este análisis es válido puesto que hemos añadido la palabra *VAR* al vector *a* en la definición del procedimiento. Si no, se producirían copias de *a* en las invocaciones recursivas, lo que incrementaría el tiempo de ejecución del procedimiento.

Respecto a la última parte del problema, necesitamos encontrar un algoritmo aún mejor que éste. La clave va a consistir en una modificación a la idea básica del algoritmo anterior, basada en un algoritmo del tipo "en línea" (véase el problema del elemento mayoritario, en este capítulo).

Supongamos que ya tenemos la solución del problema para el subvector $a[prim..i–1]$. ¿Cómo podemos extender esa solución para encontrar la solución de $a[prim..i]$? De forma análoga al razonamiento que hicimos para el algoritmo anterior, la subsecuencia de suma máxima de $a[prim..i]$ puede encontrarse en $a[prim..i–1]$, o bien contener al elemento $a[i]$.

Esto da lugar a la siguiente función:

```
PROCEDURE Sumamax4(VAR a:vector;prim,ult:CARDINAL):CARDINAL;
   VAR i:CARDINAL;
       suma,max_anterior,max_aux:INTEGER;
BEGIN
   max_anterior:=0;
   max_aux:=0;

   FOR i:=prim TO ult DO
      max_aux:=Max2(max_aux+a[i],0);
      max_anterior:=Max2(max_anterior,max_aux)
   END;
   RETURN max_anterior;
END Sumamax4;
```

Este algoritmo no es intuitivo ni fácil de entender a primera vista. La clave del algoritmo se encuentra en la variable *max_aux*, que representa el valor de la suma de la subsecuencia de suma máxima del subvector *a*[*prim..i*] que contiene al elemento *a*[*i*], y que se calcula a partir de su valor para el subvector *a*[*prim..i*–1]. Este valor se incrementa en *a*[*i*] mientras que esa suma sea positiva, pero vuelve a valer cero cada vez que se hace negativa, indicando que la subsecuencia de suma máxima que contiene al elemento *a*[*i*] es la subsecuencia vacía.

El algoritmo resultante es de complejidad lineal, y un análisis detallado de esta función puede encontrase en [GRI80].

Capítulo 4

ALGORITMOS ÁVIDOS

4.1 INTRODUCCIÓN

El método que produce algoritmos ávidos es un método muy sencillo y que puede ser aplicado a numerosos problemas, especialmente los de optimización.

Dado un problema con *n* entradas el método consiste en obtener un subconjunto de éstas que satisfaga una determinada restricción definida para el problema. Cada uno de los subconjuntos que cumplan las restricciones diremos que son soluciones *prometedoras*. Una solución prometedora que maximice o minimice una función objetivo la denominaremos solución óptima.

Como ayuda para identificar si un problema es susceptible de ser resuelto por un algoritmo ávido vamos a definir una serie de elementos que han de estar presentes en el problema:

- Un conjunto de *candidatos*, que corresponden a las *n* entradas del problema.
- Una *función de selección* que en cada momento determine el candidato idóneo para formar la solución de entre los que aún no han sido seleccionados ni rechazados.
- Una función que compruebe si un cierto subconjunto de candidatos es *prometedor*. Entendemos por prometedor que sea posible seguir añadiendo candidatos y encontrar una solución.
- Una *función objetivo* que determine el valor de la solución hallada. Es la función que queremos maximizar o minimizar.
- Una función que compruebe si un subconjunto de estas entradas es solución al problema, sea óptima o no.

Con estos elementos, podemos resumir el funcionamiento de los algoritmos ávidos en los siguientes puntos:

1. Para resolver el problema, un algoritmo ávido tratará de encontrar un subconjunto de candidatos tales que, cumpliendo las restricciones del problema, constituya la solución óptima.
2. Para ello trabajará por etapas, tomando en cada una de ellas la decisión que le parece la mejor, sin considerar las consecuencias futuras, y por tanto escogerá

de entre todos los candidatos el que produce un óptimo local para esa etapa, suponiendo que será a su vez óptimo global para el problema.

3. Antes de añadir un candidato a la solución que está construyendo comprobará si es prometedora al añadilo. En caso afirmativo lo incluirá en ella y en caso contrario descartará este candidato para siempre y no volverá a considerarlo.
4. Cada vez que se incluye un candidato comprobará si el conjunto obtenido es solución.

Resumiendo, los algoritmos ávidos construyen la solución en etapas sucesivas, tratando siempre de tomar la decisión óptima para cada etapa. A la vista de todo esto no resulta difícil plantear un esquema general para este tipo de algoritmos:

```
PROCEDURE AlgoritmoAvido(entrada:CONJUNTO):CONJUNTO;
   VAR x:ELEMENTO; solucion:CONJUNTO; encontrada:BOOLEAN;
BEGIN
   encontrada:=FALSE; crear(solucion);
   WHILE NOT EsVacio(entrada) AND (NOT encontrada) DO
      x:=SeleccionarCandidato(entrada);
      IF EsPrometedor(x,solucion) THEN
         Incluir(x,solucion);
         IF EsSolucion(solucion) THEN
            encontrada:=TRUE
         END;
      END
   END;
   RETURN solucion;
END AlgoritmoAvido;
```

De este esquema se desprende que los algoritmos ávidos son muy fáciles de implementar y producen soluciones muy eficientes. Entonces cabe preguntarse ¿por qué no utilizarlos siempre? En primer lugar, porque no todos los problemas admiten esta estrategia de solución. De hecho, la búsqueda de óptimos locales no tiene por qué conducir siempre a un óptimo global, como mostraremos en varios ejemplos de este capítulo. La estrategia de los algoritmos ávidos consiste en tratar de ganar todas las batallas sin pensar que, como bien saben los estrategas militares y los jugadores de ajedrez, para ganar la guerra muchas veces es necesario perder alguna batalla.

Desgraciadamente, y como en la vida misma, pocos hechos hay para los que podamos afirmar sin miedo a equivocarnos que lo que parece bueno para hoy siempre es bueno para el futuro. Y aquí radica la dificultad de estos algoritmos. Encontrar la función de selección que nos garantice que el candidato escogido o rechazado en un momento determinado es el que ha de formar parte o no de la solución óptima sin posibilidad de reconsiderar dicha decisión. Por ello, una parte muy importante de este tipo de algoritmos es la demostración formal de que la función de selección escogida consigue encontrar óptimos globales para cualquier entrada del algoritmo. No basta con diseñar un procedimiento ávido, que seguro

que será rápido y eficiente (en tiempo y en recursos), sino que hay que demostrar que siempre consigue encontrar la solución óptima del problema.

Debido a su eficiencia, este tipo de algoritmos es muchas veces utilizado aun en los casos donde se sabe que no necesariamente encuentran la solución óptima. En algunas ocasiones la situación nos obliga a encontrar pronto una solución razonablemente buena, aunque no sea la óptima, puesto que si la solución óptima se consigue demasiado tarde, ya no vale para nada (piénsese en el localizador de un avión de combate, o en los procesos de toma de decisiones de una central nuclear). También hay otras circunstancias, como veremos en el capítulo dedicado a los algoritmos que siguen la técnica de Ramificación y Poda, en donde lo que interesa es conseguir cuanto antes una solución del problema y, a partir de la información suministrada por ella, conseguir la óptima más rápidamente. Es decir, la eficiencia de este tipo de algoritmos hace que se utilicen aunque no consigan resolver el problema de optimización planteado, sino que sólo den una solución "aproximada".

El nombre de algoritmos ávidos, también conocidos como voraces (su nombre original proviene del término inglés *greedy*) se debe a su comportamiento: en cada etapa "toman lo que pueden" sin analizar consecuencias, es decir, son glotones por naturaleza. En lo que sigue veremos un conjunto de problemas que muestran cómo diseñar algoritmos ávidos y cuál es su comportamiento. En este tipo de algoritmos el proceso no acaba cuando disponemos de la implementación del procedimiento que lo lleva a cabo. Lo importante es la demostración de que el algoritmo encuentra la solución óptima en todos los casos, o bien la presentación de un contraejemplo que muestra los casos en donde falla.

4.2 EL PROBLEMA DEL CAMBIO

Suponiendo que el sistema monetario de un país está formado por monedas de valores $v_1,v_2,...,v_n$, el problema del cambio de dinero consiste en descomponer cualquier cantidad dada M en monedas de ese país utilizando el menor número posible de monedas.

En primer lugar, es fácil implementar un algoritmo ávido para resolver este problema, que es el que sigue el proceso que usualmente utilizamos en nuestra vida diaria. Sin embargo, tal algoritmo va a depender del sistema monetario utilizado y por ello vamos a plantearnos dos situaciones para las cuales deseamos conocer si el algoritmo ávido encuentra siempre la solución óptima:

a) Suponiendo que cada moneda del sistema monetario del país vale al menos el doble que la moneda de valor inferior, que existe una moneda de valor unitario, y que disponemos de un número ilimitado de monedas de cada valor.
b) Suponiendo que el sistema monetario está compuesto por monedas de valores 1, p, p^2, p^3,..., p^n, donde $p > 1$ y $n > 0$, y que también disponemos de un número ilimitado de monedas de cada valor.

Solución (𝒮)

Comenzaremos con la implementación de un algoritmo ávido que resuelve el problema del cambio de dinero:

```
TYPE  MONEDAS =(M500,M200,M100,M50,M25,M5,M1);(*sistema monetario*)
      VALORES = ARRAY MONEDAS OF CARDINAL; (* valores de monedas *)
      SOLUCION = ARRAY MONEDAS OF CARDINAL;

PROCEDURE Cambio(n:CARDINAL;VAR valor:VALORES;VAR cambio:SOLUCION);
(* n es la cantidad a descomponer, y el vector "valor" contiene los
valores de cada una de las monedas del sistema monetario *)
   VAR moneda:MONEDAS;
BEGIN
   FOR moneda:=FIRST(MONEDAS) TO LAST(MONEDAS) DO
      cambio[moneda]:=0
   END;
   FOR moneda:=FIRST(MONEDAS) TO LAST(MONEDAS) DO
      WHILE valor[moneda]<=n DO
         INC(cambio[moneda]);
         DEC(n,valor[moneda])
      END
   END
END Cambio;
```

Este algoritmo es de complejidad lineal respecto al número de monedas del país, y por tanto muy eficiente.

Respecto a las dos cuestiones planteadas, comenzaremos por la primera. Supongamos que nuestro sistema monetario esta compuesto por las siguientes monedas:

```
TYPE MONEDAS = (M11,M5,M1);  valor:={11,5,1};
```

Tal sistema verifica las condiciones del enunciado pues disponemos de moneda de valor unitario, y cada una de ellas vale más del doble de la moneda inmediatamente inferior.

Consideremos la cantidad $n = 15$. El algoritmo ávido del cambio de monedas descompone tal cantidad en:

$$15 = 11 + 1 + 1 + 1 + 1,$$

es decir, mediante el uso de cinco monedas. Sin embargo, existe una descomposición que utiliza menos monedas (exactamente tres):

$$15 = 5 + 5 + 5.$$

Aunque queda comprobado que bajo estas circunstancias el diseño ávido no puede utilizarse, las razones por las que el algoritmo falla quedarán al descubierto cuando analicemos el siguiente punto.

b) En cuanto a la segunda situación, y para demostrar que el algoritmo ávido encuentra la solución óptima, vamos a apoyarnos en una propiedad general de los números naturales:

Si p es un número natural mayor que 1, todo número natural x puede expresarse de forma única como:

$$x = r_0 + r_1 p + r_2 p^2 + ... + r_n p^n, \qquad [4.1]$$

con $0 \le r_i < p$ para todo $0 \le i \le n$ y siendo n el menor natural tal que $x < p^{n+1}$, es decir, $n = \lfloor \log_p x \rfloor$.

El algoritmo del cambio de monedas lo que hace en nuestro caso es calcular los r_i, que indican el número de monedas a devolver de valor p^i ($0 \le i \le n$). Lo que tenemos que demostrar es que esa descomposición es óptima, esto es, que si

$$x = s_0 + s_1 p + s_2 p^2 + ... + s_m p^m$$

es otra descomposición distinta, entonces:

$$\sum_{i=0}^{n} r_i < \sum_{i=0}^{m} s_i \ .$$

Para realizar esta demostración lo haremos primero para $p = 2$ porque intuitivamente resulta más sencillo de entender el proceso de la demostración. El caso general resulta ser análogo.

Sea entonces

$$x = r_0 + 2r_1 + 2^2 r_2 + ... + 2^n r_n \qquad [4.2]$$

la descomposición obtenida por el algoritmo ávido. Por tanto $x < 2^{n+1}$ y los coeficientes r_i toman los valores 0 ó 1. Consideremos además otra descomposición distinta:

$$x = s_0 + 2s_1 + 2^2 s_2 + ... + 2^m s_m.$$

Paso 1:

En primer lugar, como se verifica que $x < 2^{n+1}$, esto implica que $m \le n$. Definimos entonces $s_{m+1} = s_{m+2} = ... = s_n = 0$ para poder disponer de n términos en cada descomposición.

Paso 2:

Queremos ver que

$$r_0 + r_1 + ... + r_n < s_0 + s_1 + ... + s_n.$$

Como ambas descomposiciones son distintas, sea k el primer índice tal que $r_k \ne s_k$. Podemos suponer sin perder generalidad que $k = 0$, puesto que si no lo fuera podríamos restar a ambos lados de la desigualdad los términos iguales y dividir por la potencia de 2 adecuada. Veamos que si $r_0 \ne s_0$ entonces $r_0 < s_0$.

- Si x es par entonces $r_0 = 0$. Como $s_0 \ge 0$ y estamos suponiendo que $r_0 \ne s_0$, s_0 ha de ser mayor que cero y por tanto podemos deducir que $r_0 < s_0$.

- Si x es impar entonces $r_0 = 1$. Pero en la segunda descomposición de x también ha de haber al menos una moneda de una unidad, y por tanto $s_0 \geq 1$. Al estar suponiendo que $r_0 \neq s_0$, podemos deducir también aquí que $r_0 < s_0$.

Con esto, consideremos la cantidad $s_0 - r_0 > 0$. Tal cantidad ha de ser par pues $x - r_0$ lo es (por la expresión [4.2]). Y por ser par, siempre podremos "mejorar" la segunda descomposición $(s_0,s_1,...,s_n)$ cambiando $s_0 - r_0$ monedas de 1 unidad por $(s_0 - r_0)/2$ monedas de 2 unidades, obteniendo:

$$s_0 + s_1 + \ldots + s_n > r_0 + \left(s_1 + \frac{s_0 - r_0}{2} \right) + s_2 + \ldots + s_n . \qquad [4.3]$$

Paso 3:

Mediante el razonamiento anterior hemos obtenido una nueva descomposición, mejor que la segunda, y manteniendo además que:

$$r_0 + \left(s_1 + \frac{s_0 - r_0}{2} \right) 2 + s_2 2^2 + \ldots + s_n 2^n = x = r_0 + r_1 2 + \ldots + r_n 2^n .$$

Podemos volver a aplicar el razonamiento del paso 2 sobre esta nueva descomposición, y así sucesivamente ir viendo que $s_i \geq r_i$ para todo $0 \leq i \leq n-1$, e ir obteniendo nuevas descomposiciones, cada una mejor que la anterior, hasta llegar en el último paso a una descomposición de la forma:

$$r_0 + r_1 2 + r_2 2^2 + \ldots + r_{n-1} 2^{n-1} + \left(s_n + \frac{s_{n-1} - acum_{n-1}}{2} \right) 2^n = x \qquad [4.4]$$

en la que hemos ido acumulando las diferencias en el último término, y que además verifica que:

$$r_0 + r_1 + \ldots + r_i + \left(s_{i+1} + \frac{s_i - acum_i}{2} \right) + \ldots + s_n \geq r_0 + r_1 + \ldots + r_n , \; (0 \leq i \leq n-1)$$

Una vez llegado a este punto la demostración está ya realizada, puesto que si se verifica [4.4], por la unicidad de la descomposición a la que hacía referencia la propiedad [4.1], se ha de cumplir que

$$\left(s_n + \frac{s_{n-1} - acum_{n-1}}{2} \right) = r_n ,$$

y esto, junto a la cadena de desigualdades [4.3], hace que sea cierta nuestra afirmación. Para el caso $p > 2$ el razonamiento es igual.

Resta sólo preguntarnos por qué esta demostración no funciona para cualquier sistema monetario. La razón fundamental se encuentra en la expresión [4.4], que en este sistema permite pasar monedas de una unidad a otra sin problemas, no siendo válido para todos.

4.3 RECORRIDOS DEL CABALLO DE AJEDREZ

Dado un tablero de ajedrez y una casilla inicial, queremos decidir si es posible que un caballo recorra todos y cada uno de los escaques sin duplicar ninguno. No es necesario en este problema que el caballo vuelva al escaque de partida. Un posible algoritmo ávido decide, en cada iteración, colocar el caballo en la casilla desde la cual domina el menor número posible de casillas aún no visitadas.

a) Implementar dicho algoritmo a partir de un tamaño de tablero $n \times n$ y una casilla inicial (x_0,y_0).

b) Buscar, utilizando el algoritmo realizado en el apartado anterior, todas las casillas iniciales para los que el algoritmo encuentra solución.

c) Basándose en los resultados del apartado anterior, encontrar el patrón general de las soluciones del recorrido del caballo.

Solución (☺/😐)

a) Para implementar el algoritmo pedido comenzaremos definiendo las constantes y tipos que utilizaremos:

```
CONST TAMMAX = ...; (* dimension maxima del tablero *)
TYPE tablero = ARRAY[1..TAMMAX],[1..TAMMAX] OF CARDINAL;
```

Cada una de las casillas del tablero va a almacenar un número natural que indica el número de orden del movimiento del caballo en el que visita la casilla. Podrá tomar también el valor cero, indicando que la casilla no ha sido visitada aún. Inicialmente todas las casillas tomarán este valor.

Una posible implementación del algoritmo viene dada por la función *Caballo* que se muestra a continuación, la cual, dado un tablero *t*, su dimensión *n* y una posición inicial (x,y), decide si el caballo recorre todo el tablero o no.

```
PROCEDURE Caballo(VAR t:tablero; n:CARDINAL; x,y:CARDINAL):BOOLEAN;
   VAR i:CARDINAL;
BEGIN
   InicTablero(t,n); (* inicializa las casillas del tablero a 0 *)
   FOR i:=1 TO n*n DO
      t[x,y]:=i;
      IF NOT NuevoMov(t,n,x,y) AND (i<n*n-1) THEN RETURN FALSE END;
   END;
   RETURN TRUE; (* hemos recorrido las n*n casillas *)
END Caballo;
```

La función *NuevoMov* es la que va a ir calculando la nueva casilla a la que salta el caballo siguiendo la indicación del enunciado, devolviendo *FALSE* si no puede moverse:

```
PROCEDURE NuevoMov(VAR t:tablero; n:CARDINAL; VAR x,y:CARDINAL)
                                                        :BOOLEAN;
```

```
   VAR accesibles,minaccesibles:CARDINAL;
       i,solx,soly,nuevax,nuevay:CARDINAL;
BEGIN
   minaccesibles:=9;
   solx:=x; soly:=y;
   FOR i:=1 TO 8 DO
      IF Salto(t,n,i,x,y,nuevax,nuevay) THEN
         accesibles:=Cuenta(t,n,nuevax,nuevay);
         IF (accesibles>0) AND (accesibles<minaccesibles) THEN
            minaccesibles:=accesibles;
            solx:=nuevax; soly:=nuevay;
         END
      END
   END;
   x:=solx; y:=soly;
   RETURN (minaccesibles<9);
END NuevoMov;
```

Para su implementación necesitamos dos funciones auxiliares: *Salto* y *Cuenta*. La primera calcula las coordenadas de la casilla a donde salta el caballo (tiene 8 posibilidades), y devuelve si es posible realizar ese movimiento o no (puede estar ocupada o bien salirse del tablero):

```
PROCEDURE Salto(VAR t:tablero;n:CARDINAL;i:CARDINAL;
                x,y:CARDINAL;VAR nx,ny:CARDINAL) :BOOLEAN;
   (* i indica el numero del movimiento, (x,y) es la casilla
      actual, y (nx,ny) es la casilla a donde salta. *)
BEGIN
   CASE i OF
      |1: nx:=x-2; ny:=y+1;  |2: nx:=x-1; ny:=y+2;
      |3: nx:=x+1; ny:=y+2;  |4: nx:=x+2; ny:=y+1;
      |5: nx:=x+2; ny:=y-1;  |6: nx:=x+1; ny:=y-2;
      |7: nx:=x-1; ny:=y-2;  |8: nx:=x-2; ny:=y-1;
   END;
   RETURN((1<=nx)AND(nx<=n)AND(1<=ny)AND(ny<=n)AND(t[nx,ny]=0));
END Salto;
```

Dicha función intenta los movimientos en el orden que muestra la siguiente figura:

	2		3	
1				4
		X		

8				5
	7		6	

La otra función es *Cuenta*, que devuelve el número de casillas a las que el caballo puede saltar desde una posición dada:

```
PROCEDURE Cuenta(VAR t:tablero;n:CARDINAL;x,y:CARDINAL):CARDINAL;
   VAR acc,i,nx,ny:CARDINAL;
BEGIN
   acc:=0;
   FOR i:=1 TO 8 DO
      IF Salto(t,n,i,x,y,nx,ny) THEN INC(acc) END
   END;
   RETURN acc;
END Cuenta;
```

Obsérvese que hemos utilizado el paso del tablero por referencia (mediante *VAR*) en vez de por valor en todos los procedimientos aunque no se modifique el vector, para evitar su copia en la pila.

b) Para resolver esta cuestión necesitamos un programa que nos permita ir recorriendo todas las posibilidades e imprimiendo aquellas casillas iniciales desde donde se consigue solución:

```
MODULE Caballos;
   ....
   VAR t:tablero; n,i,j:CARDINAL;
BEGIN
   FOR n:=4 TO TAMMAX DO
      WrStr('Dimension = '); WrCard(n,0); WrLn();
      FOR i:=1 TO n DO FOR j:=1 TO n DO
            IF Caballo(t,n,i,j) THEN
               WrStr(' Desde: '); WrCard(i,0); WrStr(',');
               WrCard(j,0); WrStr(' tiene solucion.'); WrLn();
            END
      END END;
      WrLn();
   END
END Caballos.
```

c) La salida del programa anterior nos permite inferir un patrón general para las soluciones del problema:

- Para $n = 4$, el problema no tiene solución.
- Para $n > 4$, n par, el problema tiene solución para cualquier casilla inicial.

- Para $n > 4$, n impar, el problema tiene solución para aquellas casillas iniciales (x_0,y_0) que verifiquen que $x_0 + y_0$ sea par, es decir, si el caballo comienza su recorrido en una escaque blanco.

Pero observemos que el algoritmo implementado no ha encontrado solución en todas estas situaciones. Por ejemplo, para $n = 5$, $x_0 = 5$ e $y_0 = 3$ el programa dice que no la hay. Sin embargo, sí la encuentra para $n = 5$, $x_0 = 1$ e $y_0 = 3$, para $n = 5$, $x_0 = 3$ e $y_0 = 1$ y para $n = 5$, $x_0 = 3$ e $y_0 = 5$, que son casos simétricos. De existir solución para alguno de ellos, por simetría se obtiene para los otros. ¿Por qué no la encuentra nuestro algoritmo?

La respuesta a esta pregunta se encuentra en cómo buscamos la siguiente casilla a donde saltar. Por la forma en la que funciona el programa, *siempre* probamos las ocho casillas en el sentido de las agujas del reloj, siguiendo la pauta mostrada en la función *Salto*. Esto hace que nuestro algoritmo no sea simétrico. En resumen, estamos ante un algoritmo ávido que no funciona para todos los casos.

4.4 LA DIVISIÓN EN PÁRRAFOS

Dada una secuencia de palabras p_1, p_2, ..., p_n de longitudes l_1, l_2, ..., l_n se desea agruparlas en líneas de longitud L. Las palabras están separadas por espacios cuya amplitud ideal (en milímetros) es b, pero los espacios pueden reducirse o ampliarse si es necesario (aunque sin solapamiento de palabras), de tal forma que una línea p_i, p_{i+1}, ..., p_j tenga exactamente longitud L. Sin embargo, existe una penalización por reducción o ampliación en el número total de espacios que aparecen o desaparecen. El *costo* de fijar la línea p_i, p_{i+1}, ..., p_j es $(j - i)|b' - b|$, siendo b' el ancho real de los espacios, es decir $(L - l_i - l_{i+1} - ... - l_j)/(j - i)$. No obstante, si $j = n$ (la última palabra) el costo será cero a menos que $b' < b$ (ya que no es necesario ampliar la última línea).

En primer lugar, necesitamos plantear un algoritmo ávido para resolver el problema, implementarlo y dar un ejemplo donde este algoritmo no encuentre solución óptima o bien demostrar que tal ejemplo no existe.

Por otra lado, consideraremos el caso especial de usar una impresora de líneas, en donde por sus características especiales el valor óptimo de b es 1 y no se puede producir reducción de espacios (ya que b no puede ser 0).

Solución (☺)

Para resolver este problema mediante un algoritmo ávido pensemos en lo que haríamos en la práctica para solucionarlo. En primer lugar, iríamos construyendo la línea empezando por la primera palabra y añadiendo las demás en orden, separándolas con espacios de tamaño óptimo b, hasta llegar a una palabra p_a (a>1) que no quepa en la línea, es decir:

$$l_1 + l_2 + ... + l_a + (a-1)*b > L.$$

Si ocurriera que $l_1 + l_2 + ... + l_a + (a-1)*b = L$, esto es, que la palabra encajara perfectamente en la línea, sencillamente imprimiríamos la línea y continuaríamos con la siguiente. Pero si no, necesitaríamos tomar una decisión: o se comprimen las palabras $p_1,...,p_{a-1}$ (recortando el tamaño de los espacios que las separan) para que

pueda caber también p_a en la línea; o bien se pasa la palabra p_a a la siguiente línea y se imprime la línea en curso, aumentando antes los espacios entre las palabras $p_1,...,p_{a-1}$ para que la línea tenga exactamente longitud L.

El algoritmo ávido simplemente va a escoger aquella opción que suponga un menor coste. Obsérvese que estamos ante un típico algoritmo ávido, pues siempre toma su decisión basado en una optimización local y nunca "guarda historia".

Para implementar tal algoritmo, vamos a disponer de un vector que almacena las longitudes de las palabras, y la solución vamos a darla como un vector de registros, uno por cada línea. Cada registro contiene los índices (número de orden) de las palabras que comienzan y terminan la línea, el tamaño de los espacios entre las palabras y el coste de la línea. Esto da lugar al siguiente algoritmo:

```
CONST MAXPALABRAS = ...;
      MAXLINEAS   = MAXPALABRAS; (* para cubrir el peor caso *)
TYPE  REGISTRO=  RECORD
                   primera,ultima:CARDINAL;
                   espacio,coste:REAL;
                 END;
      SOLUCION=  ARRAY [1..MAXLINEAS] OF REGISTRO;
      LONGPALS=  ARRAY [1..MAXPALABRAS] OF CARDINAL;

PROCEDURE Parrafo(L:CARDINAL;n:CARDINAL;b:CARDINAL;VAR l:LONGPALS;
                  VAR sol:SOLUCION):CARDINAL;
    (* L es la longitud de la linea, n el numero de palabras, b el
       tamano optimo de los espacios, l es el vector con las
       longitudes de las n palabras, y en sol almacena la solucion.
       Devuelve el numero de lineas que ha necesitado *)

   VAR  tamanopalabras:CARDINAL; (* long de palabras de la linea *)
        tamanolinea:CARDINAL; (* tamano de la linea en curso *)
        nlinea:CARDINAL;      (* linea en curso *)
        npalabra:CARDINAL;    (* palabra en curso *)
        nespacios:CARDINAL;   (* num. espacios linea en curso *)

   PROCEDURE Espacio(L,tampalabras,nesp:CARDINAL):REAL;
   (* devuelve cero si nesp = 0, o bien un numero mayor que 1 *)
   BEGIN
      IF nesp=0 THEN RETURN 0.0 END;
      RETURN REAL(L-tampalabras)/REAL(nesp);
   END Espacio;

   PROCEDURE ResetContadores(linea,npal:CARDINAL);
   BEGIN
      IF npal<=n THEN (* para la ultima palabra no hacemos nada *)
         sol[linea].primera:=npal;
         sol[linea].coste:=0.0;
```

```
        tamanopalabras:=l[npal];
        tamanolinea:=l[npal];
        nespacios:=0
      END;
    END ResetContadores;

    PROCEDURE Coste(L,b,tampalabras,nesp:CARDINAL):REAL;
      VAR bprima:REAL;
    BEGIN
      bprima:=Espacio(L,tampalabras,nesp);
      IF bprima>REAL(b) THEN RETURN REAL(nesp)*(bprima-REAL(b))
      ELSE RETURN REAL(nesp)*(REAL(b)-bprima)
      END;
    END Coste;

    PROCEDURE CerrarLinea(linea,npal:CARDINAL);
    BEGIN
      sol[linea].ultima:=npal-1;
      sol[linea].espacio:=Espacio(L,tamanopalabras,nespacios);
      sol[linea].coste:=Coste(L,b,tamanopalabras,nespacios);
    END CerrarLinea;
  BEGIN      (* programa principal del procedimiento Parrafo *)
    nlinea:=1;
    ResetContadores(nlinea,1); (* metemos la primera palabra *)
    npalabra:=2;
    WHILE (npalabra<=n) DO
      IF tamanolinea+b+l[npalabra]<=L THEN (* cabe *)
        INC(tamanolinea,b+l[npalabra]);
        INC(tamanopalabras,l[npalabra]);
        INC(nespacios)
      ELSE  (* no cabe de forma optima *)
        IF (tamanopalabras+l[npalabra]+nespacios+1)>L THEN
          (* no cabe en cualquier caso: la pasamos a otra linea *)
          CerrarLinea(nlinea,npalabra);
          INC(nlinea);       (* reinicializamos contadores *)
          ResetContadores(nlinea,npalabra);
        ELSE (* podria caber. Tenemos que tomar una decision *)
          IF Coste(L,b,tamanopalabras,nespacios)>=
          Coste(L,b,tamanopalabras+l[npalabra],nespacios+1) THEN
            INC(npalabra);  (* la metemos en la linea en curso *)
          END;
          (* si no, la pasamos a la otra linea *)
          CerrarLinea(nlinea,npalabra);
          INC(nlinea);
          ResetContadores(nlinea,npalabra);
        END
```

```
      END;
      INC(npalabra);
    END;
    IF sol[nlinea].primera=0 THEN RETURN nlinea-1 END;
    IF sol[nlinea].ultima=0 THEN CerrarLinea(nlinea,npalabra) END;
    RETURN nlinea;
  END Parrafo;
```

La complejidad de este algoritmo es de orden O(n), debido al bucle que se repite a lo más n–1 veces (una por cada palabra menos la primera y aquellas que decidamos meter comprimiendo la línea), y que dentro del bucle todas las operaciones que se realizan son de complejidad constante.

En cuanto a su funcionamiento, desafortunadamente no podemos afirmar que encuentre solución óptima en todos los casos, como pone de manifiesto el siguiente ejemplo.

Supongamos que L = 26, b = 2, y que disponemos de n = 7 palabras, cuyas longitudes son 10, 10, 4, 8, 10, 12 y 12.

El algoritmo anterior, tras meter las dos primeras palabras en la primera línea, tiene que tomar una decisión en cuanto a si la tercera palabra (de longitud 4) debe estar en la primera línea o no. En caso de estar, hay que comprimir los espacios, lo que ocasiona un coste de valor 2; por otro lado, si la pasa a la segunda línea es necesario expandir el espacio entre las dos palabras, lo que supone un coste de valor 4.

Ante esta disyuntiva, el algoritmo decide incluirla en la primera línea, lo que da lugar a la siguiente descomposición en líneas (expresadas con paréntesis):

(10, 10, 4), (8, 10), (12, 12).

El coste global de esta descomposición es 8 (=2+6+0), mientras que si hubiera tomado la alternativa que inicialmente tenía más coste hubiera llegado a la descomposición:

(10, 10), (4, 8, 10), (12, 12)

cuyo coste global es 4 (= 4+0+0).

El motivo del fallo de este algoritmo es su “glotonería”, como le ocurre a todos los algoritmos ávidos. En general este problema lo va a tener cualquier algoritmo que, sin disponer de posibilidades de decidir el orden en el que se van produciendo las entradas, no sea capaz de hacer sacrificios locales para obtener resultados globales óptimos.

En cuanto al segundo caso que se plantea en el enunciado de este problema, la situación es mucho más simple ya que no hay que tomar decisiones. O la palabra cabe, o si no hay que pasarla a la siguiente línea pues no se pueden comprimir los espacios entre palabras. El algoritmo que implementa tal estrategia puede obtenerse modificando el anterior:

```
PROCEDURE Parrafo2(L:CARDINAL;n:CARDINAL;VAR l:LONGPALS;
                   VAR sol:SOLUCION):CARDINAL;
```

```
    (* L es la longitud de la linea, n el numero de palabras, y l es
       el vector con las longitudes de las n palabras. Devuelve el
       numero de lineas que ha necesitado *)

  VAR  tamanopalabras:CARDINAL;
       (* longitud de las palabras de la linea hasta el momento *)
       tamanolinea:CARDINAL; (* tamano de la linea en curso *)
       nlinea:CARDINAL;      (* linea en curso *)
       npalabra:CARDINAL;    (* palabra en curso *)
       nespacios:CARDINAL;   (* num. espacios linea en curso *)

  PROCEDURE Espacio(L,tampalabras,nesp:CARDINAL):REAL;
  BEGIN
     IF nesp=0 THEN RETURN 0.0 END;
     RETURN REAL(L-tampalabras)/REAL(nesp);
  END Espacio;

  PROCEDURE ResetContadores(linea,npal:CARDINAL);
  BEGIN
     IF npal<=n THEN (* para la ultima palabra no hacemos nada *)
        sol[linea].primera:=npal;
        sol[linea].coste:=0.0;
        tamanopalabras:=l[npal];
        tamanolinea:=l[npal];
        nespacios:=0
     END;
  END ResetContadores;

  PROCEDURE Coste(L,tampalabras,nesp:CARDINAL):REAL;
  BEGIN
     RETURN REAL(nesp)*(Espacio(L,tampalabras,nesp)-1.0);
  END Coste;

  PROCEDURE CerrarLinea(linea,npal:CARDINAL);
  BEGIN
     sol[linea].ultima:= npal-1;
     sol[linea].espacio:= Espacio(L,tamanopalabras,nespacios);
     sol[linea].coste:= Coste(L,tamanopalabras,nespacios);
  END CerrarLinea;

BEGIN      (* programa principal del procedimiento Parrafo2 *)
  (* metemos la primera palabra *)
  nlinea:=1;ResetContadores(nlinea,1);
  npalabra:=2;
  WHILE (npalabra<=n) DO
```

```
      IF tamanolinea+1+l[npalabra]<=L THEN (* cabe *)
         INC(tamanolinea,1+l[npalabra]);
         INC(tamanopalabras,l[npalabra]);
         INC(nespacios)
      ELSE  (* no cabe *)
         CerrarLinea(nlinea,npalabra); INC(nlinea);
         ResetContadores(nlinea,npalabra);
      END;
      INC(npalabra);
   END;
   RETURN nlinea;
END Parrafo2;
```

4.5 LOS ALGORITMOS DE PRIM Y KRUSKAL

Partimos de un grafo conexo, ponderado y no dirigido $g = (V,A)$ de arcos no negativos, y deseamos encontrar el árbol de recubrimiento de g de coste mínimo. Por árbol de recubrimiento de un grafo g entendemos un subgrafo sin ciclos que contenga a todos sus vértices. En caso de haber varios árboles de coste mínimo, nos quedaremos de entre ellos con el que posea menos arcos.

Existen al menos dos algoritmos muy conocidos que resuelven este problema, como son el de Prim y el de Kruskal. En ambos se va construyendo el árbol por etapas, y en cada una se añade un arco. La forma en la que se realiza esa elección es la que distingue a ambos algoritmos.

El algoritmo de Prim comienza por un vértice y escoge en cada etapa el arco de menor peso que verifique que uno de sus vértices se encuentre en el conjunto de vértices ya seleccionados y el otro no. Al incluir un nuevo arco a la solución, se añaden sus dos vértices al conjunto de vértices seleccionados.

En el de Kruskal se ordenan primero los arcos por orden creciente de peso, y en cada etapa se decide qué hacer con cada uno de ellos. Si el arco no forma un ciclo con los ya seleccionados (para poder formar parte de la solución), se incluye en ella; si no, se descarta.

Nuestro objetivo en esta sección no es la de describir en detalle estos algoritmos desde el punto de vista de matemática discreta o la teoría de grafos, sino la de considerarlos desde la perspectiva de los algoritmos ávidos.

a) En primer lugar, nos planteamos la implementación de ambos algoritmos siguiendo el esquema descrito y el análisis de su complejidad (espacio y tiempo).
b) Estos algoritmos trabajan sobre grafos conexos. Nos preguntamos lo que ocurriría si por error se les suministrara un grafo no conexo.

Solución (✍)

a) Para conseguir una implementación sencilla de ambos algoritmos, supondremos que los vértices del grafo ponderado no dirigido $g = (V,A)$ están numerados de 1 a n, así que $V = \{1,2,3,...,n\}$, y que el conjunto de arcos A viene dado por su matriz

de adyacencia ponderada *g*, siendo *g*[*i*,*j*] el peso del arco (*i*,*j*) o bien ∞ si tal arco no existe. Por tanto, vamos a disponer de las siguientes definiciones;

```
CONST n = ...; (* numero de vertices *)
TYPE GRAFO = ARRAY [1..n],[1..n] OF BOOLEAN;
TYPE GRAFO_PONDERADO = ARRAY [1..n],[1..n] OF CARDINAL;
```

Para almacenar el árbol de recubrimiento mínimo (también llamado de expansión), utilizaremos la matriz de adyacencia de un grafo no ponderado.

Comenzaremos implementando el algoritmo de Kruskal, que necesita ordenar los arcos del grafo por orden creciente de peso:

```
PROCEDURE Kruskal(VAR g:GRAFO_PONDERADO; VAR sol:GRAFO);
   VAR  p:PARTICION;
        c1,c2:CARDINAL; (* indican componentes de la particion *)
        g2:GRAFO_ORDENADO;
        i,narcos:CARDINAL; (* numero de arcos del grafo *)
BEGIN
   InicParticion(p);
   narcos:=Ordenar(g,g2);     (* construye g2 a partir de g y *)
   i:=0;                      (* devuelve el numero de sus arcos *)
   WHILE (NOT FinParticion(p)) AND (i<narcos) DO
      (* recorremos todos los arcos *)
      INC(i);
      c1:=ObtenerComponente(p,g2[i].origen);
      c2:=ObtenerComponente(p,g2[i].destino);
      IF c1<>c2 THEN
         Fusionar(p,c1,c2);
         sol[g2[i].origen,g2[i].destino]:=TRUE
      END;
   END
END Kruskal;
```

Veamos los tipos y funciones auxiliares utilizados. En primer lugar, *PARTICION* es un vector que indica a qué componente conexa del grafo pertenece cada vértice, puesto que lo que hacemos es asignar cada vértice a una componente. Cada componente será identificada por el valor de su menor elemento.

```
TYPE PARTICION = ARRAY [1..n] OF CARDINAL;
```

Inicialmente disponemos de todos los vértices del grafo y ningún arco, por lo cual cada uno de los vértices está asignado a una partición distinta (la que constituye el propio vértice aislado). Conforme se van añadiendo los arcos en cada paso del algoritmo el número de particiones va disminuyendo, y los vértices van siendo asignados a las particiones correspondientes. Al incluir un arco que conecta dos particiones, a los elementos de la mayor partición se les asigna el valor de la menor.

Por otro lado, el algoritmo necesita ordenar los arcos del grafo según su peso. Para ello utiliza:

```
TYPE GRAFO_ORDENADO = ARRAY [1..n*(n-1)/2] OF ITEM;
TYPE ITEM = RECORD origen,destino:CARDINAL; peso:CARDINAL END;
```

La función *InicParticion* se necesita para inicializar las correspondientes particiones, constituyendo cada vértice como una partición distinta:

```
PROCEDURE InicParticion (VAR p:PARTICION);
   VAR i:CARDINAL;
BEGIN
   (* cada vertice en una componente distinta *)
   FOR i:=1 TO n DO p[i]:=i END
END InicParticion;
```

Para manejar las particiones, el procedimiento *Fusionar*, como su nombre indica, fusiona dos componentes, asignando a los elementos de la mayor el valor de los elementos de la menor:

```
PROCEDURE Fusionar (VAR p:PARTICION;a,b:CARDINAL);
   VAR i,temp:CARDINAL;
BEGIN
   IF (a>b) THEN (* los intercambiamos *)
      temp:=a; a:=b; b:=temp
   END;
   FOR i:=1 TO n DO
      IF p[i]=b THEN p[i]:=a END
   END;
END Fusionar;
```

La función *FinParticion* comprueba si existe solamente una componente conexa en toda la partición:

```
PROCEDURE FinParticion (VAR p:PARTICION):BOOLEAN;
   VAR i:CARDINAL;
BEGIN
   FOR i:=1 TO n DO
      IF p[i]<>1 THEN RETURN FALSE END
   END;
   RETURN TRUE;
END FinParticion;
```

Y la función *ObtenerComponente* devuelve el representante de la componente a la que pertenece un vértice:

```
PROCEDURE ObtenerComponente (VAR p:PARTICION; i:CARDINAL):CARDINAL;
```

```
BEGIN
   RETURN p[i]
END ObtenerComponente;
```

Por último, la función *Ordenar* construye un *GRAFO_ORDENADO* a partir del grafo original con todos sus arcos no vacíos ordenados de menor a mayor peso. Esta función devuelve el número de arcos no vacíos que componen el grafo original, y no la incluimos aquí por no extender excesivamente el desarrollo del problema. Para implementarla puede seguirse cualquier método de ordenación:

```
PROCEDURE Ordenar (VAR g:GRAFO_PONDERADO;
                   VAR g2:GRAFO_ORDENADO):CARDINAL;
```

Para el cálculo de su complejidad temporal, veamos el orden de complejidad de las partes que lo componen:

- En primer lugar, *Buscar* es de orden O(1) e *InicParticion* de orden O(n).
- El tiempo de ejecución de las funciones *Fusionar* y *FinParticion* va a depender del número de componentes conexas existentes en la partición pero como en cada paso este número se divide por dos, podemos concluir que su complejidad es de orden O(log$n$).
- Por otro lado, la ordenación de los arcos puede realizarse en un tiempo del orden de O($a$log$a$), siendo a el número de arcos del grafo. Como se verifica que $(n–1) \leq a \leq n(n–1)/2$ por tratarse de un grafo conexo, su orden es O($a$log$n$).

Resumiendo, el algoritmo consta de una inicialización de orden O($a$log$n$), seguido por un bucle que se repite a veces en donde existen dos operaciones de orden O(log$n$) y varias de orden O(1). Por consiguiente, su complejidad temporal es de orden O($a$log$n$).

Una vez más, es importante hacer notar en este punto que las afirmaciones anteriores se deben a que en esta implementación hemos utilizado el paso de argumentos que sean vectores o matrices por referencia en vez de por valor aun cuando no fuera necesario modificar el valor de tales argumentos. En caso contrario, cada invocación de función supondría una copia de los argumentos a la pila, con el tiempo que eso conlleva.

Respecto a su complejidad espacial, ésta es de orden O(n^2) pues de esta complejidad es la tabla que representa el grafo ordenado. Si en vez de utilizar matrices de adyacencia hubiésemos utilizado una representación no acotada, la complejidad espacial del algoritmo sería de orden O(a) (puesto que sólo hay que almacenar los arcos, y por ser un grafo conexo sabemos que $n–1 \leq a \leq n(n–1)/2$), aunque quizá se hubiera empeorado la complejidad temporal por el tiempo de acceso asociado a este tipo de estructuras.

Veamos ahora el algoritmo de Prim. Para su implementación vamos a necesitar definir dos tipos especiales:

```
TYPE MASPROXIMO = ARRAY [2..n] OF CARDINAL;
TYPE DISTMINIMA = ARRAY [2..n] OF CARDINAL;
```

siendo *MASPROXIMO*[*i*] el vértice del conjunto de vértices tratados hasta el momento más cercano al vértice *i*, y *DISTMINIMA*[*i*] la distancia desde *i* a ese vértice más próximo. Así, podemos implementar el algoritmo de Prim como sigue:

```
PROCEDURE Prim(VAR g:GRAFO_PONDERADO; VAR sol:GRAFO);
   VAR masproximo:MASPROXIMO;distmin:DISTMINIMA;
       min,i,j,k:CARDINAL;
BEGIN
   InicProx(g,masproximo,distmin);
   FOR i:=2 TO n DO
      min:=MAX(CARDINAL);
      FOR j:=2 TO n DO
         IF (distmin[j]<min) AND (distmin[j]<>0) THEN
            min:=distmin[j]; k:=j
         END
      END;
      sol[k,masproximo[k]]:=TRUE;
      distmin[k]:=0;
      FOR j:=2 TO n DO
         IF (g[j,k]<distmin[k]) THEN
            distmin[k]:=g[j,k];
            masproximo[j]:=k
         END
      END
   END
END Prim;
```

El procedimiento *InicProx* inicializa adecuadamente las variables:

```
PROCEDURE InicProx (VAR g:GRAFO_PONDERADO;VAR v:MASPROXIMO;
                                          VAR d:DISTMINIMA);
   VAR  i:CARDINAL;
BEGIN
   FOR i:=2 TO n DO
      v[i]:=1; d[i]:=g[i,1]
   END
END InicProx;
```

En cuanto a su complejidad, el bucle principal se repite n–1 veces, y los dos más internos también, lo que da lugar a un tiempo de complejidad del orden de $O((n–1)2(n–1)) = O(n^2)$.

Respecto a su complejidad espacial, ésta es también de orden $O(n^2)$ por la representación que hemos utilizado en este caso mediante matrices de adyacencia. En caso de haber utilizado una representación no acotada de los grafos podríamos haber conseguido una complejidad espacial de $O(a)$, aunque quizá se hubiera

empeorado la complejidad temporal del algoritmo por el tiempo de acceso que suponen este tipo de estructuras.

b) Supongamos que suministramos un grafo no conexo como entrada al algoritmo de Kruskal. En primer lugar el algoritmo terminaría puesto que el bucle va recorriendo todos los arcos de tal grafo. Y en segundo lugar su salida sería un árbol de expansión no conexo, pero que si observamos con detenimiento descubriremos que corresponde a la unión de los árboles de expansión mínimos de cada una de las componentes conexas del grafo. En este sentido, el algoritmo es bastante robusto.

No ocurre así con el de Prim, que no funciona en este caso puesto que hace uso de que sea conexo para buscar en cada paso el vértice k sobre el cual construir la solución. El que no sea conexo hace que, o bien k valga cero y por tanto se indexe erróneamente la matriz solución, o bien no se modifique su valor en cada paso, lo que hace que el algoritmo no termine nunca. Podemos concluir por tanto que el suministrar un grafo conexo como entrada es una precondición fuerte del algoritmo de Prim implementado.

Una vez analizados ambos algoritmos, el uso de uno u otro va a estar condicionado por el tipo de grafo que tratemos. La complejidad del algoritmo de Prim es siempre de orden $O(n^2)$ mientras que el orden de complejidad del algoritmo de Kruskal $O(a\log n)$ no sólo depende del número de vértices, sino también del número de arcos. Así, para grafos densos el número de arcos a es cercano a $n(n-1)/2$ por lo que el orden de complejidad del algoritmo de Kruskal es $O(n^2\log n)$, peor que la complejidad $O(n^2)$ de Prim. Sin embargo, para grafos dispersos en los que a es próximo a n, el algoritmo de Kruskal es de orden $O(n\log n)$, comportándose probablemente de forma más eficiente que el de Prim.

4.6 EL VIAJANTE DE COMERCIO

Se conocen las distancias entre un cierto número de ciudades. Un viajante debe, a partir de una de ellas, visitar cada ciudad exactamente una vez y regresar al punto de partida habiendo recorrido en total la menor distancia posible.

Este problema también puede ser enunciado más formalmente como sigue: dado un grafo g conexo y ponderado y dado uno de sus vértices v_0, encontrar el ciclo Hamiltoniano de coste mínimo que comienza y termina en v_0.

Cara a intentar solucionarlo mediante un algoritmo ávido, nos planteamos las siguientes estrategias:

a) Sea (C,v) el camino construido hasta el momento que comienza en v_0 y termina en v. Inicialmente C es vacío y $v = v_0$. Si C contiene todos los vértices de g, el algoritmo incluye el arco (v,v_0) y termina. Si no, incluye el arco (v,w) de longitud mínima entre todos los arcos desde v a los vértices w que no están en el camino C.

b) Otro posible algoritmo ávido escogería en cada iteración el arco más corto aún no considerado que cumpliera las dos condiciones siguientes: (*i*) no formar un ciclo con los arcos ya seleccionados, excepto en la última iteración, que es donde completa el viaje; y (*ii*) no es el tercer arco que incide en un mismo vértice de entre los ya escogidos.

Nos piden implementar ambos algoritmos y probar su funcionamiento, dando ejemplos en donde encuentren solución y en donde fallen, si es que esto ocurre.

Solución (☺)

a) El algoritmo pedido puede ser implementado utilizando los tipos de datos usados en el problema anterior, resultando:

```
TYPE PRESENCIA=ARRAY [1..n] OF BOOLEAN;(* vertices considerados *)

PROCEDURE Viajante1(VAR g:GRAFO_PONDERADO; VAR sol:GRAFO);
(* supone que el recorrido comienza en el vertice 1 *)
   VAR yaesta:PRESENCIA;
       i,verticeencurso,verticeanterior:CARDINAL;
BEGIN
   FOR i:=1 TO n DO yaesta[i]:=FALSE END;
   verticeencurso:=1;
   FOR i:=1 TO n DO
      verticeanterior:=verticeencurso;
      yaesta[verticeanterior]:=TRUE;
      verticeencurso:=Busca(g,verticeencurso,yaesta);
      sol[verticeanterior,verticeencurso]:=TRUE;
   END;
END Viajante1;
```

La clave de este algoritmo está en la función *Busca*, que es la que realiza el proceso de selección, decidiendo en cada paso el siguiente vértice de entre los posibles candidatos:

```
PROCEDURE Busca(VAR g:GRAFO_PONDERADO; vertice:CARDINAL;
                VAR yaesta:PRESENCIA):CARDINAL;
   VAR mejorvertice,i,min:CARDINAL;
BEGIN
   mejorvertice:=1; min:=MAX(CARDINAL);
   FOR i:=1 TO n DO
      IF (i<>vertice)AND(NOT(yaesta[i]))AND(g[vertice,i]<min) THEN
         min:=g[vertice,i]; mejorvertice:=i;
      END
   END;
   RETURN mejorvertice;
END Busca;
```

Respecto a los ejemplos de grafos en donde el algoritmo encuentra o no la solución óptima, comenzaremos por un grafo en donde la encuentra. Sea entonces

	2	3	4

1	1	5	2
2		4	6
3			3

una tabla que representa la matriz de adyacencia de un grafo ponderado g_1 con cuatro vértices. Partiendo del vértice 1, el algoritmo encuentra la solución óptima, que está formada por los arcos

$$(1,2),(2,3),(3,4),(4,1)$$

lo que da lugar al ciclo (1,2,3,4,1), cuyo coste es 1 + 4 + 3 + 2 = 10, óptimo pues el resto de soluciones poseen costes superiores o iguales a él: 15, 17, 14, 17 y 10.

Para ver un ejemplo en donde el algoritmo falla, consideraremos un grafo ponderado g_2 con seis vértices definido por la siguiente matriz de adyacencia:

	2	3	4	5	6
1	3	10	11	7	25
2		6	12	8	26
3			9	4	20
4				5	15
5					18

Partiendo del vértice 1 el algoritmo va a ir escogiendo la secuencia de arcos

$$(1,2),(2,3),(3,5),(5,4),(4,6),(6,1)$$

lo que da lugar al ciclo (1,2,3,5,4,6,1), cuyo coste es 3 + 6 + 4 + 5 + 15 + 25 = 58. Sin embargo, éste no es el ciclo con menor coste, pues el camino definido por los arcos:

$$(1,2),(2,3),(3,6),(6,4),(4,5),(5,1)$$

tiene un coste de 3 + 6 + 20 + 15 + 5 + 7 = 56.

b) El algoritmo pedido en este caso es muy similar al algoritmo de Kruskal que hemos visto en el problema anterior:

```
PROCEDURE Viajante2 (VAR g:GRAFO_PONDERADO; VAR sol:GRAFO);
   VAR   p:PARTICION;
         c1,c2:CARDINAL; (* indican componentes de la particion *)
         g_ordenado:GRAFO_ORDENADO;
         i,narcos:CARDINAL; (* numero de arcos del grafo *)
         u,v:CARDINAL; (* vertices tratados en cada paso *)
         ndest:ARRAY [1..n] OF CARDINAL; (* num. veces que cada
                              vertice es destino en la solucion *)
BEGIN
```

```
   InicParticion(p);
   FOR i:=1 TO n DO ndest[i]:=0 END;
   narcos:=Ordenar(g,g_ordenado);   (* devuelve el num. de arcos *)
   i:=0;
   WHILE (NOT FinParticion(p)) AND (i<narcos) DO
      INC(i);
      u:=g_ordenado[i].origen;
      v:=g_ordenado[i].destino;
      c1:=ObtenerComponente(p,u);
      c2:=ObtenerComponente(p,v);
      IF (c1<>c2) AND (ndest[u]<2) AND (ndest[v]<2) THEN
         Fusionar(p,c1,c2);
         sol[u,v]:=TRUE;
         INC(ndest[u]);
         INC(ndest[v]);
      END;
   END;
(* ahora solo nos queda el ultimo vertice, que cierra el ciclo *)
   WHILE (i<narcos) DO
      INC(i);
      u:=g_ordenado[i].origen;
      v:=g_ordenado[i].destino;
      IF (ndest[u]<2) AND (ndest[v]<2) THEN (* lo encontramos! *)
         sol[u,v]:=TRUE;
         INC(ndest[u]);
         INC(ndest[v]);
         i:=narcos; (* para salirnos del bucle *)
      END;
   END;
END Viajante2;
```

Los tipos y funciones utilizados por este procedimiento son los ya vistos en el problema anterior para el algoritmo de Kruskal.

El grafo g_1 es un ejemplo para el cual el algoritmo encuentra la solución óptima, al igual que ocurría con el anterior. Sin embargo, este algoritmo no encuentra la solución óptima en todos los casos, como ocurre por ejemplo con el grafo g_2 del apartado anterior. Para él, y partiendo del vértice 1, el algoritmo va a ir escogiendo la secuencia de arcos

$$(1,2),(3,5),(4,5),(2,3),(4,6),(1,6)$$

que da lugar al mismo ciclo que obteníamos antes, (1,2,3,5,4,6,1), de coste 58 y por tanto no óptimo.

4.7 LA MOCHILA

Dados n elementos e_1, e_2, ..., e_n con pesos p_1, p_2, ..., p_n y beneficios b_1, b_2, ..., b_n, y dada una mochila capaz de albergar hasta un máximo de peso M (capacidad de la

mochila), queremos encontrar las proporciones de los n elementos x_1, x_2, ..., x_n ($0 \le x_i \le 1$) que tenemos que introducir en la mochila de forma que la suma de los beneficios de los elementos escogidos sea máxima.

Esto es, hay que encontrar valores (x_1, x_2, ..., x_n) de forma que se maximice la cantidad $\sum_{i=1}^{n} b_i x_i$, sujeta a la restricción $\sum_{i=1}^{n} p_i x_i \le M$.

Solución (😐)

Un algoritmo ávido que resuelve este problema ordena los elementos de forma decreciente respecto a su ratio b_i / p_i y va añadiendo objetos mientras éstos vayan cabiendo.

Para implementar este algoritmo vamos a definir los siguientes tipos y constantes:

```
CONST MAXELEM = ...;  (* numero maximo de elementos *)
TYPE  REGISTRO  = RECORD peso:REAL; beneficio:CARDINAL END;
      ELEMENTOS = ARRAY [1..MAXELEM] OF REGISTRO;
      MOCHILA   = ARRAY [1..MAXELEM] OF REAL;(* composicion final*)
```

Con ellos, el algoritmo ávido para resolver el problema pedido con n elementos y para una capacidad de la mochila M es:

```
PROCEDURE Mochila(VAR e:ELEMENTOS; n:CARDINAL; M:REAL;
                  VAR sol:MOCHILA);
(* supone que los elementos de "e" estan en orden decreciente de
   su ratio b_i/p_i *)
   VAR peso_en_curso:REAL; i:CARDINAL;
BEGIN
   FOR i:=1 TO MAXELEM DO sol[i]:=0.0 END;
   peso_en_curso:=0.0; i:=1;
   WHILE (peso_en_curso<M) AND (i<=n) DO
      IF (e[i].peso+peso_en_curso)<=M THEN sol[i]:=1.0;
      ELSE sol[i]:=(M-peso_en_curso)/e[i].peso
      END;
      peso_en_curso:=peso_en_curso+(sol[i]*e[i].peso); INC(i)
   END
END Mochila;
```

Respecto al tiempo de ejecución del algoritmo, éste consta de la ordenación previa, de complejidad O($n$log$n$), y de un bucle que como máximo recorre todos los elementos, de complejidad O(n), por lo que el tiempo total resulta ser de orden O($n$log$n$).

Para demostrar que siguiendo la ordenación dada el algoritmo encuentra la solución óptima, vamos a suponer sin pérdida de generalidad que los elementos ya

están ordenados de esta forma, es decir, que $b_i / p_i \geq b_j / p_j$ si $i < j$. Por simplicidad en la notación utilizaremos los símbolos de sumatorios sin los índices.

Sea $X = (x_1,x_2,...,x_n)$ la solución encontrada por el algoritmo. Si $x_i = 1$ para todo i, la solución es óptima. Si no, sea j el menor índice tal que $x_j < 1$. Por la forma en que trabaja el algoritmo, $x_i = 1$ para todo $i < j$, $x_i = 0$ para todo $i > j$, y además $\Sigma x_i p_i = M$. Sea $B(X) = \Sigma x_i b_i$ el beneficio que se obtiene para esa solución.

Consideremos entonces $Y = (y_1,y_2,...,y_n)$ otra solución, y sea $B(Y) = \Sigma y_i b_i$ su beneficio. Por ser solución cumple que $\Sigma y_i p_i \leq M$. Entonces, restando ambas capacidades, podemos afirmar que $\Sigma(x_i p_i - y_i p_i) \geq 0$.

Calculemos entonces la diferencia de beneficios:

$$B(X) - B(Y) = \Sigma(x_i - y_i)b_i = \Sigma(x_i - y_i)p_i\ (b_i/p_i).$$

La segunda igualdad se obtiene multiplicando y dividiendo por p_i. Con esto, para el índice j escogido anteriormente sabemos que ocurre:

- Si $i < j$ entonces $x_i = 1$, y por tanto $(x_i - y_i) \geq 0$. Además, $(b_i/p_i) \geq (b_j/p_j)$ por la ordenación escogida (decreciente).
- Si $i > j$ entonces $x_i = 0$, y por tanto $(x_i - y_i) \leq 0$. Además, $(b_i/p_i) \leq (b_j/p_j)$ por la ordenación escogida (decreciente).
- Por último, si $i = j$ entonces $(b_i/p_i) = (b_j/p_j)$.

En consecuencia, podemos afirmar que $(x_i - y_i)(b_i/p_i) \geq (x_i - y_i)(b_j/p_j)$ para todo i, y por tanto:

$$B(X) - B(Y) = \Sigma(x_i - y_i)p_i\ (b_i/p_i) \geq (b_j/p_j)\Sigma(x_i - y_i)p_i \geq 0,$$

esto es, $B(X) \geq B(Y)$, como queríamos demostrar.

4.8 LA MOCHILA (0,1)

Consideremos una modificación al problema de la Mochila en donde añadimos el requerimiento de que no se pueden escoger fracciones de los elementos, es decir, $x_i = 0$ ó $x_i = 1$, $1 \leq i \leq n$. Como en el problema original, deseamos maximizar la cantidad $\sum_{i=1}^{n} b_i x_i$ sujeta a la restricción $\sum_{i=1}^{n} p_i x_i \leq M$. ¿Seguirá funcionando el algoritmo anterior en este caso?

Solución (☺)

Lamentablemente no funciona, como pone de manifiesto el siguiente ejemplo. Supongamos una mochila de capacidad $M = 6$, y que disponemos de los siguientes elementos (ya ordenados respecto a su ratio *beneficio*/*peso*):

	x_1	x_2	x_3
Peso	5	3	3
Beneficio	11	6	6

El algoritmo sólo introduciría el primer elemento, con un beneficio de 11, aunque sin embargo es posible obtener una mejor elección: podemos introducir los dos últimos elementos en la mochila puesto que no superan su capacidad, con un beneficio total de 12.

4.9 EL FONTANERO DILIGENTE

Un fontanero necesita hacer n reparaciones urgentes, y sabe de antemano el tiempo que le va a llevar cada una de ellas: en la tarea i-ésima tardará t_i minutos. Como en su empresa le pagan dependiendo de la satisfacción del cliente, necesita decidir el orden en el que atenderá los avisos para minimizar el tiempo medio de espera de los clientes.

En otras palabras, si llamamos E_i a lo que espera el cliente i-ésimo hasta ver reparada su avería por completo, necesita minimizar la expresión:

$$E(n) = \sum_{i=1}^{n} E_i \ .$$

Deseamos diseñar un algoritmo ávido que resuelva el problema y probar su validez, bien mediante demostración formal o con un contraejemplo que la refute.

Solución (😐)

En primer lugar hemos de observar que el fontanero siempre tardará el mismo tiempo global $T = t_1 + t_2 + ... + t_n$ en realizar todas las reparaciones, independientemente de la forma en que las ordene. Sin embargo, los tiempos de espera de los clientes sí dependen de esta ordenación.

En efecto, si mantiene la ordenación original de las tareas (1, 2, ..., n), la expresión de los tiempos de espera de los clientes viene dada por:

$$\begin{aligned} E_1 &= t_1 \\ E_2 &= t_1 + t_2 \\ &..... \\ E_n &= t_1 + t_2 + ... + t_n \ . \end{aligned}$$

Lo que queremos encontrar es una permutación de las tareas en donde se minimice la expresión de $E(n)$ que, basándonos en las ecuaciones anteriores, viene dada por:

$$E(n) = \sum_{i=1}^{n} E_i = \sum_{i=1}^{n} (n - i + 1)t_i \ .$$

Vamos a demostrar que la permutación óptima es aquella en la que los avisos se atienden en orden creciente de sus tiempos de reparación.

Para ello, denominemos $X = (x_1,x_2,...,x_n)$ a una permutación de los elementos $(1,2,...,n)$, y sean $(s_1,s_2,...,s_n)$ sus respectivos tiempos de ejecución, es decir, $(s_1,s_2,...,s_n)$ va a ser una permutación de los tiempos orginales $(t_1,t_2,...,t_n)$. Supongamos que no está ordenada en orden creciente de tiempo de reparación, es decir, que existen dos números $x_i < x_j$ tales que $s_i > s_j$.

Sea $Y = (y_1,y_2,...,y_n)$ la permutación obtenida a partir de X intercambiando x_i con x_j, es decir, $y_k = x_k$ si $k \neq i$ y $k \neq j$, $y_i = x_j$, $y_j = x_i$.

Si probamos que $E(Y) < E(X)$ habremos demostrado lo que buscamos, pues mientras más ordenada (según el criterio dado) esté la permutación, menor tiempo de espera supone. Pero para ello, basta darse cuenta que

$$E(Y) = (n - x_i + 1)s_j + (n - x_j + 1)s_i + \sum_{k=1,k\neq i,k\neq j}^{n} (n - k + 1)s_k$$

y que, por tanto:

$$E(X) - E(Y) = (n - x_i + 1)(s_i - s_j) + (n - x_j + 1)(s_j - s_i) = (x_j - x_i)(s_i - s_j) > 0.$$

En consecuencia, el algoritmo pedido consiste en atender a las llamadas en orden inverso a su tiempo de reparación. Con esto conseguirá minimizar el tiempo medio de espera de los clientes, tal y como hemos probado.

4.10 MÁS FONTANEROS

Supongamos que en la empresa del fontanero del apartado anterior aumenta el número de clientes debido a su buena calidad de servicio y deciden contratar a más personal, con lo que disponen de un total de F fontaneros para realizar las n tareas.

Modificar el diseño del algoritmo para que realice la asignación de tareas a fontaneros siguiendo con el criterio de calidad expuesto anteriormente.

Solución (😐)

En este caso también tenemos que minimizar el tiempo medio de espera de los clientes, pero lo que ocurre es que ahora existen F fontaneros dando servicio simultámeamente. Basándonos en el método utilizado anteriormente, la forma óptima de atender los avisos va a ser la siguiente:

- En primer lugar, se ordenan los avisos por orden creciente de tiempo de reparación.

- Un vez hecho esto, se van asignando los avisos por este orden, siempre al fontanero menos ocupado. En caso de haber varios con el mismo grado de ocupación, se escoge el de número menor.

En otras palabras, si los avisos están ordenados de forma que $t_i \leq t_j$ si $i < j$, asignaremos al fontanero k los avisos k, $k+F$, $k+2F$, ...

4.11 LA ASIGNACIÓN DE TAREAS

Supongamos que disponemos de n trabajadores y n tareas. Sea $b_{ij} > 0$ el coste de asignarle el trabajo j al trabajador i. Una asignación de tareas puede ser expresada como una asignación de los valores 0 ó 1 a las variables x_{ij}, donde $x_{ij} = 0$ significa que al trabajador i no le han asignado la tarea j, y $x_{ij} = 1$ indica que sí. Una asignación válida es aquella en la que a cada trabajador sólo le corresponde una tarea y cada tarea está asignada a un trabajador. Dada una asignación válida, definimos el *coste* de dicha asignación como:

$$\sum_{i=1}^{n}\sum_{j=1}^{n} x_{ij} b_{ij} .$$

Diremos que una asignación es óptima si es de mínimo coste. Cara a diseñar un algoritmo ávido para resolver este problema podemos pensar en dos estrategias distintas: asignar cada trabajador la mejor tarea posible, o bien asignar cada tarea al mejor trabajador disponible. Sin embargo, ninguna de las dos estrategias tiene por qué encontrar siempre soluciones óptimas. ¿Es alguna mejor que la otra?

Solución (☺)

Este es un problema que aparece con mucha frecuencia, en donde los costes son o bien tarifas (que los trabajadores cobran por cada tarea) o bien tiempos (que tardan en realizarlas). Para implementar ambos algoritmos vamos a definir la matriz de costes (b_{ij}):

```
TYPE COSTES = ARRAY[1..n],[1..n] OF CARDINAL;
```

que forma parte de los datos de entrada del problema, y la matriz de asignaciones (x_{ij}), que es la que buscamos:

```
TYPE ASIGNACION = ARRAY[1..n],[1..n] OF BOOLEAN;
```

Con esto, el primer algoritmo puede ser implementado como sigue:

```
PROCEDURE AsignacionOptima(VAR b:COSTES; VAR x:ASIGNACION);
   VAR trabajador,tarea:CARDINAL;
```

```
BEGIN
   FOR trabajador:=1 TO n DO (* inicializamos la matriz solucion *)
      FOR tarea:=1 TO n DO
         x[trabajador,tarea]:=FALSE
      END
   END;
   FOR trabajador:=1 TO n DO
      x[trabajador,MejorTarea(b,x,trabajador)]:=TRUE
   END
END AsignacionOptima;
```

La función *MejorTarea* es la que busca la mejor tarea aún no asignada para ese trabajador:

```
PROCEDURE MejorTarea (VAR b:COSTES; VAR x:ASIGNACION;
                          i:CARDINAL):CARDINAL;
   VAR tarea,min,mejortarea:CARDINAL;
BEGIN
   min:=MAX(CARDINAL);
   FOR tarea:=1 TO n DO
      IF (NOT YaEscogida(x,i,tarea))AND(b[i,tarea]<min) THEN
         min:=b[i,tarea];
         mejortarea:=tarea
      END
   END;
   RETURN mejortarea;
END MejorTarea;
```

Por último, la función *YaEscogida* decide si una tarea ya ha sido asignada previamente:

```
PROCEDURE YaEscogida(VAR x:ASIGNACION;
                       trabajador,tarea:CARDINAL):BOOLEAN;
   VAR i:CARDINAL;
BEGIN
   FOR i:=1 TO trabajador-1 DO
      IF x[i,tarea] THEN RETURN TRUE END
   END;
   RETURN FALSE;
END YaEscogida;
```

Lamentablemente, este algoritmo ávido no funciona para todos los casos como pone de manifiesto la siguiente matriz de valores:

Tarea

		1	2	3
	1	16	20	18
Trabajador	2	11	15	17
	3	17	1	20

Para ella, el algoritmo produce una matriz de asignaciones en donde los "unos" están en las posiciones (1,1), (2,2) y (3,3), esto es, asigna la tarea *i* al trabajador *i* (*i* = 1, 2 ,3), con un valor de la asignación de 51 (= 16 + 15 + 20). Sin embargo la asignación óptima se consigue con los "unos" en posiciones (1,3), (2,1) y (3,2), esto es, asigna la tarea 3 al trabajador 1, la 1 al trabajador 2 y la tarea 2 al trabajador 3, con un valor de la asignación de 30 (= 18 + 11 + 1).

Si utilizamos la segunda estrategia nos encontramos en una situación análoga. En primer lugar, su implementación es:

```
PROCEDURE AsignacionOptima2(VAR b:COSTES; VAR x:ASIGNACION);
   VAR trabajador,tarea:CARDINAL;
BEGIN
   FOR trabajador:=1 TO n DO (* inicializamos la matriz solucion *)
      FOR tarea:=1 TO n DO
         x[trabajador,tarea]:=FALSE
      END
   END;
   FOR tarea:=1 TO n DO
      x[MejorTrabajador(b,x,tarea),tarea]:=TRUE
   END;
END AsignacionOptima2;
```

La función *MejorTrabajador* es la que busca el mejor trabajador aún no asignado para esa tarea:

```
PROCEDURE MejorTrabajador (VAR b:COSTES; VAR x:ASIGNACION;
                           i:CARDINAL):CARDINAL;
   VAR trabajador,min,mejortrabajador:CARDINAL;
BEGIN
   min:=MAX(CARDINAL);
   FOR trabajador:=1 TO n DO
     IF(NOT YaEscogido(x,i,trabajador))AND(b[trabajador,i]<min)THEN
        min:=b[trabajador,i];
        mejortrabajador:=trabajador
      END
   END;
   RETURN mejortrabajador;
END MejorTrabajador;
```

Por último, la función *YaEscogido* decide si un trabajador ya ha sido asignado previamente:

```
PROCEDURE YaEscogido(VAR x:ASIGNACION;
                     trabajador,tarea:CARDINAL):BOOLEAN;
   VAR i:CARDINAL;
BEGIN
   FOR i:=1 TO tarea-1 DO
      IF x[trabajador,i] THEN RETURN TRUE END
   END;
   RETURN FALSE;
END YaEscogido;
```

Lamentablemente, este algoritmo ávido tampoco funciona para todos los casos como pone de manifiesto la siguiente matriz de valores:

		Tarea		
		1	2	3
	1	16	11	17
Trabajador	2	20	15	1
	3	18	17	20

Para ella, el algoritmo produce una matriz de asignaciones en donde los "unos" vuelven a estar en las posiciones (1,1), (2,2) y (3,3), con un valor de la asignación de 51 (=16+15+20). Sin embargo la asignación óptima se consigue con los "unos" en posiciones (3,1), (1,2) y (2,3), con un valor de la asignación de 30 (=18+11+1).

Respecto a la pregunta de si una estrategia es mejor que la otra, la respuesta es que no. La razón es que las soluciones son simétricas. Aún más, una es la imagen especular de la otra. Por tanto, si suponemos equiprobables los valores de las matrices, ambos algoritmos van a tener el mismo número de casos favorables y desfavorables.

4.12 LOS FICHEROS Y EL DISQUETE

Supongamos que disponemos de n ficheros $f_1, f_2, ..., f_n$ con tamaños $l_1, l_2, ..., l_n$ y un disquete de capacidad $d < l_1 + l_2 + ... + l_n$.

a) Queremos maximizar el número de ficheros que ha de contener el disquete, y para eso ordenamos los ficheros por orden creciente de su tamaño y vamos metiendo ficheros en el disco hasta que no podamos meter más. Determinar si este algoritmo ávido encuentra solución óptima en todos los casos.

b) Queremos llenar el disquete tanto como podamos, y para eso ordenamos los ficheros por orden decreciente de su tamaño, y vamos metiendo ficheros en el disco hasta que no podamos meter más. Determinar si este algoritmo ávido encuentra solución óptima en todos los casos.

Solución (😐)

a) Supongamos los ficheros $f_1, f_2, ..., f_n$ ordenados respecto a su tamaño, esto es, $l_1 \le l_2 \le ... \le l_n$. Dicho de otra forma, si llamamos L a la función que devuelve la longitud de un fichero dado, lo que tenemos es que $L(f_1) \le L(f_2) \le ... \le L(f_n)$.

El algoritmo ávido indicado en el enunciado de este apartado sugiere ir tomando los ficheros según están ordenados hasta que no quepa ninguno más.

Vamos a demostrar que el número de ficheros que caben de esta forma es el óptimo. Sea m el número de ficheros que dice el algoritmo que caben en un disquete de capacidad d. Si $d \ge \Sigma L(f_i)$ entonces m coincide con n. Pero si $d < \Sigma L(f_i)$, por la forma en la que trabaja el algoritmo sabemos que se verifica la siguiente relación:

$$\sum_{i=1}^{m} L(f_i) \le d < \sum_{i=1}^{m+1} L(f_i). \qquad [4.5]$$

Sea entonces $g_1, g_2, ..., g_s$ otro subconjunto de ficheros que caben también en el disquete, es decir, tal que

$$\sum_{i=1}^{s} L(g_i) \le d\,. \qquad [4.6]$$

Veamos que $s \le m$. En primer lugar, vamos a suponer sin pérdida de generalidad que el conjunto de los ficheros g_i está también ordenado en orden creciente de tamaño:

$$L(g_1) \le L(g_2) \le ... \le L(g_s).$$

Como ambas descomposiciones son distintas, sea k el primer índice tal que $f_k \ne g_k$. Podemos suponer sin perder generalidad que $k = 1$, puesto que si hasta f_{k-1} los ficheros son iguales podemos eliminarlos y restar la suma de los tamaños de tales ficheros a la capacidad de nuestro disquete.

Por la forma en que funciona el algoritmo, si $f_1 \ne g_1$ entonces $L(f_1) \le L(g_1)$ pues f_1 era el fichero de menor tamaño. Además, g_1 corresponderá a un fichero f_a en la ordenación inicial, con $a > 1$. Análogamente, g_2 corresponderá a un fichero f_b en la ordenación inicial, con $b > a > 1$, y por tanto $b > 2$, por lo que $L(g_2) \ge L(f_2)$. Repitiendo el razonamiento, los ficheros g_i se corresponderán con ficheros de la ordenación inicial, pero siempre cumpliendo que:

$$L(g_i) \ge L(f_i) \qquad (1 \le i \le s). \qquad [4.7]$$

Ahora bien, por la relaciones [4.6] y [4.7] obtenemos

$$d \ge \sum_{i=1}^{s} L(g_i) \ge \sum_{i=1}^{s} L(f_i)$$

Pero entonces, por [4.5], s ha de ser estrictamente menor que m+1, y por tanto $s \le m$, como queríamos demostrar.

b) En este caso el algoritmo no funciona, como pone de manifiesto el siguiente ejemplo: sean (15,10,10,2) los tamaños de cuatro ficheros ($n = 4$) ya ordenados en orden decreciente, y supongamos que disponemos de un disquete con capacidad

d = 22. La solución que encontraría el algoritmo ávido es 17 (=15+2), almacenando en el disquete el primer y el último fichero. Sin embargo existe una solución que aprovecha aún más el disquete, la formada por los tres últimos ficheros. Con ellos se ocupa completamente el disquete.

4.13 EL CAMIONERO CON PRISA

Un camionero conduce desde Bilbao a Málaga siguiendo una ruta dada y llevando un camión que le permite, con el tanque de gasolina lleno, recorrer n kilómetros sin parar. El camionero dispone de un mapa de carreteras que le indica las distancias entre las gasolineras que hay en su ruta. Como va con prisa, el camionero desea pararse a repostar el menor número de veces posible.

Deseamos diseñar un algoritmo ávido para determinar en qué gasolineras tiene que parar y demostrar que el algoritmo encuentra siempre la solución óptima.

Solución (☺/😐)

Supondremos que existen G gasolineras en la ruta que sigue el camionero entre Bilbao y Málaga, incluyendo una en la ciudad destino, y que están numeradas del 0 (gasolinera en Bilbao) a G–1 (la situada en Málaga).

Supondremos además que disponemos de un vector con la información que tiene el camionero sobre las distancias entre ellas:

```
TYPE DISTANCIA = ARRAY [1..G-1] OF CARDINAL;
```

de forma que el i-ésimo elemento del vector indica los kilómetros que hay entre las gasolineras i–1 e i. Para que el problema tenga solución hemos de suponer que ningún valor de ese vector es mayor que el número n de kilómetros que el camión puede recorrer sin repostar.

Con todo esto, el algoritmo ávido pedido va a consistir en intentar recorrer el mayor número posible de kilómetros sin repostar, esto es, tratar de ir desde cada gasolinera en donde se pare a repostar a la más lejana posible, así hasta llegar al destino.

Para demostrar la validez de este algoritmo ávido, sean $x_1,x_2,...,x_s$ las gasolineras en donde este algoritmo decide que hay que parar a repostar, y sea $y_1,y_2,...,y_t$ otro posible conjunto solución de gasolineras. Llamaremos X a un camión que sigue la primera solución, e Y a un camión que se guía por la segunda. Sea N el número total de kilómetros a recorrer (distancia entre las dos ciudades), y sea $D[i]$ la distancia recorrida por el camionero hasta la i-ésima gasolinera ($1 \le i \le G$–1). Es decir,

$$D[i] = \sum_{k=1}^{i} d[k] \quad \text{y} \quad D[G\text{–}1] = N.$$

Lo que tenemos que demostrar es que $s \le t$, puesto que lo que queríamos minimizar era el número de paradas a realizar. Para probarlo, basta con demostrar que $x_k \ge y_k$ para todo k.

En primer lugar, como ambas descomposiciones son distintas, sea k el primer índice tal que $x_k \neq y_k$. Podemos suponer sin perder generalidad que $k = 1$, puesto que hasta x_{k-1} los viajes son iguales, y en la gasolinera x_{k-1} ambos camiones rellenan su tanque completamente.

Por la forma en que funciona el algoritmo, si $x_1 \neq y_1$ entonces $x_1 > y_1$, pues x_1 era la gasolinera más alejada a donde podía viajar el camionero sin repostar.

Además, también se tiene que $x_2 \geq y_2$, pues x_2 era la gasolinera más alejada a donde podía viajar desde x_1 el camionero sin repostar. Para probar este hecho, supongamos por reducción al absurdo que y_2 fuera estrictamente mayor que x_2. Pero si Y consigue ir desde y_1 a y_2 es que hay menos de n kilómetros entre ellas, es decir,

$$D[y_2] - D[y_1] < n.$$

Por tanto desde x_1 también hay menos de n kilómetros hasta y_2, esto es,

$$D[y_2] - D[x_1] < n$$

puesto que $D[y_1] < D[x_1]$. Entonces el método no hubiera escogido x_2 como siguiente gasolinera a x_1 sino y_2, porque el algoritmo busca siempre la gasolinera más alejada de entre las que alcanza.

Repitiendo el proceso, vamos obteniendo en cada paso que $x_k \geq y_k$ para todo k, hasta llegar a la ciudad destino, lo que demuestra la hipótesis.

El siguiente procedimiento implementa este algoritmo, devolviendo un vector que indica en qué gasolineras ha de pararse y en cuáles no:

```
TYPE SOLUCION = ARRAY [1..G-1] OF BOOLEAN;
PROCEDURE Deprisa(n:CARDINAL; VAR d:DISTANCIA; VAR sol:SOLUCION);
   VAR i,numkilometros:CARDINAL;
BEGIN
   FOR i:=1 TO G-1 DO sol[i]:=FALSE END;
   i:=0;
   numkilometros:=0;
   REPEAT
      REPEAT
         INC(i);
         numkilometros:=numkilometros+d[i];
      UNTIL (numkilometros>n) OR (i=G-1);
      IF numkilometros>n THEN  (* si nos hemos pasado... *)
         DEC(i);               (* volvemos atras una gasolinera *)
         sol[i]:=TRUE;         (* y repostamos en ella. *)
         numkilometros:=0;     (* reset contador *)
      END
   UNTIL (i=G-1);
END Deprisa;
```

4.14 LA MULTIPLICACIÓN ÓPTIMA DE MATRICES

Necesitamos en este problema calcular la matriz producto M de n matrices dadas $M=M_1M_2...M_n$. Por ser asociativa la multiplicación de matrices, existen muchas formas posibles de realizar esa operación, cada una con un coste asociado (en términos del número de multiplicaciones escalares). Si cada M_i es de dimensión d_{i-1}xd_i ($1 \le i \le n$), multiplicar $M_i M_{i+1}$ requiere $d_{i-1}d_i d_{i+1}$ operaciones.

El problema consiste en encontrar el mínimo número de operaciones necesario para calcular el producto M.

En general, el coste asociado a las distintas formas de multiplicar las n matrices puede ser bastante diferente de unas a otras. Por ejemplo, para n = 4 y para las matrices M_1, M_2, M_3 y M_4 cuyos órdenes son:

$$M_1(30\times1),\ M_2(1\times40),\ M_3(40\times10),\ M_4(10\times25)$$

hay cinco formas distintas de multiplicarlas, y sus costes asociados (en términos de las multiplicaciones escalares que necesitan) son:

$$\begin{array}{lll}
((M_1M_2)M_3)M_4 & = 30\cdot1\cdot40 + 30\cdot40\cdot10 + 30\cdot10\cdot25 & = 20.700 \\
M_1(M_2(M_3M_4)) & = 40\cdot10\cdot25 + 1\cdot40\cdot25 + 30\cdot1\cdot25 & = 11.750 \\
(M_1M_2)(M_3M_4) & = 30\cdot1\cdot40 + 40\cdot10\cdot25 + 30\cdot40\cdot25 & = 41.200 \\
M_1((M_2M_3)M_4) & = 1\cdot40\cdot10 + 1\cdot10\cdot25 + 30\cdot1\cdot25 & = 1.400 \\
(M_1(M_2M_3))M_4 & = 1\cdot40\cdot10 + 30\cdot1\cdot10 + 30\cdot10\cdot25 & = 8.200
\end{array}$$

Como puede observarse, la mejor forma necesita casi treinta veces menos multiplicaciones que la peor, por lo cual es importante elegir una buena asociación. Podríamos pensar en calcular el coste de cada una de las opciones posibles y escoger la mejor de entre ellas antes de multiplicar. Sin embargo, para valores grandes de n esta estrategia es inútil, pues el número de opciones crece exponencialmente con n. De hecho, el número de opciones posible sigue la sucesión de los números de Catalán:

$$T(n) = \sum_{i=1}^{n-1} T(i)T(n-i) = \frac{1}{n}\binom{2n-2}{n-1} \in \Theta\left(\frac{4^n}{\sqrt{n}}\right).$$

Parece entonces muy útil la búsqueda de un algoritmo ávido que resuelva nuestro problema. La siguiente lista presenta cuatro estrategias diferentes:

a) Multiplicar primero las matrices M_iM_{i+1} cuya dimensión común d_i sea la menor entre todas, y repetir el proceso.

b) Multiplicar primero las matrices M_iM_{i+1} cuya dimensión común d_i sea la mayor entre todas, y repetir el proceso.

c) Realizar primero la multiplicación de las matrices M_iM_{i+1} que requiera menor número de operaciones ($d_{i-1}d_id_{i+1}$), y repetir el proceso.

d) Realizar primero la multiplicación de las matrices M_iM_{i+1} que requiera mayor número de operaciones ($d_{i-1}d_id_{i+1}$), y repetir el proceso.

Queremos determinar si alguna de las estrategias propuestas encuentra siempre solución óptima. Como es habitual en los algoritmos ávidos para comprobar su funcionamiento, sería necesario una demostración formal o bien dar un contraejemplo que justifique la respuesta.

Solución (☺)

Lamentablemente, ninguna de las estrategias presentadas encuentra la solución óptima. Por tanto, para todas es posible dar un contraejemplo en donde el algoritmo falla.

a) La primera estrategia consiste en multiplicar siempre primero las matrices $M_i\, M_{i+1}$ cuya dimensión común d_i es la menor entre todas. Pero observando el ejemplo anterior esta estrategia se muestra errónea, pues corresponde al producto $(M_1M_2)(M_3M_4)$, que resulta ser el peor de todos.

b) Si multiplicamos siempre primero las matrices M_iM_{i+1} cuya dimensión común d_i es la mayor entre todas encontramos la solución óptima para el ejemplo anterior, pero existen otros ejemplos donde falla esta estrategia, como el siguiente:

   Sean $M_1(2\times5)$, $M_2(5\times4)$ y $M_3(4\times1)$. Según esta estrategia, el producto escogido como mejor sería $(M_1M_2)M_3$, con un coste de 48 ($2\cdot5\cdot4 + 2\cdot4\cdot1$). Sin embargo, el producto $M_1(M_2M_3)$ tiene un coste asociado de 30 ($5\cdot4\cdot1 + 2\cdot5\cdot1$), menor que el anterior.

c) Si decidimos realizar siempre primero la multiplicación de las matrices $M_i\, M_{i+1}$ que requiera menor número de operaciones, encontraríamos la solución óptima para los dos ejemplos anteriores, pero no para el siguiente:

   Sean $M_1(3\times1)$, $M_2(1\times100)$ y $M_3(100\times5)$. Según esta estrategia, el producto escogido como mejor sería $(M_1M_2)M_3$, de coste 1.800 ($3\cdot1\cdot100 + 3\cdot100\cdot5$). Sin embargo, el coste del producto $M_1(M_2M_3)$ es de 515 ($1\cdot100\cdot5 + 3\cdot1\cdot5$), menor que el anterior.

d) Análogamente, realizando siempre primero la multiplicación de las matrices $M_i\, M_{i+1}$ que requiera mayor número de operaciones vamos a encontrar la solución óptima para este último ejemplo, pero no para los dos primeros.

Capítulo 5

PROGRAMACIÓN DINÁMICA

5.1 INTRODUCCIÓN

Existe una serie de problemas cuyas soluciones pueden ser expresadas recursivamente en términos matemáticos, y posiblemente la manera más natural de resolverlos es mediante un algoritmo recursivo. Sin embargo, el tiempo de ejecución de la solución recursiva, normalmente de orden exponencial y por tanto impracticable, puede mejorarse substancialmente mediante la Programación Dinámica.

En el diseño Divide y Vencerás del capítulo 3 veíamos cómo para resolver un problema lo dividíamos en subproblemas independientes, los cuales se resolvían de manera recursiva para combinar finalmente las soluciones y así resolver el problema original. El inconveniente se presenta cuando los subproblemas obtenidos no son independientes sino que existe solapamiento entre ellos; entonces es cuando una solución recursiva no resulta eficiente por la repetición de cálculos que conlleva. En estos casos es cuando la Programación Dinámica nos puede ofrecer una solución aceptable. La eficiencia de esta técnica consiste en resolver los subproblemas una sola vez, guardando sus soluciones en una tabla para su futura utilización.

La Programación Dinámica no sólo tiene sentido aplicarla por razones de eficiencia, sino porque además presenta un método capaz de resolver de manera eficiente problemas cuya solución ha sido abordada por otras técnicas y ha fracasado.

Donde tiene mayor aplicación la Programación Dinámica es en la resolución de problemas de optimización. En este tipo de problemas se pueden presentar distintas soluciones, cada una con un valor, y lo que se desea es encontrar la solución de valor óptimo (máximo o mínimo).

La solución de problemas mediante esta técnica se basa en el llamado principio de óptimo enunciado por Bellman en 1957 y que dice:

"*En una secuencia de decisiones óptima toda subsecuencia ha de ser también óptima*".

Hemos de observar que aunque este principio parece evidente no siempre es aplicable y por tanto es necesario verificar que se cumple para el problema en cuestión. Un ejemplo claro para el que no se verifica este principio aparece al tratar de encontrar el camino de coste máximo entre dos vértices de un grafo ponderado.

Para que un problema pueda ser abordado por esta técnica ha de cumplir dos condiciones:

- La solución al problema ha de ser alcanzada a través de una secuencia de decisiones, una en cada etapa.
- Dicha secuencia de decisiones ha de cumplir el principio de óptimo.

En grandes líneas, el diseño de un algoritmo de Programación Dinámica consta de los siguientes pasos:

1. Planteamiento de la solución como una sucesión de decisiones y verificación de que ésta cumple el principio de óptimo.
2. Definición recursiva de la solución.
3. Cálculo del valor de la solución óptima mediante una tabla en donde se almacenan soluciones a problemas parciales para reutilizar los cálculos.
4. Construcción de la solución óptima haciendo uso de la información contenida en la tabla anterior.

5.2 CÁLCULO DE LOS NÚMEROS DE FIBONACCI

Antes de abordar problemas más complejos veamos un primer ejemplo en el cual va a quedar reflejada toda esta problemática. Se trata del cálculo de los términos de la sucesión de números de Fibonacci. Dicha sucesión podemos expresarla recursivamente en términos matemáticos de la siguiente manera:

$$Fib(n) = \begin{cases} 1 & \text{si } n = 0,1 \\ Fib(n-1) + Fib(n-2) & \text{si } n > 1 \end{cases}$$

Por tanto, la forma más natural de calcular los términos de esa sucesión es mediante un programa recursivo:

```
PROCEDURE FibRec(n:CARDINAL):CARDINAL;
BEGIN
      IF n<=1 THEN RETURN 1
      ELSE
         RETURN FibRec(n-1) + FibRec(n-2)
      END
END FibRec;
```

El inconveniente es que el algoritmo resultante es poco eficiente ya que su tiempo de ejecución es de orden exponencial, como se vió en el primer capítulo.

Como podemos observar, la falta de eficiencia del algoritmo se debe a que se producen llamadas recursivas repetidas para calcular valores de la sucesión, que habiéndose calculado previamente, no se conserva el resultado y por tanto es necesario volver a calcular cada vez (véase el apartado 3.11 del capítulo 3, en donde se determina el número exacto de veces que se repite cada cálculo).

Para este problema es posible diseñar un algoritmo que en tiempo lineal lo resuelva mediante la construcción de una tabla que permita ir almacenando los cálculos realizados hasta el momento para poder reutilizarlos:

Fib(0)	*Fib*(1)	*Fib*(2)	...	*Fib*(*n*)

El algoritmo iterativo que calcula la sucesión de Fibonacci utilizando tal tabla es:

```
TYPE TABLA = ARRAY [0..n] OF CARDINAL
PROCEDURE FibIter(VAR T:TABLA;n:CARDINAL):CARDINAL;
   VAR i:CARDINAL;
BEGIN
   IF n<=1 THEN RETURN 1
   ELSE
      T[0]:=1;
      T[1]:=1:
      FOR i:=2 TO n DO
         T[i]:=T[i-1]+T[i-2]
      END;
      RETURN T[n]
   END
END FibIter;
```

Existe aún otra mejora a este algoritmo, que aparece al fijarnos que únicamente son necesarios los dos últimos valores calculados para determinar cada término, lo que permite eliminar la tabla entera y quedarnos solamente con dos variables para almacenar los dos últimos términos:

```
PROCEDURE FibIter2(n: CARDINAL):CARDINAL;
   VAR i,suma,x,y:CARDINAL; (* x e y son los 2 ultimos terminos *)
BEGIN
   IF n<=1 THEN RETURN 1
   ELSE
      x:=1; y:=1;
      FOR i:=2 TO n DO
         suma:=x+y; y:=x; x:=suma;
      END;
      RETURN suma
   END
END FibIter2;
```

Aunque esta función sea de la misma complejidad temporal que la anterior (lineal), consigue una complejidad espacial menor, pues de ser de orden O(*n*) pasa a ser O(1) ya que hemos eliminado la tabla.

El uso de estructuras (vectores o tablas) para eliminar la repetición de los cálculos, pieza clave de los algoritmos de Programación Dinámica, hace que en

este capítulo nos fijemos no sólo en la complejidad temporal de los algoritmos estudiados, sino también en su complejidad espacial.

En general, los algoritmos obtenidos mediante la aplicación de esta técnica consiguen tener complejidades (espacio y tiempo) bastante razonables, pero debemos evitar que el tratar de obtener una complejidad temporal de orden polinómico conduzca a una complejidad espacial demasiado elevada, como veremos en alguno de los ejemplos de este capítulo.

5.3 CÁLCULO DE LOS COEFICIENTES BINOMIALES

En la resolución de un problema, una vez encontrada la expresión recursiva que define su solución, muchas veces la dificultad estriba en la creación del vector o la tabla que ha de conservar los resultados parciales. Así en este segundo ejemplo, aunque también sencillo, observamos que vamos a necesitar una tabla bidimensional algo más compleja. Se trata del cálculo de los coeficientes binomiales, definidos como:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \quad \text{si } 0 < k < n, \quad \binom{n}{0} = \binom{n}{n} = 1 .$$

El algoritmo recursivo que los calcula resulta ser de complejidad exponencial por la repetición de los cálculos que realiza. No obstante, es posible diseñar un algoritmo con un tiempo de ejecución de orden O(*nk*) basado en la idea del Triángulo de Pascal. Para ello es necesario la creación de una tabla bidimensional en la que ir almacenando los valores intermedios que se utilizan posteriormente:

	0	1	2	3	...	*k*-1	*k*
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
...	...	...	...	...	...		
...	...	...	...	...	...	...	
n-1						C(*n*-1,*k*-1) +	C(*n*-1,*k*)
n							C(*n*,*k*)

Iremos construyendo esta tabla por filas de arriba hacia abajo y de izquierda a derecha mediante el siguiente algoritmo de complejidad polinómica:

```
PROCEDURE CoefIter(n,k: CARDINAL):CARDINAL;
   VAR i,j: CARDINAL;
       C: TABLA;
BEGIN
```

```
   FOR i:=0 TO n DO C[i,0]:=1 END;
   FOR i:=1 TO n DO C[i,1]:=i END;
   FOR i:=2 TO k DO C[i,i]:=1 END;
   FOR i:=3 TO n DO
      FOR j:=2 TO i-1 DO
         IF j<=k THEN
            C[i,j]:=C[i-1,j-1]+C[i-1,j]
         END
      END
   END;
   RETURN C[n,k]
END CoefIter.
```

5.4 LA SUBSECUENCIA COMÚN MÁXIMA

Hay muchos problemas para los cuales no sólo deseamos encontrar el valor de la solución óptima sino que además deseamos conocer cuál es la composición de esta solución, es decir, los elementos que forman parte de ella. En estos casos es necesario ir conservando no sólo los valores de las soluciones parciales, sino también cómo se llega a ellas. Esta información adicional puede ser almacenada en la misma tabla que las soluciones parciales, o bien en otra tabla al efecto.

Veamos un ejemplo en el que se crea una tabla y a partir de ella se reconstruye la solución. Se trata del cálculo de la subsecuencia común máxima. Vamos en primer lugar a definir el problema.

Dada una secuencia $X=\{x_1\ x_2\ ...\ x_m\}$, decimos que $Z=\{z_1\ z_2\ ...\ z_k\}$ es una subsecuencia de X (siendo $k \leq m$) si existe una secuencia creciente $\{i_1\ i_2\ ...\ i_k\}$ de índices de X tales que para todo $j = 1, 2, ..., k$ tenemos $x_{ij} = z_j$.

Dadas dos secuencias X e Y, decimos que Z es una subsecuencia común de X e Y si es subsecuencia de X y subsecuencia de Y. Deseamos determinar la subsecuencia de longitud máxima común a dos secuencias.

Solución (😐)

Llamaremos $L(i,j)$ a la longitud de la secuencia común máxima (SCM) de las secuencias X_i e Y_j , siendo X_i el i-ésimo prefijo de X (esto es, $X_i = \{x_1\ x_2\ ...\ x_i\}$) e Y_j el j-ésimo prefijo de Y, ($Y_j = \{y_1\ y_2\ ...\ y_j\}$).

Aplicando el principio de óptimo podemos plantear la solución como una sucesión de decisiones en las que en cada paso determinaremos si un carácter forma o no parte de la SCM. Escogiendo una estrategia hacia atrás, es decir, comenzando por los últimos caracteres de las dos secuencias X e Y, la solución viene dada por la siguiente relación en recurrencia:

$$L(i,j) = \begin{cases} 0 & \text{si } i = 0 \text{ o } j = 0 \\ L(i-1, j-1) + 1 & \text{si } i \neq 0,\ j \neq 0 \text{ y } x_i = y_j \\ Max\{L(i, j-1), L(i-1, j)\} & \text{si } i \neq 0,\ j \neq 0 \text{ y } x_i \neq y_j \end{cases}$$

La solución recursiva resulta ser de orden exponencial, y por tanto Programación Dinámica va a construir una tabla con los valores $L(i, j)$ para evitar la repetición de cálculos. Para ilustrar la construcción de la tabla supondremos que X e Y son las secuencias de valores:

$$X = \{1\ 0\ 0\ 1\ 0\ 1\ 0\ 1\}$$
$$Y = \{0\ 1\ 0\ 1\ 1\ 0\ 1\ 1\ 0\}$$

La tabla que permite calcular la subsecuencia común máxima es:

		0	1	2	3	4	5	6	7	8
			1	0	0	1	0	1	0	1
0		0	0	0	0	0	0	0	0	0
1	0	0	0 Sup	1 Diag	1 Diag	1 Izq	1 Diag	1 Izq	1 Diag	1 Izq
2	1	0	1 Diag	1 Sup	1 Sup	2 Diag	2 Izq	2 Diag	2 Izq	2 Diag
3	0	0	1 Sup	2 Diag	2 Diag	2 Sup	3 Diag	3 Izq	3 Diag	3 Izq
4	1	0	1 Diag	2 Sup	2 Sup	3 Diag	3 Sup	4 Diag	4 Izq	4 Diag
5	1	0	1 Diag	2 Sup	2 Sup	3 Diag	3 Sup	4 Diag	4 Sup	5 Diag
6	0	0	1 Sup	2 Diag	3 Diag	3 Sup	4 Diag	4 Sup	5 Diag	5 Sup
7	1	0	1 Diag	2 Sup	3 Sup	4 Diag	4 Sup	5 Diag	5 Sup	6 Diag
8	1	0	1 Diag	2 Sup	3 Sup	4 Diag	4 Sup	5 Diag	5 Sup	6 Diag
9	0	0	1 Sup	2 Diag	3 Diag	4 Sup	5 Diag	5 Sup	6 Diag	6 Sup

Esta tabla se va construyendo por filas y rellenando de izquierda a derecha. Como podemos ver en cada $L[i,j]$ hay dos datos: uno el que corresponde a la longitud de cada subsecuencia, y otro necesario para la construcción de la subsecuencia óptima.

La solución a la subsecuencia común máxima de las secuencias X e Y se encuentra en el extremo inferior derecho ($L[9,8]$) y por tanto su longitud es seis. Si queremos obtener cuál es esa subsecuencia hemos de recorrer la tabla (zona sombreada) a partir de esta posición siguiendo la información que nos indica cómo obtener las longitudes óptimas a partir de su procedencia (izquierda, diagonal o

superior). El algoritmo para construir la tabla tiene una complejidad de orden O(*nm*), siendo *n* y *m* las longitudes de las secuencias *X* e *Y*.

```
CONST N = ...; (* longitud maxima de una secuencia *)
TYPE  SECUENCIA = ARRAY [1..N] OF CARDINAL;
      PARES = RECORD numero:CARDINAL; procedencia:CHAR; END;
      TABLA = ARRAY [0..N],[0..N] OF PARES;

PROCEDURE SubSecMaxima(VAR X,Y:SECUENCIA;n,m:CARDINAL;VAR L:TABLA);
   VAR i,j:CARDINAL;
BEGIN
   FOR i:=0 TO m DO  (* condiciones iniciales *)
      L[i,0].numero:=0
   END;
   FOR j:=0 TO n DO
      L[0,j].numero:=0
   END;
   FOR i:=1 TO m DO
      FOR j:=1 TO n DO
         IF Y[i] = X[j] THEN
            L[i,j].numero:=L[i-1,j-1].numero+1;
            L[i,j].procedencia:="D"
         ELSIF L[i-1,j].numero >=L[i,j-1].numero THEN
            L[i,j].numero:=L[i-1,j].numero;
            L[i,j].procedencia:="S"
         ELSE
            L[i,j].numero:=L[i,j-1].numero;
            L[i,j].procedencia:="I"
         END
      END
   END
END SubSecMaxima.
```

Para encontrar cuál es esa subsecuencia óptima hacemos uso de la información contenida en el campo procedencia de la tabla *L*, sabiendo que '*I*'(por "Izq") significa que la información la toma de la casilla de la izquierda, '*S*' ("Sup") de la casilla superior y de la misma manera '*D*'("Diag") corresponde a la casilla que está en la diagonal. El algoritmo que recorre la tabla construyendo la solución a partir de esta información y de la secuencia *Y* es el que sigue:

```
PROCEDURE Escribir(VAR L:TABLA; VAR Y:SECUENCIA; i,j:CARDINAL;
                   VAR sol:SECUENCIA; VAR l:CARDINAL);
   (* sol es la secuencia solucion, l su longitud, i es la longitud
      de la secuencia Y, y j la de X *)
BEGIN
   IF (i=0) OR (j=0) THEN RETURN END;
```

```
    IF L[i,j].procedencia = "D" THEN
       Escribir(L,Y,i-1,j-1,sol,l);
       sol[l]:=Y[i];
       INC(l);
    ELSIF L[i,j].procedencia = "S" THEN
       Escribir(L,Y,i-1,j,sol,l)
    ELSE
       Escribir(L,Y,i,j-1,sol,l)
    END
END Escribir
```

La complejidad de este algoritmo es de orden O(n+m) ya que en cada paso de la recursión puede ir disminuyendo o bien el parámetro i o bien j hasta alcanzar la posición $L[i,j]$ para $i = 0$ ó $j = 0$.

5.5 INTERESES BANCARIOS

Dadas n funciones f_1, f_2, ..., f_n y un entero positivo M, deseamos maximizar la función $f_1(x_1) + f_2(x_2) + ... + f_n(x_n)$ sujeta a la restricción $x_1 + x_2 + ... + x_n = M$, donde $f_i(0) = 0$ (i=1,..,n), x_i son números naturales, y todas las funciones son monótonas crecientes, es decir, $x \geq y$ implica que $f_i(x) > f_i(y)$. Supóngase que los valores de cada función se almacenan en un vector.

Este problema tiene una aplicación real muy interesante, en donde f_i representa la función de interés que proporciona el banco i, y lo que deseamos es maximizar el interés total al invertir una cantidad determinada de dinero M. Los valores x_i van a representar la cantidad a invertir en cada uno de los n bancos.

Solución (😐)

Sea f_i un vector que almacena el interés del banco i ($1 \leq i \leq n$) para una inversión de 1, 2, 3, ..., M pesetas. Esto es, $f_i(j)$ indicará el interés que ofrece el banco i para j pesetas, con $0 < i \leq n$, $0 < j \leq M$.

Para poder plantear el problema como una sucesión de decisiones, llamaremos $I_n(M)$ al interés máximo al invertir M pesetas en n bancos,

$$I_n(M) = f_1(x_1) + f_2(x_2) + ... + f_n(x_n)$$

que es la función a maximizar, sujeta a la restricción $x_1 + x_2 + ... + x_n = M$.

Veamos cómo aplicar el principio de óptimo. Si $I_n(M)$ es el resultado de una secuencia de decisiones y resulta ser óptima para el problema de invertir una cantidad M en n bancos, cualquiera de sus subsecuencias de decisiones ha de ser también óptima y así la cantidad

$$I_{n-1}(M - x_n) = f_1(x_1) + f_2(x_2) + ... + f_{n-1}(x_{n-1})$$

será también óptima para el subproblema de invertir ($M - x_n$) pesetas en $n - 1$ bancos. Y por tanto el principio de óptimo nos lleva a plantear la siguiente relación en recurrencia:

$$I_n(x) = \begin{cases} f_1(x) & \text{si } n = 1 \\ \underset{0 \le t \le x}{Max}\{I_{n-1}(x-t) + f_n(t)\} & \text{en otro caso.} \end{cases} \quad [5.1]$$

Para resolverla y calcular $I_n(M)$, vamos a utilizar una matriz I de dimensión nxM en donde iremos almacenando los resultados parciales y así eliminar la repetición de los cálculos. El valor de $I[i,j]$ va a representar el interés de j pesetas cuando se dispone de i bancos, por tanto la solución buscada se encontrará en $I[n,M]$. Para guardar los datos iniciales del problema vamos a utilizar otra matriz F, de la misma dimensión, y donde $F[i,j]$ representa el interés del banco i para j pesetas.

En consecuencia, para calcular el valor pedido de $I[n,M]$ rellenaremos la tabla por filas, empezando por los valores iniciales de la ecuación en recurrencia, y según el siguiente algoritmo:

```
CONST n = ...; (* numero de bancos *)
      M = ...; (* cantidad a invertir *)
TYPE MATRIZ = ARRAY [1..n],[0..M] OF CARDINAL;

PROCEDURE Intereses(VAR F:MATRIZ;VAR I:MATRIZ):CARDINAL;
   VAR i,j: CARDINAL;
BEGIN
   FOR i:=1 TO n DO I[i,0]:=0 END;
   FOR j:=1 TO M DO I[1,j]:=F[1,j] END;
   FOR i:=2 TO n DO
      FOR j:=1 TO M DO
         I[i,j]:=Max(I,F,i,j)
      END
   END;
   RETURN I[n,M]
END Intereses;
```

La función *Max* es la que calcula el máximo que aparece en la expresión [5.1]:

```
PROCEDURE Max(VAR I,F:MATRIZ;i,j:CARDINAL):CARDINAL;
   VAR max,t:CARDINAL;
BEGIN
   max:= I[i-1,j] + F[i,0];
   FOR t:=1 TO j DO
      max:=Max2(max,I[i-1,j-t]+F[i,t])
   END;
   RETURN max
END Max;
```

La función *Max2* es la que calcula el máximo de dos números naturales. La complejidad del algoritmo completo es de orden $O(nM^2)$, puesto que la complejidad de *Max* es $O(j)$ y se invoca dentro de dos bucles anidados que se

desarrollan desde 1 hasta *M*. Es importante hacer notar el uso de parámetros por referencia en lugar de por valor para evitar la copia de las matrices en la pila de ejecución del programa.

Por otro lado, la complejidad espacial del algoritmo es del orden O(*nM*), pues de este orden son las dos matrices que se utilizan para almacenar los resultados intermedios.

En este ejemplo queda de manifiesto la efectividad del uso de estructuras en los algoritmos de Programación Dinámica para conseguir obtener tiempos de ejecución de orden polinómico, frente a los tiempos exponenciales de los algoritmos recursivos iniciales.

5.6 EL VIAJE MÁS BARATO POR RÍO

Sobre el río Guadalhorce hay *n* embarcaderos. En cada uno de ellos se puede alquilar un bote que permite ir a cualquier otro embarcadero río abajo (es imposible ir río arriba). Existe una tabla de tarifas que indica el coste del viaje del embarcadero *i* al *j* para cualquier embarcadero de partida *i* y cualquier embarcadero de llegada *j* más abajo en el río ($i < j$). Puede suceder que un viaje de *i* a *j* sea más caro que una sucesión de viajes más cortos, en cuyo caso se tomaría un primer bote hasta un embarcadero *k* y un segundo bote para continuar a partir de *k*. No hay coste adicional por cambiar de bote.

Nuestro problema consiste en diseñar un algoritmo eficiente que determine el coste mínimo para cada par de puntos *i*,*j* ($i < j$) y determinar, en función de *n*, el tiempo empleado por el algoritmo.

Solución (☺)

Llamaremos *T*[*i*,*j*] a la tarifa para ir del embarcadero *i* al *j* (directo). Estos valores se almacenarán en una matriz triangular superior de orden *n*, siendo *n* el número de embarcaderos.

El problema puede resolverse mediante Programación Dinámica ya que para calcular el coste óptimo para ir del embarcadero *i* al *j* podemos hacerlo de forma recurrente, suponiendo que la primera parada la realizamos en un embarcadero intermedio *k* ($i < k \leq j$):

$$C(i,j) = T(i,k) + C(k,j).$$

En esta ecuación se contempla el viaje directo, que corresponde al caso en el que *k* coincide con *j*. Esta ecuación verifica también que la solución buscada *C*(*i*,*j*) satisface el principio del óptimo, pues el coste *C*(*k*,*j*), que forma parte de la solución, ha de ser, a su vez, óptimo. Podemos plantear entonces la siguiente expresión de la solución:

$$C(i,j) = \begin{cases} 0 & \text{si } i = j \\ \underset{i<k\leq j}{Min}\{T(i,k) + C(k,j)\} & \text{si } i < j \end{cases} \qquad [5.2]$$

La idea de esta segunda expresión surge al observar que en cualquiera de los trayectos siempre existe un primer salto inicial óptimo.

Para resolverla según la técnica de Programación Dinámica, hace falta utilizar una estructura para almacenar resultados intermedios y evitar la repetición de los cálculos. La estructura que usaremos es una matriz triangular de costes *C*[*i,j*], que iremos rellenando por diagonales mediante el procedimiento que hemos denominado *Costes*. La solución al problema es la propia tabla, y sus valores *C*[*i,j*] indican el coste óptimo para ir del embarcadero *i* al *j*.

```
CONST MAXEMBARCADEROS = ...;
TYPE MATRIZ=ARRAY[1..MAXEMBARCADEROS],[1..MAXEMBARCADEROS] OF CARDINAL;

   PROCEDURE Costes(VAR C:MATRIZ;n:CARDINAL);
      VAR i, diagonal:CARDINAL;
   BEGIN
      FOR i:=1 TO n DO C[i,i]:=0 END; (* condiciones iniciales *)
      FOR diagonal:=1 TO n-1 DO
         FOR i:=1 TO n-diagonal DO
            C[i,i+diagonal]:=Min(C,i,i+diagonal)
         END
      END
   END Costes;
```

Dicho procedimiento utiliza la siguiente función, que permite calcular la expresión del mínimo que aparece en la ecuación en recurrencia [5.2]:

```
   PROCEDURE Min(VAR C:MATRIZ; i,j:CARDINAL):CARDINAL;
      VAR k,min:CARDINAL;
   BEGIN
      min:=MAX(CARDINAL);
      FOR k:=i+1 TO j DO
         min:=Min2(min,T[i,k] + C[k,j])
      END;
      RETURN min
   END Min;
```

La función *Min2* es la que calcula el mínimo de dos números naturales. Es importante observar que esta función, por la forma en que se va rellenando la matriz *C*, sólo hace uso de los elementos calculados hasta el momento.

La complejidad del algoritmo es de orden $O(n^3)$, pues está compuesto por dos bucles anidados de tamaño *n*, que contienen la llamada a una función de orden $O(n)$, la que calcula el mínimo.

5.7 TRANSFORMACIÓN DE CADENAS

Sean *u* y *v* dos cadenas de caracteres. Se desea transformar *u* en *v* con el mínimo número de operaciones básicas del tipo siguiente: eliminar un carácter, añadir un

carácter, y cambiar un carácter. Por ejemplo, podemos pasar de *abbac* a *abcbc* en tres pasos:

abbac	→ *abac*	(eliminamos *b* en la posición 3)
	→ *ababc*	(añadimos *b* en la posición 4)
	→ *abcbc*	(cambiamos *a* en la posición 3 por *c*)

Sin embargo, esta tranformación no es óptima. Lo que queremos en este caso es diseñar un algoritmo que calcule el número mínimo de operaciones, de esos tres tipos, necesarias para transformar *u* en *v* y cuáles son esas operaciones, estudiando su complejidad en función de las longitudes de *u* y *v*.

Solución (😐)

En primer lugar, la transformación mostrada arriba no es óptima ya que podemos pasar de *abbac* a *abcbc* en sólo dos pasos:

abbac	→ *abcac*	(cambiamos *b* en la posicion 3 por *c*)
	→ *abcbc*	(cambiamos *a* en la posición 4 por *c*)

Llamaremos *m* a la longitud de la cadena *u*, *n* a la longitud de la cadena *v*, y $OB(m,n)$ indicará el número de operaciones básicas mínimo para transformar una cadena *u* de longitud *m* en otra cadena *v* de longitud *n*.

Para resolver el problema utilizando Programación Dinámica es necesario plantearlo como una sucesión de decisiones que satisfaga el principio de óptimo.

Para plantearla, vamos a fijarnos en el último elemento de cada una de las cadenas. Si los dos son iguales, entonces tendremos que calcular el número de operaciones básicas necesarias para obtener de la primera cadena menos el último elemento, y la segunda cadena también sin el último elemento, es decir,

$$OB(m,n) = OB(m-1,n-1) \text{ si } u_m = v_n.$$

Pero si los últimos elementos fueran distintos habría que escoger la situación más beneficiosa de entre tres posibles: (*i*) considerar la primera cadena y la segunda pero sin el último elemento, o bien (*ii*) la primera cadena menos el último elemento y la segunda cadena, o bien (*iii*) las dos cadenas sin el último elemento. Esto da lugar a la siguiente relación en recurrencia para $OB(m,n)$ para este caso:

$$OB(m,n)=1+Min\{OB(m,n-1),OB(m-1,\text{n}),OB(m-1,n-1)\} \text{ si } m\neq 0,\ n\neq 0 \text{ y } u_m\neq v_n.$$

En cuanto a las condiciones iniciales, tenemos las tres siguientes:

$$OB(0,0) = 0, \quad OB(m,0) = m \quad \text{y} \quad OB(0,n) = n.$$

Una vez disponemos de la ecuación en recurrencia necesitamos resolverla utilizando alguna estructura que nos permita reutilizar resultados intermedios, como mostramos a continuación:

```
CONST MAXCARACTERES = ...;
TYPE  CADENA=ARRAY[1..MAXCARACTERES] OF CHAR;
      TABLA=ARRAY[0..MAXCARACTERES],[0..MAXCARACTERES] OF CARDINAL;
```

```
PROCEDURE Cadena(VAR OB:TABLA;u,v:CADENA;n,m:CARDINAL):CARDINAL;
   VAR i,j: CARDINAL;
BEGIN
   FOR i:=0 TO m DO OB[i,0]:=i; END;
   FOR j:=0 TO n DO OB[0,j]:=j END;
   FOR i:=1 TO m DO
      FOR j:=1 TO n DO
         IF u[i]= v[j] THEN OB[i,j]:=OB[i-1,j-1]
         ELSE OB[i,j]:=Min3(OB[i,j-1],OB[i-1,j],OB[i-1,j-1])+1;
         END
      END
   END;
   RETURN OB[m,n]
END Cadena;
```

El procedimiento *Cadena* va a permitir la creación de la tabla *OB* que calcula el número mínimo de operaciones básicas. La solución se encuentra en *OB*[*m,n*], y la tabla se construye fila a fila (a partir de los valores que definen las condiciones iniciales) para poder ir reutilizando los valores calculados previamente. La función *Min3* es la que calcula el mínimo de tres enteros.

Como el algoritmo se limita a dos bucles anidados que sólo incluyen operaciones constantes la complejidad de este algoritmo es de orden O(*mn*).

5.8 LA FUNCIÓN DE ACKERMANN

La función de Ackermann se define recursivamente del modo siguiente:

$$\begin{cases} Ack(0,n) = n+1 \\ Ack(m,0) = Ack(m-1,1) \; si \; m > 0 \\ Ack(m,n) = Ack(m-1, Ack(m,n-1)) \; si \; m,n > 0. \end{cases}$$

Nos planteamos los beneficios de diseñar, si es posible, un algoritmo de Programación Dinámica para calcular *Ack*(*m*,*n*).

Solución (☺)

Este problema nos permite analizar uno de los aspectos más importantes de la Programación Dinámica: la búsqueda de estructuras que permitan resolver una ecuación en recurrencia reutilizando los cálculos realizados hasta el momento. Como veremos en este ejemplo, esta tarea es a veces complicada.

Si observamos la definición recursiva de esta función para valores de *m* y *n* mayores que 0, vemos que para calcular el valor de *Ack*(*m*,*n*) será necesario utilizar un vector *A* suficientemente grande sobre el cual se irán almacenando de izquierda a derecha los sucesivos valores de la función, comenzando con $m = 0$. En cada paso

m del algoritmo se actualizarán los elementos $A[i]$ ($1 \le i \le n$), que van a almacenar el valor de $Ack(m,i)$.

Para el cálculo de cada elemento $A[i]$ se necesitará, además del elemento anterior (obsérvese que $A[i-1]$ contiene el valor de $Ack(m,i-1)$), un elemento del vector calculado en el paso anterior. La dificultad que entraña es que, conforme aumenta *m*, el elemento al que tenemos que referirnos del vector previamente calculado tiene un índice excesivamente elevado,

$$2^{2^{\cdot^{\cdot^{2^m}}}}$$

con lo cual la dimensión del vector *A* ha de ser muy grande.

Un algoritmo que resuelve el problema siguiendo estas indicaciones es el siguiente:

```
CONST MaxIndice = ...;
TYPE VECTOR = ARRAY[0..MaxIndice] OF CARDINAL;

PROCEDURE Ackerman(m,n:CARDINAL):CARDINAL;
   VAR i,j:CARDINAL; max,l:LONGREAL; A:VECTOR;
BEGIN
   max:=1.0;
   FOR i:=1 TO m DO
      max:=Pot(2.0,max)
   END;
   FOR i:=0 TO VAL(CARDINAL,Pot(max,LONGREAL(n))) DO
      A[i]:=i+1
   END;
   l:=max;
   FOR i:=1 TO m DO
      A[0]:=A[1];
      l:=Log(2.0,l);
      FOR j:=1 TO VAL(CARDINAL,Pot(l,LONGREAL(n))) DO
         A[j]:=A[A[j-1]]
      END;
   END;
   RETURN A[n]
END Ackerman;
```

La función *Pot(a,b)* es la que calcula la potencia *b*-ésima de un número *a* dado, esto es, a^b, y la función $Log(a,b)$ la que calcula el logaritmo en base *a* de *b*.

La complejidad temporal del algoritmo viene determinada en primer lugar por el valor del parámetro *m* ya que ha de actualizarse *m* veces el vector *A*, y además por el tamaño de este vector, resultando en un orden $O(m \cdot MaxIndice)$ debido a los dos bucles anidados del programa.

Este procedimiento es, sin embargo, muy "ingenuo". Y decimos esto porque, aunque perfectamente correcto desde un punto de vista teórico, su utilidad práctica es más bien poca. Al tener que manejar números tan grandes, que a su vez deben

ser usados como índices del vector, es muy pequeño el número de pasos que soporta sin exceder la capacidad de cálculo de cualquier ordenador. Desgraciadamente no conseguimos de esta forma manejar la "intratabilidad" de la función de Ackerman.

5.9 EL PROBLEMA DEL CAMBIO

Dentro del tema dedicado a algoritmos ávidos vimos un algoritmo para minimizar, dado un sistema monetario, el número de monedas necesarias para reunir una cantidad. Aquel algoritmo funcionaba cuando los tipos de monedas eran, por ejemplo, de 1, 5, 10 y 25 unidades, pero no obtenía necesariamente la descomposición óptima si añadíamos una moneda de 12 unidades al sistema.

Dado que el algoritmo ávido para este problema falla en algunas ocasiones, nos planteamos si puede resolverse utilizando Programación Dinámica de forma que la solución sea satisfactoria en todos los casos.

Solución (😐)

Sea n el número de tipos de monedas distintos, L la cantidad a conseguir y $T[1..n]$ un vector con el valor de cada tipo de moneda del sistema. Supondremos que disponemos de una cantidad inagotable de monedas de cada tipo.

Llamaremos $C(i,j)$ $(1 \le i \le n,\ 1 \le j \le L)$ al número mínimo de monedas para obtener la cantidad j restringiéndose a los tipos $T[1]$, $T[2]$, ..., $T[i]$. Si no se puede conseguir dicha cantidad entonces $C(i,j) = \infty$. En primer lugar hemos de encontrar una expresión recursiva de $C(i,j)$. Para ello observemos que en cada paso existen dos opciones:

1. No incluir ninguna moneda del tipo $T(i)$. Esto supone que el valor de $C(i,j)$ va a coincidir con el de $C(i-1,j)$, y por tanto $C(i,j) = C(i-1,j)$.
2. Sí incluirla. Pero entonces, al incluir la moneda del tipo $T(i)$, el número de monedas global coincide con el número óptimo de monedas para una cantidad $(j - T(i))$ más esta moneda $T(i)$ que se incluye, es decir podemos expresar $C(i,j)$ en este caso como $C(i,j) = 1 + C(i,j - T(i))$.

El cálculo de $C(i,j)$ óptimo tomará la solución más favorable, es decir, el menor valor de ambas opciones. Con esto, la relación en recurrencia queda definida como:

$$C(i,j) = \begin{cases} \infty & \text{si } i = 1 \text{ y } 1 \le j < T(i) \\ 0 & \text{si } j = 0 \\ 1 + C(i, j - T(i)) & \text{si } i = 1 \text{ y } j \ge T(i) \\ C(i-1, j) & \text{si } i > 1 \text{ y } j < T(i) \\ Min\{C(i-1, j), 1 + C(i, j - T(i))\} & \text{en otro caso} \end{cases}$$

Una vez disponemos de la solución recursiva del problema, aplicaremos un algoritmo de Programación Dinámica para calcular los $C(n,j)$, $1 \le j \le L$, mediante el uso de un vector de longitud L.

Llamemos C a dicho vector, que verifica que en cada paso i ($1 \leq i \leq n$), $C[j]$ va a contener el valor de $C(i,j)$. La idea es ir actualizando dicho vector paso a paso hasta llegar al paso n. El algoritmo que construye este vector es el siguiente:

```
CONST  n = ...; (* num. tipos de monedas distintos del sistema *)
       L = ...; (* cantidad a conseguir *)
TYPE   TIPOMONEDA = ARRAY[1..n] OF CARDINAL;
       VECTOR = ARRAY[0..L] OF CARDINAL;

PROCEDURE Cambio(VAR C:VECTOR;L,n:CARDINAL;VAR T:TIPOMONEDA):CARDINAL;
   VAR i,j:CARDINAL;
BEGIN
   C[0]:=0;
   FOR i:=1 TO n DO
      FOR j:=1 TO L DO
         IF (i=1) AND (j<T[i]) THEN
            C[j]:=MAX(CARDINAL)
         ELSIF i=1 THEN
            C[j]:=1+C[j-T[1]]
         ELSIF j>=T[i] THEN
            C[j]:=Min2(C[j],1+C[j-T[i]])
         (* ELSE C[j] no se modifica *)
         END
      END;
   END;
   RETURN C[L]
END Cambio;
```

El algoritmo devuelve un valor, que es el número óptimo de monedas necesario para obtener la cantidad L, siendo su complejidad temporal de orden O(nL) y la espacial de orden O(L).

Una vez calculado el vector que nos permite encontrar el número mínimo de monedas es posible diseñar un algoritmo que construya la solución.

Para ello va a ser necesario mantener una tabla de valores lógicos $P[i,j]$ que indique la procedencia del valor $C(i,j)$ en la expresión en recurrencia, es decir, si $C(i,j) = C(i–1,j)$ (lo que indica que no se toma una moneda de valor $T[i]$) o bien $C(i,j) = C(i,j–T[i]) + 1$ (indicando que sí se incluye una moneda del valor $T[i]$). Por tanto, bastará definir $P[i,j] = FALSE$ en el primer caso, $TRUE$ en el segundo.

Para poder calcular los valores de esta matriz de procedencia $P[i,j]$ se necesitan tener presentes todos los valores de $C(i,j)$, para lo cual ya no es suficiente un vector C como el utilizado en el apartado anterior, sino que será necesaria la creación de una matriz $C[i,j]$ que conserve los distintos valores para $i = 1,2,...,n$. El algoritmo que implementa tal estrategia es el siguiente:

```
TYPE  MONEDAS = ARRAY[1..n] OF CARDINAL;
      MATRIZ = ARRAY[1..n],[0..L] OF BOOLEAN;
      CAMBIO = ARRAY[1..n],[0..L] OF CARDINAL;
```

```
PROCEDURE Procedencia(VAR P:MATRIZ; VAR C:CAMBIO; L,n:CARDINAL;
                      T:MONEDAS);
   VAR i,j:CARDINAL;
BEGIN
   FOR i:=1 TO n DO
      P[i,0]:=FALSE;
      C[i,0]:=0
   END;
   FOR i:=1 TO n DO
      FOR j:=1 TO L DO
         IF (i=1) AND (j<T[i]) THEN
            C[i,j]:=MAX(CARDINAL);
            P[i,j]:=FALSE
         ELSIF i=1 THEN
            C[i,j]:=1 + C[i,j-T[1]];
            P[i,j]:=TRUE
         ELSIF j<T[i] THEN
            C[i,j]:=C[i-1,j];
            P[i,j]:=FALSE
         ELSE
            C[i,j]:=Min2(C[i-1,j],1+C[i,j-T[i]]);
            P[i,j]:=(C[i,j] <> C[i-1,j])
         END
      END
   END
END Procedencia;
```

La solución se encuentra recorriendo la tabla *P* en sentido inverso, comenzando por el valor *P*[*n,L*], como se muestra en el siguiente algoritmo:

```
TYPE NUMMONEDAS = ARRAY[1..n] OF CARDINAL;

PROCEDURE Monedas(P:MATRIZ;C:CAMBIO;L,n:CARDINAL;T:MONEDAS):NUMMONEDAS;
   VAR monedas:NUMMONEDAS; ind,i,j:CARDINAL;
BEGIN
   i:=n; j:=L;
   FOR ind:=1 TO n DO monedas[ind]:=0 END;
   WHILE (i<>0) AND (j<>0) DO
      IF P[i,j]=FALSE THEN DEC(i)
      ELSE monedas[i]:=monedas[i]+1; j:=j-T[i]
      END
   END;
   IF i=0 THEN monedas[1]:=C[i,j]+monedas[1] END;
   RETURN monedas
END Monedas;
```

La resolución del problema requiere por una parte el algoritmo *Procedencia* para la construcción de la tabla P que permite conocer el número de monedas, y por tanto cuando se trata de n tipos de monedas diferentes y una cantidad L, su orden de complejidad es $O(nL)$.

Además, para conocer la solución es necesario recorrer la tabla desde la posición $P[n,L]$ hasta $P[0,0]$ a través de $n - 1$ pasos de orden de complejidad $O(n)$ para llegar a la fila 1. Asimismo hay que tener en cuenta los pasos que hay que dar de derecha a izquierda hasta llegar a la columna cero, que viene dado por el número de monedas que intervienen en la solución, es decir, por el valor de $C[n,L]$. Podemos concluir por tanto que su complejidad es $O(n+C[n,L])$.

5.10 EL ALGORITMO DE DIJKSTRA

Sea un grafo ponderado $g = (V,A)$, donde V es su conjunto de vértices, A el conjunto de arcos y sea $L[i,j]$ su matriz de adyacencia. Queremos calcular el camino más corto entre un vértice v_i tomado como origen y cada vértice restante v_j del grafo.

El clásico algoritmo de Dijkstra trabaja en etapas, en donde en cada una de ellas va añadiendo un vértice al conjunto D que representa aquellos vértices para los que se conoce su distancia al vértice origen. Inicialmente el conjunto D contiene sólo al vértice origen.

Aún siendo el algoritmo de Dijkstra un claro ejemplo de algoritmo ávido, nos preguntamos si puede ser planteado como un algoritmo de Programación Dinámica, y si de ello se deriva alguna ventaja.

Solución (☺)

La técnica de la Programación Dinámica tiene grandes ventajas, y una de ellas es la de ofrecer un diseño adecuado y eficiente a todos los problemas que puedan plantearse de forma recursiva y cumplan el principio del óptimo.

Así, es posible plantear el algoritmo de Dijkstra en términos de la Programación Dinámica, y de esta forma aprovechar el método de diseño y las ventajas que esta técnica ofrece.

En primer lugar, observemos que es posible aplicar el principio de óptimo en este caso: si en el camino mínimo de v_i a v_j está un vértice v_k como intermedio, los caminos parciales de v_i a v_k y de v_k a v_j han de ser a su vez mínimos.

Llamaremos $D(j)$ al vector que contiene el camino mínimo desde el vértice origen $i = 1$ a cada vértice v_j, $2 \le j \le n$, siendo n el número de vértices. Inicialmente D contiene los arcos $L(1,j)$, o bien ∞ si no existe el arco. A continuación, y para cada vértice v_k del grafo con $k \neq 1$, se repetirá:

$$D(j) = \underset{1<k\le n}{Min}\{D(j), D(k) + L(k, j)\} \qquad [5.3]$$

De esta forma el algoritmo que resuelve el problema puede ser implementado como sigue:

```
CONST n = ...; (* numero de vertices del grafo *)
TYPE MATRIZ = ARRAY [1..n],[1..n] OF CARDINAL;
     MARCA = ARRAY [1..n] OF BOOLEAN;(* elementos ya considerados*)
     SOLUCION = ARRAY [2..n] OF CARDINAL;

PROCEDURE Dijkstra(VAR L:MATRIZ;VAR D:SOLUCION);
   VAR i,j,menor,pos,s:CARDINAL; S:MARCA;
BEGIN
   FOR i:=2 TO n DO
      S[i]:=FALSE;
      D[i]:=L[1,i]
   END;
   S[1]:=TRUE;
   FOR i:=2 TO n-1 DO
      menor:=Menor(D,S,pos);
      S[pos]:=TRUE;
      FOR j:=2 TO n DO
         IF NOT(S[j]) THEN
            D[j]:= Min2(D[j],D[pos]+L[pos,j])
         END;
      END;
   END
END Dijkstra;
```

La función *Menor* es la que calcula el mínimo de la expresión en recurrencia [5.3] que define la solución del problema:

```
PROCEDURE Menor(VAR D:SOLUCION; VAR S:MARCA; VAR pos:CARDINAL)
                                                          :CARDINAL;
   VAR menor,i:CARDINAL;
BEGIN
   menor:=MAX(CARDINAL); pos:=1;
   FOR i:=2 TO n DO
      IF NOT(S[i]) THEN
         IF D[i]<menor THEN
            menor:=D[i]; pos:=i
         END
      END
   END;
   RETURN menor
END Menor;
```

La complejidad temporal del algoritmo es de orden $O(n^2)$, siendo de orden $O(n)$ su complejidad espacial. No ganamos sustancialmente en eficiencia mediante el uso de esta técnica frente al planteamiento ávido del algoritmo, pero sin embargo sí

ganamos en sencillez del diseño e implementación de la solución a partir del planteamiento del problema.

5.11 EL ALGORITMO DE FLOYD

Sea *g* un grafo dirigido y ponderado. Para calcular el menor de los caminos mínimos entre dos vértices cualesquiera del grafo, podemos aplicar el algoritmo de Dijkstra a todos los pares posibles y calcular su mínimo, o bien aplicamos el siguiente algoritmo (Floyd) que, dada la matriz *L* de adyacencia del grafo *g*, calcula una matriz *D* con la longitud del camino mínimo que une cada par de vértices:

```
CONST n = ...; (* numero de vertices del grafo *)
TYPE  MATRIZ = ARRAY[1..n],[1..n] OF CARDINAL;

PROCEDURE Floyd (VAR L,D:MATRIZ);
   VAR i,j,k: CARDINAL;
BEGIN
   FOR i:=1 TO n DO FOR j:=1 TO n DO
        D[i,j]:=L[i,j]
   END END;
   FOR k:=1 TO n DO
      FOR i:=1 TO n DO
         FOR j:=1 TO n DO
            D[i,j]:=Min2(D[i,j],D[i,k]+D[k,j])
         END
      END
   END
END Floyd;
```

Nos planteamos si tal algoritmo puede ser considerado o no de Programación Dinámica, es decir, si reune las características esenciales de ese tipo de algoritmos.

Solución (☺)

Este algoritmo puede ser considerado de Programación Dinámica ya que es aplicable el principio de óptimo, que puede enunciarse para este problema de la siguiente forma: si en el camino mínimo de v_i a v_j, v_k es un vértice intermedio, los caminos de v_i a v_k y de v_k a v_j han de ser a su vez caminos mínimos. Por lo tanto, puede plantearse la relación en recurrencia que resuelve el problema como:

$$D_k(i,j) = \underset{k\geq 1}{Min}\{D_{k-1}(i,k), D_{k-1}(k,j)\}$$

Tal ecuación queda resuelta mediante el algoritmo presentado que, siguiendo el esquema de la Programación Dinámica, utiliza una matriz para evitar la repetición de los cálculos. Con ello consigue que su complejidad temporal sea de orden $O(n^3)$ debido al triple bucle anidado en cuyo interior hay tan sólo operaciones constantes.

5.12 EL ALGORITMO DE WARSHALL

Al igual que ocurre con el algoritmo de Floyd descrito en el apartado anterior, estamos interesados en encontrar caminos entre cada dos vértices de un grafo. Sin embargo, aquí no nos importa su longitud, sino sólo su existencia. Por tanto, lo que deseamos es diseñar un algoritmo que permita conocer si dos vértices de un grafo están conectados o no, lo que nos llevaría al cierre transitivo del grafo.

Solución (☺)

Para un grafo $g = (V,A)$ cuya matriz de adyacencia sea L, el algoritmo pedido puede ser implementado como sigue:

```
CONST n = ...; (* numero de vertices del grafo *)
TYPE  MATRIZ = ARRAY[1..n],[1..n] OF BOOLEAN;

PROCEDURE Warshall (VAR L,D:MATRIZ);
   VAR i,j,k: CARDINAL;
BEGIN
   FOR i:=1 TO n DO
      FOR j:=1 TO n DO
         D[i,j]:=L[i,j]
      END
   END;
   FOR k:=1 TO n DO
      FOR i:=1 TO n DO
         FOR j:=1 TO n DO
            D[i,j]:=D[i,j] OR (D[i,k] AND D[k,j])
         END
      END
   END
END Warshall;
```

Tras la ejecución del algoritmo, la solución se encuentra en la matriz D, que verifica que $D[i,j] = TRUE$ si y sólo si existe un camino entre los vértices i y j. Obsérvese la similitud entre este algoritmo y el de Floyd. En cuanto a su complejidad, podemos afirmar que es de orden $O(n^3)$ debido al triple bucle anidado que posee, en cuyo interior sólo se realizan operaciones constantes.

5.13 ORDENACIONES DE OBJETOS ENTRE DOS RELACIONES

Dados n objetos, queremos calcular el número de ordenaciones posibles según las relaciones "<" e "=". Por ejemplo, dados tres objetos A, B y C, el algoritmo debe determinar que existen 13 ordenaciones distintas: A=B=C, A=B<C, A<B=C, A<B<C, A<C<B, A=C<B, B<A=C, B<A<C, B<C<A, B=C<A, C<A=B, C<A<B y C<B<A.

Solución (✍)

Llamaremos C_n al número de ordenaciones posible con n objetos. Si $1 \le k \le n$, podemos expresar C_k como:

$$C_k = I_0^{(k)} + I_1^{(k)} + ... + I_{k-1}^{(k)} = \sum_{j=0}^{k-1} I_j^{(k)}$$

siendo $I_j^{(k)}$ el número de formas posibles de poner k elementos en donde hay j símbolos "=" (es decir, $k - j - 1$ símbolos distintos), con $0 \le j < k$, $1 \le k \le n$. Vamos a tratar de expresar C_k en función de C_{k-1}. Para ello, supongamos que ya tenemos

$$C_{k-1} = \sum_{j=0}^{k-2} I_j^{(k-1)}$$

y añadimos un nuevo elemento. Pueden ocurrir dos casos: que sea distinto a todos los k–1 elementos anteriores, o bien que sea igual a uno de ellos. Entonces, la expresión de C_k va a venir dada por:

$$C_k = (kI_0^{(k-1)} + (k-1)I_1^{(k-1)} + (k-2)I_2^{(k-1)} + ... + 2I_{k-2}^{(k-1)}) +$$
$$((k-1)I_0^{(k-1)} + (k-2)I_1^{(k-1)} + ... + 2I_{k-3}^{(k-1)} + I_{k-2}^{(k-1)}) = \sum_{j=0}^{k-2} (2(k-j)-1)I_j^{(k-1)}$$

Con esto, tenemos C_k en función de los $I_j^{(k-1)}$, es decir, de los componentes del caso anterior. Ahora bien, es posible también relacionar los $I^{(k)}$ con los $I^{(k-1)}$ de la siguiente manera:

$$I_0^{(k)} = kI_0^{(k-1)}$$
$$I_1^{(k)} = (k-1)I_1^{(k-1)} + (k-1)I_0^{(k-1)}$$
$$I_2^{(k)} = (k-2)I_2^{(k-1)} + (k-2)I_1^{(k-1)}$$
$$...$$
$$I_{k-2}^{(k)} = 2I_{k-2}^{(k-1)} + 2I_{k-3}^{(k-1)}$$
$$I_{k-1}^{(k)} = I_{k-2}^{(k-1)}$$

cuyas condiciones iniciales son $I_0^{(2)} = 2, I_1^{(2)} = 1$. Esto también puede expresarse como sigue:

$$I_j^{(k)} = (k-j)(I_j^{(k-1)} + I_{j-1}^{(k-1)}) \quad \text{para } 0 \le j \le k-1 \text{ y } 2 \le k \le n$$
$$I_0^{(2)} = 2$$
$$I_1^{(2)} = 1$$
$$I_{-1}^{(k)} = 0$$
$$I_k^{(k)} = 0$$

Así, el problema puede resolverse calculando cada $I_j^{(n)}$ ($0 \le j \le n$–1), para finalmente calcular C_n mediante la expresión:

$$C_n = \sum_{j=0}^{n-1} I_j^{(n)}$$

El algoritmo que implementa tal estrategia es el siguiente:

```
CONST n = ...; (* numero de objetos *)
TYPE VECTOR = ARRAY [-1..n] OF INTEGER;

PROCEDURE Ordenaciones(VAR I:VECTOR; n:INTEGER):INTEGER;
   VAR x,y,s,k,j:INTEGER;
BEGIN
   IF n <= 1 THEN RETURN n END; (* caso base *)
   FOR j:=-1 TO n DO I[j]:=0 END; (* inicializamos el vector I *)
   I[0]:=1; x:=0;
   FOR k:=2 TO n DO
      FOR j:=0 TO n-1 DO
         IF j>1 THEN I[j-2]:=y END;
         y:=x;
         x:=(k-j)*(I[j]+I[j-1])
      END;
      I[n-2]:=y; I[n-1]:=x;
   END;
   s:=0;
   FOR j:=0 TO n-1 DO
      s:=s+I[j]
   END;
   RETURN s
END Ordenaciones;
```

Respecto a su complejidad espacial, tan sólo utiliza el vector *I* por lo que es de orden O(n). Y en cuanto a su complejidad temporal, el algoritmo utiliza dos bucles anidados para el cálculo de los valores del vector, por lo que podemos afirmar que su orden es O(n^2).

5.14 EL VIAJANTE DE COMERCIO

¿Podría aplicarse la técnica de Programación Dinámica al problema del viajante de comercio? Recordemos que este problema consistía en encontrar el camino sin ciclos de menor peso de un grafo ponderado que recorra todos los vértices y vuelva al vértice original.

Solución (😐)

En primer lugar, vamos a plantear la solución del problema como una sucesión de decisiones que verifique el principio de óptimo. La idea va a consistir en construir una solución mediante la búsqueda sucesiva de recorridos mínimos de tamaño 1, 2, 3, etc.

Representando el problema a través de un grafo $g = (V,A)$, y siendo L su matriz de adyacencia, cada recorrido del viajante que parte del vértice v_1 estará formado por un arco (v_1,v_k) para algún vértice v_k perteneciente a $V-\{v_1\}$ y un camino de v_k al vértice v_1.

Pero si el recorrido es óptimo también ha de ser óptimo el camino de v_k al vértice v_1, pues si no lo fuese llegaríamos a una contradicción. Si no lo fuese y existiera otro camino mejor, incluyendo a éste en el recorrido original obtendríamos un camino mejor que el óptimo, lo cual es imposible. Por tanto, se cumple el principio de óptimo.

Planteemos entonces la relación en recurrencia. Para ello, llamaremos $D(v_i,S)$ a la longitud del camino mínimo que partiendo del vértice v_i pasa por todos los vértices del conjunto S y vuelve al vértice v_i. La solución al problema del viajante vendrá dada entonces por $D(v_1,V-\{v_1\})$:

$$D(v_i, V-\{v_1\}) = \underset{2\le k\le n}{Min}\{L(v_1,v_k) + D(v_k, V-\{v_1,v_k\}\}$$

Generalizando para comenzar el recorrido desde cualquier vértice:

$$D(v_i, V) = \underset{i\notin V,\, j\in V}{Min}\{L(v_i,v_j) + D(v_j, V-v_j)\}$$

$$D(v_i, \{\}) = L(v_i, v_1) \text{ para } 1 \le i \le n$$

Obsérvese la diferencia que existe entre la estrategia de este algoritmo y los que tratamos de diseñar siguiendo dos técnicas ávidas (ver el problema 4.4 del capítulo anterior). En los algoritmos ávidos se ha de escoger una de las posibles opciones en cada paso, y una vez tomada –o descartada–, ya no vuelve a ser considerada nunca. Son algoritmos que no guardan "historia", y por tanto no siempre funcionan. Sin embargo, en la Programación Dinámica la solución al problema total se va construyendo de otra forma: a partir de las soluciones óptimas para problemas más pequeños.

No obstante, el diseño aquí realizado tiene un serio inconveniente: su implementación utilizando una estructura de datos que permita reutilizar los cálculos. Tal estructura debería contener las soluciones intermedias necesarias para el cómputo de $D(v_1,V-\{v_1\})$, pero estas son demasiadas.

En efecto, la tabla debe tener n filas, y 2^n columnas, pues éste es el cardinal de las partes del conjunto V, que son todas las posibilidades que puede tomar el segundo parámetro de D en su definición.

Por tanto, sí existe una solución al problema del viajante utilizando Programación Dinámica, pero no sólo no consigue mejorar la eficiencia de su versión clásica mediante Vuelta Atrás (véase el siguiente capítulo), sino que tampoco ofrece una mejora en cuanto a la simplicidad de su implementación.

5.15 HORARIOS DE TRENES

Una compañía de ferrocarriles sirve n estaciones $S_1,...,S_n$ y trata de mejorar su servicio al cliente mediante terminales de información. Dadas una estación origen S_o y una estación destino S_d, un terminal debe ofrecer (inmediatamente) la información sobre el horario de los trenes que hacen la conexión entre S_o y S_d y que minimizan el tiempo de trayecto total.

Necesitamos implementar un algoritmo que realice esta tarea a partir de la tabla con los horarios, suponiendo que las horas de salida de los trenes coinciden con las de sus llegadas (es decir, que no hay tiempos de espera) y que, naturalmente, no todas las estaciones están conectadas entre sí por líneas directas; así, en muchos casos hay que hacer transbordos aunque se supone que tardan tiempo cero en efectuarse.

Solución (☺)

Llamaremos $T(i,j,V)$ al tiempo del trayecto mínimo para ir de la estación de origen i a la estación destino j, pudiendo utilizar como estaciones intermedias las contenidas en el conjunto V, y llamaremos $L(i,j)$ al tiempo del trayecto directo de i a j, siendo ∞ si esta conexión no existe. De forma análoga a como razonábamos para el problema de los embarcaderos sobre el río (apartado 5.6), podemos plantear la solución al problema mediante una ecuación en recurrencia:

$$T(i,j,V) = \underset{k \in V, k \neq i, k \neq j}{Min} \{L(i,j), T(k,j,V-k) + L(i,k)\}$$

Esta ecuación trata de comprobar si es más beneficioso ir de forma directa o a través de cada uno de los posibles caminos. Esto hay que hacerlo para cada par (i,j) desde 1 hasta n. Obsérvese cómo llegamos a ella por el principio de óptimo pues cualquier subtrayecto de un trayecto óptimo a de ser, a su vez, óptimo.

Podemos representar nuestro problema mediante un grafo siendo las estaciones los vértices del grafo y las aristas las conexiones entre dos estaciones, pudiendo no existir si no hay trayecto directo entre ambas. La solución al problema se puede alcanzar resolviendo el algoritmo de Dijkstra para cada vértice del grafo, puesto que tal algoritmo calcula los caminos mínimos desde un único origen hasta los demás vértices en un grafo.

Respecto a su complejidad, ésta coincide con la del algoritmo de Dijkstra, que es aceptable para un número n de estaciones razonablemente grande.

5.16 LA MOCHILA (0,1)

En el apartado 4.7 del capítulo anterior se planteó el problema de la Mochila (0,1), que consistía en decidir de entre n objetos de pesos p_1, p_2,..., p_n y beneficios b_1, b_2,..., b_n, cuáles hay que incluir en una mochila de capacidad M sin superar dicha capacidad y de forma que se maximice la suma de los beneficios de los elementos escogidos. Los algoritmos ávidos planteados entonces no conseguían resolver el problema. Nos cuestionamos aquí si este problema admite una solución mediante Programación Dinámica.

Solución (😐)

Para encontrar un algoritmo de Programación Dinámica que lo resuelva, primero hemos de plantear el problema como una secuencia de decisiones que verifique el principio de óptimo. De aquí seremos capaces de deducir una expresión recursiva de la solución. Por último habrá que encontrar una estructura de datos adecuada que permita la reutilización de los cálculos de la ecuación en recurrencia, consiguiendo una complejidad mejor que la del algoritmo puramente recursivo.

Siendo M la capacidad de la mochila y disponiendo de n elementos, llamaremos $V(i,p)$ al valor máximo de la mochila con capacidad p cuando consideramos i objetos, con $0 \le p \le M$ y $1 \le i \le n$. La solución viene dada por el valor de $V(n,M)$. Denominaremos d_1, d_2, ..., d_n a la secuencia de decisiones que conducen a obtener $V(n,M)$, donde cada d_i podrá tomar uno de los valores 1 ó 0, dependiendo si se introduce o no el i-ésimo elemento. Podemos tener por tanto dos situaciones distintas:

- Que $d_n = 1$. La subsecuencia de decisiones d_1, d_2, ..., d_{n-1} ha de ser también óptima para el problema $V(n-1,M-p_n)$, ya que si no lo fuera y existiera otra subsecuencia e_1, e_2, ..., e_{n-1} óptima, la secuencia e_1, e_2, ..., e_{n-1}, d_n también sería óptima para el problema $V(n,M)$ lo que contradice la hipótesis.
- Que $d_n = 0$. Entonces la subsecuencia decisiones d_1, d_2, ..., d_{n-1} ha de ser también óptima para el problema $V(n-1,M)$.

Podemos aplicar por tanto el principio de óptimo para formular la relación en recurrencia. Teniendo en cuenta que en la mochila no puede introducirse una fracción del elemento sino que el elemento i se introduce o no se introduce, en una situación cualquiera $V(i,p)$ tomará el valor mayor entre $V(i-1,p)$, que indica que el elemento i no se introduce, y $V(i-1,p-p_i)+b_i$, que es el resultado de introducirlo y de ahí que la capacidad ha de disminuir en p_i y el valor aumentar en b_i, y por tanto podemos plantear la solución al problema mediante la siguiente ecuación:

$$V(i,p) = \begin{cases} 0 & \text{si } i = 0 \text{ y } p \ge 0 \\ -\infty & \text{si } p < 0 \\ Max\{V(i-1,p), V(i-1,p-p_i)+b_i\} & \text{en otro caso.} \end{cases}$$

Estos valores se van almacenando en una tabla construida mediante el algoritmo:

```
TYPE TABLA = ARRAY[1..n],[0..M] OF CARDINAL;
     DATOS = RECORD peso,valor:CARDINAL; END;
     TIPOOBJETO = ARRAY[1..n] OF DATOS;

PROCEDURE Mochila (i,p:CARDINAL; VAR obj:TIPOOBJETO):CARDINAL;
  VAR elem,cap:CARDINAL; V:TABLA;
BEGIN
  FOR elem:=1 TO i DO
    V[elem,0]:=0;
    FOR cap:=1 TO p DO
      IF (elem=1) AND (cap<obj[1].peso) THEN
        V[elem,cap]:=0
      ELSIF elem=1 THEN
        V[elem,cap]:=obj[1].valor
      ELSIF cap<obj[elem].peso THEN
        V[elem,cap]:=V[elem-1,cap]
      ELSE V[elem,cap]:=
   Max2(V[elem-1,cap],obj[elem].valor+V[elem-1,cap-obj[elem].peso])
      END
    END
  END;
  RETURN V[i,p]
END Mochila;
```

El problema se resuelve invocando a la función con *i*=*n*, *p*=*M*. La complejidad del algoritmo viene determinada por la construcción de una tabla de dimensiones $n \times M$ y por tanto su tiempo de ejecución es de orden de complejidad O(*nM*). La función *Max*2 es la que calcula el máximo de dos valores.

Si además del valor de la solución óptima se desea conocer los elementos que son introducidos, es decir, la composición de la mochila, es necesario añadir al algoritmo la construcción de una tabla de valores lógicos que indique para cada valor *E*[*i*,*j*] si el elemento *i* forma parte de la solución para la capacidad *j* o no:

```
TYPE ENTRAONO = ARRAY[1..n],[0..P] OF BOOLEAN;

PROCEDURE Max2especial(x,y:CARDINAL;VAR esmenorx:BOOLEAN):CARDINAL;
BEGIN
  IF x>y THEN
    esmenorx:=FALSE;
    RETURN x
  ELSE
    esmenorx:=TRUE;
    RETURN y
  END
END Max2especial;
```

```
PROCEDURE Mochila2(i,p:CARDINAL;obj:TIPOOBJETO;VAR E:ENTRAONO)
                                                          :CARDINAL;
   VAR elem,cap:CARDINAL; V:TABLA;
BEGIN
   FOR elem:=1 TO i DO
      V[elem,0]:=0;
      FOR cap:=1 TO p DO
         IF (elem=1) AND (cap<obj[1].peso) THEN
               V[elem,cap]:=0;
               E[elem,cap]:=FALSE
         ELSIF elem=1 THEN
               V[elem,cap]:=obj[1].valor;
               E[elem,cap]:=TRUE
         ELSIF cap<obj[elem].peso THEN
               V[elem,cap]:=V[elem-1,cap];
               E[elem,cap]:=FALSE
         ELSE  V[elem,cap]:=Max2especial(V[elem-1,cap],
          obj[elem].valor+V[elem-1,elem-obj[elem].peso],E[elem,cap]);
         END
      END
   END;
   RETURN V[i,p]
END Mochila2;
```

Por otra parte, es necesario construir un algoritmo que interprete los valores de esta tabla para componer la solución. Esto se realizará recorriéndola en sentido inverso desde los valores $i = n$, $j = M$ hasta $i = 0$, $j = 0$, mediante el siguiente algoritmo:

```
TYPE SOLUCION = ARRAY[1..n] OF CARDINAL;

PROCEDURE Componer(VAR sol:SOLUCION);
   VAR elem,cap:CARDINAL;
BEGIN
   FOR elem:=1 TO n DO (* inicializa solucion *)
      sol[elem]:=0
   END;
   elem:= n; cap:= M;
   WHILE (elem<>0) AND (cap<>0) DO
      IF entra[elem,cap] THEN
         sol[elem]:=1;
         cap:=cap-obj[elem].peso
      END;
      DEC(elem)
   END
 END Componer;
```

5.17 LA MOCHILA (0,1) CON MÚLTIPLES ELEMENTOS

Este problema se basa en el de la Mochila (0,1) pero en vez de existir n objetos distintos, de lo que disponemos es de n tipos de objetos distintos. Con esto, de un objeto cualquiera podemos escoger tantas unidades como deseemos.

Este problema se puede formular también como una modificación al problema de la Mochila (0,1), en donde sustituimos el requerimiento de que $x_i = 0$ ó $x_i = 1$, por el que x_i sean números naturales. Como en el problema original, deseamos maximizar la suma de los beneficios de los elementos introducidos, sujeta a la restricción de que éstos no superen la capacidad de la mochila.

Solución (😐)

Para encontrar un algoritmo de Programación Dinámica que resuelva el problema, primero hemos de plantearlo como una secuencia de decisiones que verifiquen el principio del óptimo. De aquí seremos capaces de deducir una expresión recursiva de la solución. Por último habrá que encontrar una estructura de datos adecuada que permita la reutilización de los cálculos de la ecuación en recurrencia, consiguiendo una complejidad mejor que la del algoritmo puramente recursivo.

Con esto en mente, llamaremos $V(i,p)$ al valor máximo de una mochila de capacidad p y con i tipos de objetos. Iremos decidiendo en cada paso si introducimos o no un objeto de tipo i. Por consiguiente, para calcular $V(i,p)$ existen dos opciones en cada paso:

- No introducir ninguna unidad del tipo i, con lo cual el valor de la mochila $V(i,p)$ es el calculado para $V(i–1,p)$.
- Introducir una unidad más del objeto i lo cual indica que el valor de $V(i,p)$ será el resultado obtenido para $V(i,p–p_i)$ más el valor del objeto v_i, con lo cual se verifica que $V(i,p) = V(i,p–p_i) + b_i$.

Esto nos permite establecer la siguiente relación en recurrencia para $V(i,p)$:

$$V(i,p) = \begin{cases} 0 & \text{si} \quad p = 0 \\ (p \div p_i)b_i & \text{si} \quad i = 1 \\ V(i-1,p) & \text{si} \quad p < p_i \\ Max\{V(i-1,p), V(i,p-p_i)+b_i\} & \text{en otro caso.} \end{cases}$$

Utilizaremos una matriz nxM para almacenar los valores de V que vayamos obteniendo y así no repetir cálculos. El algoritmo que resuelve el problema es el siguiente:

```
TYPE TABLA = ARRAY[1..n],[0..M] OF CARDINAL;
     DATOS = RECORD peso,valor:CARDINAL; END;
     TIPOOBJETO = ARRAY[1..n] OF DATOS;
```

```
PROCEDURE Mochila3(i,p:CARDINAL;VAR obj:TIPOOBJETO):CARDINAL;
   VAR elem,cap:CARDINAL; V:TABLA;
BEGIN
   FOR elem:=1 TO i DO (* condiciones iniciales *)
      V[elem,0]:=0
   END;
   FOR cap:=1 TO p DO
      V[1,cap]:=(cap DIV obj[1].peso)*obj[1].valor
   END;
   FOR elem:=2 TO num DO
      FOR cap:=1 TO p DO
         IF (cap < p[elem]) THEN V[elem,cap]:=V[elem-1,cap]
         ELSE V[elem,cap]:= Max2(V[elem-1,cap],
                        (V[elem,cap-obj[elem].peso]+obj[elem].valor))
         END
      END
   END;
   RETURN V[i,p]
END Mochila3;
```

La complejidad del algoritmo es la que corresponde a la construcción de la tabla, es decir O(nM).

5.18 LA MULTIPLICACIÓN ÓPTIMA DE MATRICES

Necesitamos calcular la matriz producto M de n matrices dadas $M = M_1M_2...M_n$ minimizando el número total de multiplicaciones escalares a realizar. Este problema ya fue planteado en el capítulo anterior, en donde vimos que los algoritmos ávidos presentados no encontraban solución en todos los casos.

Nos preguntamos ahora si existe un algoritmo que lo resuelva utilizando la Programación Dinámica.

Solución (☺)

En primer lugar, vamos a suponer como hacíamos en el capítulo anterior que cada M_i es de dimensión d_{i-1}xd_i ($1 \le i \le n$), y por tanto realizar la multiplicación $M_i\,M_{i+1}$ va a requerir un total de $d_{i-1}d_id_{i+1}$ operaciones. Llamaremos $M(i,j)$ al número mínimo de multiplicaciones escalares necesarias para el cómputo del producto de $M_i\,M_{i+1}\,...\,M_j$ con $1 \le i \le j \le n$. Por consiguiente, la solución al problema planteado coincidirá con $M(1,n)$.

Para plantear la ecuación en recurrencia que define la solución, supongamos que asociamos las matrices de la siguiente manera:

$$(M_i\,M_{i+1}\,...\,M_k)\,(M_{k+1}\,M_{k+2}\,...\,M_j).$$

Aplicando el principio de óptimo, el valor de $M(i,j)$ será la suma del número de multiplicaciones escalares necesarias para calcular el producto de las matrices $M_i\ M_{i+1}...M_k$, que corresponde a $M(i,k)$, más el número de multiplicaciones escalares para el producto de las matrices $M_{k+1}M_{k+2}\ ...M_j$ que es $M(k+1,j)$, más el producto que corresponde a la última multiplicación entre las matrices de dimensiones
$(d_{i-1}\ d_k)$ y $(d_k\ \ d_j)$ es decir, $d_{i-1}d_kd_j$. En consecuencia, el valor de $M(i,j)$ para la asociación anteriormente expuesta viene dado por la expresión:

$$M(i,j) = M(i,k) + M(k+1,j) + d_{i-1}d_k\ d_j.$$

Pero k puede tomar cualquier valor entre i y j–1, y por tanto $M(i,j)$ deberá escoger el más favorable de entre todos ellos, es decir:

$$M(i,j) = \underset{i\le k<j}{Min}\{M(i,k) + M(k+1,j) + d_{i-1}d_kd_j\},$$

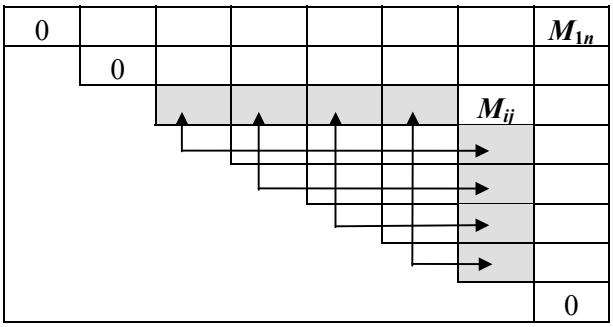
lo que nos lleva a la siguiente relación en recurrencia:

$$M(i,j) = \begin{cases} 0 & \text{si } i = j \\ \underset{i\le k<j}{Min}\{M(i,k) + M(k+1,j) + d_{i-1}d_kd_j\} & \text{en otro caso.} \end{cases} \qquad [5.4]$$

Para resolver tal ecuación en un tiempo de complejidad polinómico es necesario crear una tabla en la que se vayan almacenando los valores $M(i,j)$ $(1 \le i \le j \le n)$ y que permita a partir de las condiciones iniciales reutilizar los valores calculados en los pasos anteriores.

Esta tabla se irá rellenando por diagonales sabiendo que los elementos de la diagonal principal son todos cero. Cada elemento $M[i,j]$ será el valor mínimo de entre todos los pares $(M[i,k] + M[k+1,j])$ señalados con la línea de doble flecha en la siguiente figura, más la aportación correspondiente a la última multiplicación $(d_{i-1}d_kd_j)$. Los valores que requiere el cálculo de $M[i,j]$ y que el algoritmo reutiliza para conseguir una tiempo de ejecución aceptable se encuentran sombreados:

Al ir rellenando la tabla por diagonales se asegura que esta información $(M[i,k],M[k+1,j])$ está disponible cuando se necesita, pues cada $M[i,j]$ utiliza para su cálculo todos los elementos anteriores de su fila y todos los de su columna por debajo suya.



Rellenada la tabla, la solución la podemos encontrar en el extremo superior derecho, que nos indica el número de multiplicaciones escalares buscado, $M[1,n]$.

Si además queremos conocer cómo es la asociación que corresponde a este óptimo es necesario conservar para cada elemento $M[i,j]$ el valor de k para el cual la expresión $M[i,k] + M[k+1,j] + d_{i-1}d_kd_j$ es mínima, construyendo otra tabla que denominamos *Factor*.

El siguiente algoritmo *Matriz* es de creación de las tablas *M* y *Factor*. En la tabla *M* se almacenan los valores del número mínimo de multiplicaciones y en la tabla *Factor* la información necesaria para construir la asociación óptima.

```
TYPE MATRIZ = ARRAY [1..n],[1..n] OF CARDINAL;
     ORDEN  = ARRAY [0..n] OF CARDINAL;(* dimensiones *)

PROCEDURE Matriz(VAR d:ORDEN;n:CARDINAL;VAR M,Factor:MATRIZ);
   VAR i,diagonal:CARDINAL;
BEGIN
   FOR i:=1 TO n DO
      M[i,i]:=0
   END;
   FOR diagonal:=1 TO n-1 DO
      FOR i:=1 TO n-diagonal DO
         M[i,i+diagonal]:=
               Minimo(d,M,i,i+diagonal,Factor[i,i+diagonal]);
      END
   END
END Matriz;
```

La función *Minimo* es la que calcula el mínimo de la expresión en recurrencia [5.4], y devuelve no sólo el valor de este mínimo, sino el valor de k para el que se alcanza (mediante el parámetro *k1*):

```
PROCEDURE Minimo(VAR d:ORDEN;VAR M:MATRIZ;i,j:CARDINAL;
                 VAR k1:CARDINAL):CARDINAL;
   VAR aux,k,min:CARDINAL;
BEGIN
   min:=MAX(CARDINAL);
   FOR k:=i TO j-1 DO
      aux:=M[i,k]+M[k+1,j]+d[i-1]*d[k]*d[j];
      IF aux<min THEN min:=aux; k1:=k END
   END;
   RETURN min
END Minimo;
```

Observando el procedimiento *Matriz* vemos que existe un bucle externo que se repite desde *diagonal* = 1 hasta $n - 1$, y en su interior un bucle dependiente de la iteración estudiada y del valor de *diagonal*, y que se ejecuta desde 1 hasta $n - diagonal$. En el interior de este bucle hay una llamada al procedimiento *Minimo* que tiene una complejidad del orden del valor de la *diagonal*, y por tanto el tiempo de ejecución del algoritmo es:

$$\sum_{diagonal=1}^{n-1}(n-diagonal)diagonal = n\sum_{diagonal=1}^{n-1}diagonal - \sum_{diagonal=1}^{n-1}diagonal^2 = (n^3-n)/6$$

por lo que concluimos que su complejidad temporal es de orden $O(n^3)$. Por otro lado, la complejidad espacial del algoritmo es de orden $O(n^2)$.

En caso que deseemos reconstruir la solución a partir de la tabla *Factor*, el siguiente procedimiento muestra por pantalla la forma de multiplicar las matrices para obtener ese valor mínimo:

```
PROCEDURE EscribeOrden(VAR Factor:MATRIZ;i,j:CARDINAL);
   VAR k:CARDINAL;
BEGIN
   IF i=j THEN
      WrStr('M');
      WrCard(i,0)
   ELSE
      k:=Factor[i,j];
      WrStr('(');
      EscribeOrden(Factor,i,k);
      WrStr('*');
      EscribeOrden(Factor,k+1,j);
      WrStr(')')
   END
END EscribeOrden;
```

El algoritmo aquí presentado es capaz de encontrar el valor de la solución óptima junto con una de las formas de obtenerla. Sin embargo, existen casos en donde puede haber más de una forma de multiplicar las matrices para obtener el valor óptimo, como muestra el siguiente ejemplo.

Sean las matrices M_1 (10x10), M_2 (10x50) y M_3 (50x50). Existen dos formas de asociarlas para multiplicarlas, y en ambos casos obtenemos:

$$(M_1M_2)M_3 = 10\cdot10\cdot50 + 10\cdot50\cdot50 = 30000$$
$$M_1(M_2M_3) = 10\cdot50\cdot50 + 10\cdot10\cdot50 = 30000$$

Es posible modificar el algoritmo para que encuentre todas las soluciones que llevan al valor óptimo, y para esto es suficiente valerse de la matriz *Factor*, si bien esta modificación reviste poco interés desde un punto de vista práctico.

Capítulo 6

VUELTA ATRÁS

6.1 INTRODUCCIÓN

Dentro de las técnicas de diseño de algoritmos, el método de Vuelta Atrás (del inglés *Backtracking*) es uno de los de más ámplia utilización, en el sentido de que puede aplicarse en la resolución de un gran número de problemas, muy especialmente en aquellos de optimización.

Los métodos estudiados en los capítulos anteriores construyen la solución basándose en ciertas propiedades de la misma; así en los algoritmos Ávidos se va contruyendo la solución por etapas, siempre avanzando sobre la solución parcial previamente calculada; o bien podremos utilizar la Programación Dinámica para dar una expresión recursiva de la solución si se verifica el principio de óptimo, y luego calcularla eficientemente. Sin embargo ciertos problemas no son susceptibles de solucionarse con ninguna de estas técnicas, de manera que la única forma de resolverlos es a través de un estudio exhaustivo de un conjunto conocido a priori de posibles soluciones, en las que tratamos de encontrar una o todas las soluciones y por tanto también la óptima.

Para llevar a cabo este estudio exhaustivo, el diseño Vuelta Atrás proporciona una manera sistemática de generar todas las posibles soluciones siempre que dichas soluciones sean susceptibles de resolverse en etapas.

En su forma básica la Vuelta Atrás se asemeja a un recorrido en profundidad dentro de un árbol cuya existencia sólo es implícita, y que denominaremos *árbol de expansión*. Este árbol es conceptual y sólo haremos uso de su organización como tal, en donde cada nodo de nivel k representa una parte de la solución y está formado por k etapas que se suponen ya realizadas. Sus hijos son las prolongaciones posibles al añadir una nueva etapa. Para examinar el conjunto de posibles soluciones es suficiente recorrer este árbol construyendo soluciones parciales a medida que se avanza en el recorrido.

En este recorrido pueden suceder dos cosas. La primera es que tenga éxito si, procediendo de esta manera, se llega a una solución (una hoja del árbol). Si lo único que buscabamos era una solución al problema, el algoritmo finaliza aquí; ahora bien, si lo que buscabamos eran todas las soluciones o la mejor de entre todas ellas, el algoritmo seguirá explorando el árbol en búsqueda de soluciones alternativas.

Por otra parte, el recorrido no tiene éxito si en alguna etapa la solución parcial construida hasta el momento no se puede completar; nos encontramos en lo que llamamos *nodos fracaso*. En tal caso, el algoritmo vuelve atrás (y de ahí su

nombre) en su recorrido eliminando los elementos que se hubieran añadido en cada etapa a partir de ese nodo. En este retroceso, si existe uno o más caminos aún no explorados que puedan conducir a solución, el recorrido del árbol continúa por ellos.

La filosofía de estos algoritmos no sigue unas reglas fijas en la búsqueda de las soluciones. Podríamos hablar de un proceso de prueba y error en el cual se va trabajando por etapas construyendo gradualmente una solución. Para muchos problemas esta prueba en cada etapa crece de una manera exponencial, lo cual es necesario evitar.

Gran parte de la eficiencia (siempre relativa) de un algoritmo de Vuelta Atrás proviene de considerar el menor conjunto de nodos que puedan llegar a ser soluciones, aunque siempre asegurándonos de que el árbol "podado" siga conteniendo todas las soluciones. Por otra parte debemos tener cuidado a la hora de decidir el tipo de condiciones (*restricciones*) que comprobamos en cada nodo a fin de detectar nodos fracaso. Evidentemente el análisis de estas restricciones permite ahorrar tiempo, al delimitar el tamaño del árbol a explorar. Sin embargo esta evaluación requiere a su vez tiempo extra, de manera que aquellas restricciones que vayan a detectar pocos nodos fracaso no serán normalmente interesantes. No obstante, y como norma de actuación general, podríamos decir que las restricciones sencillas son siempre apropiadas, mientras que las más sofisticadas que requieren más tiempo en su cálculo deberían reservarse para situaciones en las que el árbol que se genera sea muy grande.

Vamos a ver como se lleva a cabo la búsqueda de soluciones trabajando sobre este árbol y su recorrido. En líneas generales, un problema puede resolverse con un algoritmo Vuelta Atrás cuando la solución puede expresarse como una *n*-tupla $[x_1, x_2, ..., x_n]$ donde cada una de las componentes x_i de este vector es elegida en cada etapa de entre un conjunto finito de valores. Cada etapa representará un nivel en el árbol de expansión.

En primer lugar debemos fijar la descomposición en etapas que vamos a realizar y definir, dependiendo del problema, la *n*-tupla que representa la solución del problema y el significado de sus componentes x_i. Una vez que veamos las posibles opciones de cada etapa quedará definida la estructura del árbol a recorrer. Vamos a ver a través de un ejemplo cómo es posible definir la estructura del árbol de expansión.

6.2 LAS *n* REINAS

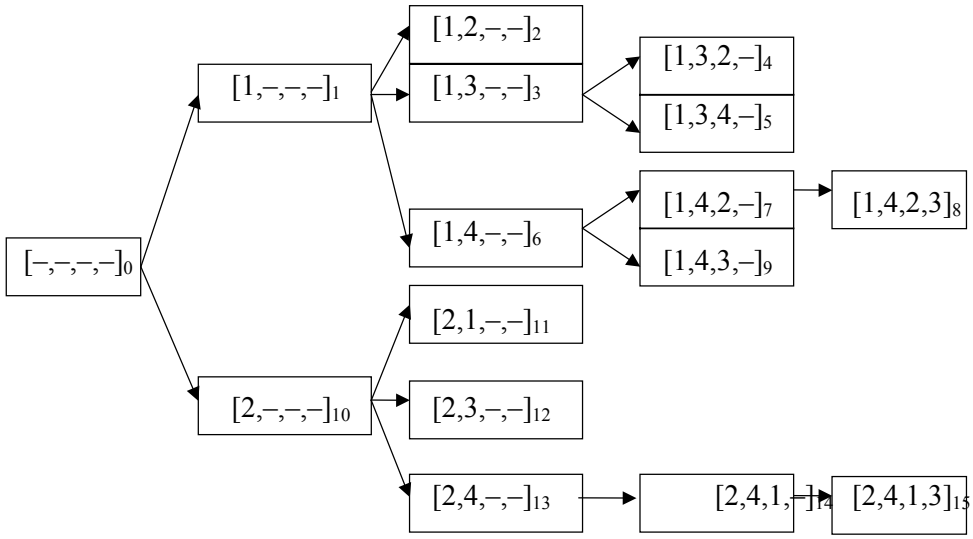
Un problema clásico que puede ser resuelto con un diseño Vuelta Atrás es el denominado de las ocho reinas y en general, de las *n* reinas. Disponemos de un tablero de ajedrez de tamaño 8x8, y se trata de colocar en él ocho reinas de manera que no se amenacen según las normas del ajedrez, es decir, que no se encuentren dos reinas ni en la misma fila, ni en la misma columna, ni en la misma diagonal.

Numeramos las reinas del 1 al 8. Cualquier solución a este problema estará representada por una 8-tupla $[x_1,x_2,x_3,x_4,x_5,x_6,x_7,x_8]$ en la que cada x_i representa la columna donde la reina de la fila *i*-ésima es colocada. Una posible solución al problema es la tupla [4,6,8,2,7,1,3,5].

Para decidir en cada etapa cuáles son los valores que puede tomar cada uno de los elementos x_i hemos de tener en cuenta lo que hemos denominado restricciones a fin de que el número de opciones en cada etapa sea el menor posible. En los algoritmos Vuelta Atrás podemos diferenciar dos tipos de restricciones:

- *Restricciones explícitas*. Formadas por reglas que restringen los valores que pueden tomar los elementos x_i a un conjunto determinado. En nuestro problema este conjunto es $S = \{1,2,3,4,5,6,7,8\}$.
- *Restricciones implícitas*. Indican la relación existente entre los posibles valores de los x_i para que éstos puedan formar parte de una n-tupla solución. En el problema que nos ocupa podemos definir dos restricciones implícitas. En primer lugar sabemos que dos reinas no pueden situarse en la misma columna y por tanto no puede haber dos x_i iguales (obsérvese además que la propia definición de la tupla impide situar a dos reinas en la misma fila, con lo cual tenemos cubiertos los dos casos, el de las filas y el de las columnas). Por otro lado sabemos que dos reinas no pueden estar en la misma diagonal, lo cual reduce el número de opciones. Esta condición se refleja en la segunda restricción implicita que, en forma de ecuación, puede ser expresada como $|x - x'| \neq |y - y'|$, siendo (x,y) y (x',y') las coordenadas de dos reinas en el tablero.

De esta manera, y aplicando las restricciones, en cada etapa k iremos generando sólo las k-tuplas con posibilidad de solución. A los prefijos de longitud k de la n-tupla solución que vamos construyendo y que verifiquen las restricciones expuestas los denominaremos *k-prometedores*, pues a priori pueden llevarnos a la solución buscada. Obsérvese que todo nodo generado es o bien fracaso o bien k-prometedor. Con estas condiciones queda definida la estructura del árbol de expansión, que representamos a continuación para un tablero 4x4:



Como podemos observar se construyen 15 nodos hasta dar con una solución al problema. El orden de generación de los nodos se indica con el subíndice que acompaña a cada tupla.

Conforme vamos construyendo el árbol debemos identificar los nodos que corresponden a posibles soluciones y cuáles por el contrario son sólo prefijos suyos. Ello será necesario para que, una vez alcanzados los nodos que sean posibles soluciones, comprobemos si de hecho lo son.

Por otra parte es posible que al alcanzar un cierto nodo del árbol sepamos que ninguna prolongación del prefijo de posible solución que representa va a ser solución a la postre (debido a las restricciones). En tal caso es absurdo que prosigamos buscando por ese camino, por lo que retrocederemos en el árbol (vuelta atrás) para seguir buscando por otra opción. Tales nodos son los que habíamos denominado nodos fracaso.

También es posible que aunque un nodo no se haya detectado a priori como fracaso (es decir, que sea *k*-prometedor) más adelante se vea que todos sus descendientes son nodos fracaso; en tal caso el proceso es el mismo que si lo hubiésemos detectado directamente. Tal es el caso para los nodos 2 y 3 de nuestro árbol. Efectivamente el nodo 2 es nodo fracaso porque al comprobar una de las restricciones (están en la misma diagonal) no se cumple. El nodo 3 sin embargo es nodo fracaso debido a que sus descendientes, los nodos 4 y 5, lo son.

Por otra parte hemos de identificar aquellos nodos que pudieran ser solución porque por ellos no se puede continuar (hemos completado la *n*-tupla), y aquellos que corresponden a soluciones parciales. No por conseguir construir un nodo hoja de nivel *n* quiere decir que hayamos encontrado una solución, puesto que para los nodos hojas también es preciso comprobar las restricciones. En nuestro árbol que representa el problema de las 4 reinas vemos cómo el nodo 8 podría ser solución ya que hemos conseguido colocar las 4 reinas en el tablero, pero sin embargo la tupla [1,4,2,3] encontrada no cumple el objetivo del problema, pues existen dos reinas $x_3 = 2$ y $x_4 = 3$ situadas en la misma diagonal. Un nodo con posibilidad de solución en el que detectamos que de hecho no lo es se comporta como nodo fracaso.

En resumen, podemos decir que Vuelta Atrás es un método exhaustivo de tanteo (prueba y error) que se caracteriza por un avance progresivo en la búsqueda de una solución mediante una serie de etapas. En dichas etapas se presentan unas opciones cuya validez ha de examinarse con objeto de seleccionar una de ellas para proseguir con el siguiente paso. Este comportamiento supone la generación de un árbol y su examen y eventual poda hasta llegar a una solución o a determinar su imposibilidad. Este avance se puede detener cuando se alcanza una solución, o bien si se llega a una situación en que ninguna de las soluciones es válida; en este caso se vuelve al paso anterior, lo que supone que deben recordarse las elecciones hechas en cada paso para poder probar otra opción aún no examinada. Este retroceso (vuelta atrás) puede continuar si no quedan opciones que examinar hasta llegar a la primera etapa. El agotamiento de todas las opciones de la primera etapa supondrá que no hay solución posible pues se habrán examinado todas las posibilidades.

El hecho de que la solución sea encontrada a través de ir añadiendo elementos a la solución parcial, y que el diseño Vuelta Atrás consista básicamente en recorrer un árbol hace que el uso de recursión sea muy apropiado. Los árboles son estructuras intrínsecamente recursivas, cuyo manejo requiere casi siempre de recursión, en especial en lo que se refiere a sus recorridos. Por tanto la

implementacion más sencilla se logra sin lugar a dudas con procedimientos recursivos.

De esta forma llegamos al esquema general que poseen los algoritmos que siguen la técnica de Vuelta Atrás:

```
PROCEDURE VueltaAtras(etapa);
BEGIN
   IniciarOpciones;
   REPEAT
      SeleccionarNuevaOpcion;
      IF Aceptable THEN
         AnotarOpcion;
         IF SolucionIncompleta THEN
            VueltaAtras(etapa_siguiente);
            IF NOT exito THEN
               CancelarAnotacion
            END
         ELSE (* solucion completa *)
            exito:=TRUE
         END
      END
   UNTIL (exito) OR (UltimaOpcion)
END VueltaAtras;
```

En este esquema podemos observar que están presentes tres elementos principales. En primer lugar hay una generación de descendientes, en donde para cada nodo generamos sus descendientes con posibilidad de solución. A este paso se le denomina expansión, ramificación o bifurcación. A continuación, y para cada uno de estos descendientes, hemos de aplicar lo que denominamos prueba de fracaso (segundo elemento). Finalmente, caso de que sea aceptable este nodo, aplicaremos la prueba de solución (tercer elemento) que comprueba si el nodo que es posible solución efectivamente lo es.

Tal vez lo más difícil de ver en este esquema es donde se realiza la vuelta atrás, y para ello hemos de pensar en la propia recursión y su mecanismo de funcionamiento, que es la que permite ir recorriendo el árbol en profundidad.

Para el ejemplo que nos ocupa, el de las *n* reinas, el algoritmo que lo soluciona quedaría por tanto como sigue:

```
CONST n = ...; (* numero de reinas; n>3 *)
TYPE SOLUCION = ARRAY[1..n] OF CARDINAL;
VAR X:SOLUCION; exito:BOOLEAN;

PROCEDURE Reinas(k: CARDINAL);
```

```
(* encuentra una manera de disponer las n reinas *)
BEGIN
   IF k>n THEN RETURN END;
   X[k]:=0;
   REPEAT
      INC(X[k]); (* seleccion de nueva opcion *)
      IF Valido(k) THEN (* prueba de fracaso *)
         IF k<>n THEN
            Reinas(k+1) (* llamada recursiva *)
         ELSE
            exito:=TRUE
         END
      END
   UNTIL (X[k]=n) OR exito;
END Reinas;
```

La función *Valido* es la que comprueba las restricciones implícitas, realizando la prueba de fracaso:

```
PROCEDURE Valido(k:CARDINAL):BOOLEAN;
(* comprueba si el vector solucion X construido hasta el paso k
   es k-prometedor, es decir, si la reina puede situarse en la
   columna k *)
   VAR i: CARDINAL;
BEGIN
   FOR i:=1 TO k-1 DO
      IF (X[i]=X[k]) OR (ValAbs(X[i],X[k])=ValAbs(i,k)) THEN
         RETURN FALSE
      END
   END;
   RETURN TRUE
END Valido;
```

Utilizamos la funcion *ValAbs*(*x*,*y*), que es la que devuelve $|x - y|$:

```
PROCEDURE ValAbs(x,y:CARDINAL):CARDINAL;
BEGIN
   IF x>y THEN RETURN x-y ELSE RETURN y-x END;
END ValAbs;
```

Cuando se desea encontrar todas las soluciones habrá que alterar ligeramente el esquema dado, de forma que una vez conseguida una solución se continúe buscando hasta agotar todas las posibilidades. Queda por tanto el siguiente esquema general para este caso:

```
PROCEDURE VueltaAtrasTodasSoluciones(etapa);
```

```
BEGIN
   IniciarOpciones;
   REPEAT
      SeleccionarNuevaOpcion;
      IF Aceptable THEN
         AnotarOpcion;
         IF SolucionIncompleta THEN
            VueltaAtrasTodasSoluciones(etapa_siguiente);
         ELSE
            ComunicarSolucion
         END;
         CancelarAnotacion
      END
   UNTIL (UltimaOpcion);
END VueltaAtrasTodasSoluciones;
```

que en nuestro ejemplo de las reinas queda reflejado en el siguiente algoritmo:

```
PROCEDURE Reinas2(k:CARDINAL);
(* encuentra todas las maneras de disponer las n reinas *)
BEGIN
   IF k>n THEN RETURN END;
   X[k]:=0; (* iniciar opciones *)
   REPEAT
      INC(X[k]); (* seleccion de nueva opcion *)
      IF Valido(k) THEN (* prueba de fracaso *)
         IF k<>n THEN
            Reinas2(k+1) (* llamada recursiva *)
         ELSE
            ComunicarSolucion(X)
         END
      END
   UNTIL (X[k]=n);
END Reinas2;
```

Aunque la solución más utilizada es la recursión, ya que cada paso es una repetición del anterior en condiciones distintas (más simples), la resolución de este método puede hacerse también utilizando la organización del árbol que determina el espacio de soluciones. Así, podemos desarrollar también un esquema general que represente el comportamiento del algoritmo de Vuelta Atrás en su versión iterativa:

```
PROCEDURE VueltaAtrasIterativo;
BEGIN
```

```
   k:=1;
   WHILE k>1 DO
      IF solucion THEN
         ComunicarSolucion
      ELSIF Fracaso(solucion) OR (k<n) THEN
         DEC(k); CalcularSucesor(k)
      ELSE
         INC(k); CalcularSucesor(k)
      END
   END
END VueltaAtrasIterativo;
```

En este esquema también vemos presentes los tres elementos anteriores: prueba de solución, prueba de fracaso y generación de descendientes.

Para cada nodo se realiza la prueba de solución en cuyo caso se terminará el proceso y la prueba de fracaso que en caso positivo da lugar a la vuelta atrás. Observamos también que si la búsqueda de descendientes no consigue ningún hijo, el nodo se convierte en nodo fracaso y se trata como en el caso anterior; en caso contrario la etapa se incrementa en uno y se continúa.

Por otra parte la vuelta atrás busca siempre un hermano del nodo que estemos analizando –descendiente de su mismo padre– para pasar a su análisis; si no existe tal hermano se decrementa la etapa *k* en curso y si *k* sigue siendo mayor que cero (aun no hemos recorrido el árbol) se repite el proceso anterior.

El algoritmo iterativo para el problema de las *n* reinas puede implementarse por tanto utilizando este esquema, lo que da lugar al siguiente procedimiento:

```
PROCEDURE Reinas_It;
   VAR k:CARDINAL;
BEGIN
   X[1]:=0; k:=1;
   WHILE k>0 DO
      X[k]:=X[k] + 1; (* selecciona nueva opcion *)
      WHILE (X[k]<=n)AND(NOT Valido(k)) DO (* fracaso? *)
         X[k]:=X[k] + 1
      END
      IF X[k]<=n THEN
         IF k=n THEN   ComunicarSolucion(X)
         ELSE INC(k); X[k]:=0
         END
      ELSE
         DEC(k) (* vuelta atras *)
      END
   END
END Reinas_It;
```

Hemos visto en este apartado cómo generar el árbol de expansión, pero sin prestar demasiada atención al orden en que lo hacemos. Usualmente los algoritmos

Vuelta Atrás son de complejidad exponencial por la forma en la que se busca la solución mediante el recorrido en profundidad del árbol. De esta forma estos algoritmos van a ser de un orden de complejidad al menos del número de nodos del árbol que se generen y este número, si no se utilizan restricciones, es de orden de z^n donde z son las posibles opciones que existen en cada etapa, y n el número de etapas que es necesario recorrer hasta construir la solución (esto es, la profundidad del árbol o la longitud de la n-tupla solución).

El uso de restricciones, tanto implícitas como explícitas, trata de reducir este número tanto como sea posible (en el ejemplo de las reinas se pasa de 8^8 nodos si no se usa ninguna restricción a poco más de 2000), pero sin embargo en muchos casos no son suficientes para conseguir algoritmos "tratables", es decir, que sus tiempos de ejecución sean de orden de complejidad razonable.

Para aquellos problemas en donde se busca una solución y no todas, es donde entra en juego la posibilidad de considerar distintas formas de ir generando los nodos del árbol. Y como la búsqueda que realiza la Vuelta Atrás es siempre en profundidad, para lograr esto sólo hemos de ir variando el orden en el que se generan los descendientes de un nodo, de manera que trate de ser lo más apropiado a nuestra estrategia.

Como ejemplo, pensemos en el problema del laberinto que veremos más adelante. En cada etapa vamos a ir generando los posibles movimientos desde la casilla en la que nos encontramos. Pero en vez de hacerlo de cualquier forma, sería interesante explorar primero aquellos que nos puedan llevar más cerca de la casilla de salida, es decir, tratar de ir siempre hacia ella.

Desde un punto de vista intuitivo, lo que intentamos hacer así es llevar lo más hacia arriba posible del árbol de expansión el nodo hoja con la solución (dibujando el árbol con la raiz a la izquierda, igual que lo hemos hecho en el problema de las reinas), para que la búsqueda en profundidad que realizan este tipo de algoritmos la encuentre antes. En algunos ejemplos, como puede ser en el del juego del Continental, que también veremos más adelante, el orden en el que se generan los movimientos hace que el tiempo de ejecución del algoritmo pase de varias horas a sólo unos segundos, lo cual no es despreciable.

6.3 RECORRIDOS DEL REY DE AJEDREZ

Dado un tablero de ajedrez de tamaño nxn, un rey es colocado en una casilla arbitraria de coordenadas (x,y). El problema consiste en determinar los n^2-1 movimientos de la figura de forma que todas las casillas del tablero sean visitadas una sola vez, si tal secuencia de movimientos existe.

Solución (☺)

La solución al problema puede expresarse como una matriz de dimensión nxn que representa el tablero de ajedrez. Cada elemento (x,y) de esta matriz solución contendrá un número natural k que indica el número de orden en que ha sido visitada la casilla de coordenadas (x,y).

El algoritmo trabaja por etapas decidiendo en cada etapa *k* hacia donde se mueve. Como existen ocho posibles movimientos en cada etapa, éste será el número máximos de hijos que se generarán por cada nodo.

Respecto a las restricciones explícitas, por la forma en la que hemos definido la estructura que representa la solución (en este caso una matriz bidimensional de números naturales), sabemos que sus componentes pueden ser números comprendidos entre cero (que indica que una casilla no ha sido visitada aún) y n^2, que es el orden del último movimiento posible. Inicialmente el tablero se encuentra relleno con ceros y sólo existe un 1 en la casilla inicial (x_0,y_0).

Las restricciones implícitas en este caso van a limitar el número de hijos que se generan desde una casilla mediante la comprobación de que el movimiento no lleve al rey fuera del tablero o sobre una casilla previamente visitada.

Una vez definida la estructura que representa la solución y las restricciones que usaremos, para implementar el algoritmo que resuelve el problema basta utilizar el esquema general, obteniendo:

```
CONST n = ...;
TYPE TABLERO = ARRAY[1..n],[1..n] OF CARDINAL;
VAR tablero:TABLERO;

PROCEDURE Rey1(k:CARDINAL;x,y:INTEGER;VAR exito:BOOLEAN);
   (* busca una solucion, si la hay. k indica la etapa, (x,y) las
      coordenadas de la casilla en donde se encuentra el rey *)
   VAR orden:CARDINAL;(* recorre cada uno de los 8 movimientos *)
       u,v:INTEGER; (* u,v indican la casilla destino desde x,y *)
BEGIN
   orden:=0;
   exito:=FALSE;
   REPEAT
      INC(orden);
      u:= x + mov_x[orden];
      v:= y + mov_y[orden];
      IF (1<=u)AND(u<=n)AND(1<=v)AND(v<=n)AND(tablero[u,v]=0) THEN
         tablero[u,v]:= k;
         IF k<n*n THEN
            Rey1(k+1,u,v,exito);
            IF NOT exito THEN tablero[u,v]:=0 END
         ELSE exito:= TRUE;
         END
      END
   UNTIL (exito) OR (orden=8);
END Rey1;
```

Las variables *mov_x* y *mov_y* contienen los movimientos legales de un rey (según las reglas de ajedrez), y son inicializadas al principio del programa principal mediante el procedimiento *MovimientosPosibles*:

```
VAR mov_x,mov_y:ARRAY[1..8] OF INTEGER;
```

```
PROCEDURE MovimientosPosibles;
BEGIN
   mov_x[1]:=0;  mov_y[1]:=1;  mov_x[2]:=-1; mov_y[2]:=1;
   mov_x[3]:=-1; mov_y[3]:=0;  mov_x[4]:=-1; mov_y[4]:=-1;
   mov_x[5]:=0;  mov_y[5]:=-1; mov_x[6]:=1; mov_y[6]:=-1;
   mov_x[7]:=1; mov_y[7]:=0;   mov_x[8]:=1; mov_y[8]:=1;
END MovimientosPosibles;
```

El programa principal es también el encargado de inicializar el tablero e invocar al procedimiento *Rey1* con los parámetros iniciales:

```
....
MovimientosPosibles();
FOR i:=1 TO n DO
   FOR j:=1 TO n DO
      tablero[i,j]:=0;
   END
END;
tablero[x0,y0]:=1; (* x0,y0 es la casilla inicial *)
Rey1(2,x0,y0,exito);
....
```

Supongamos que nos piden ahora una modificación al programa de forma que, en vez de encontrar los movimientos, calcule cuántas soluciones posee el problema, es decir, cuántos recorridos válidos distintos puede hacer el rey desde la casilla inicial dada. En este caso utilizaremos el esquema que permite encontrar todas las soluciones, lo que da lugar al siguiente programa:

```
PROCEDURE Rey2(k:CARDINAL;x,y:INTEGER;
               VAR numsoluciones:CARDINAL);
(* cuenta todas las soluciones *)
   VAR orden:CARDINAL;(*recorre cada uno de los 8 movimientos *)
       u,v:INTEGER; (* u,v indican la casilla destino desde x,y *)
BEGIN
   orden:=0;
   REPEAT
      INC(orden);
      u:= x + mov_x[orden];
      v:= y + mov_y[orden];
      IF (1<=u)AND(u<=n)AND(1<=v)AND(v<=n)AND(tablero[u,v]=0) THEN
         tablero[u,v]:= k;
         IF k<n*n THEN
            Rey2(k+1,u,v,numsoluciones)
         ELSE
```

```
            INC(numsoluciones)
          END;
          tablero[u,v]:=0
        END
    UNTIL (orden=8);
END Rey2;
```

6.4 RECORRIDOS DEL REY DE AJEDREZ (2)

Al igual que en el problema discutido anteriormente, un rey es colocado en una casilla arbitraria de coordenadas (x_0,y_0) de un tablero de ajedrez de tamaño nxn.

Si asignamos a cada casilla del tablero un peso (dado por el producto de sus coordenadas), a cada posible recorrido le podemos asignar un valor que viene dado por la suma de los pesos de las casillas visitadas por el índice del movimiento que nos llevó a esa casilla dentro del recorrido.

Esto es, si (x_0,y_0) es la casilla inicial y el recorrido R viene dado por los movimientos $[(x_1,y_1), (x_2,y_2), ..., (x_k,y_k)]$, con $k = n^2 - 1$, el peso asignado a R vendrá dado por la expresión:

$$P(R) = \sum_{i=0}^{k} i x_i y_i \ .$$

El problema consiste en averiguar el recorrido de peso mínimo para una casilla inicial dada.

Solución (☺)

Utilizaremos las mismas estructuras de datos que en el problema anterior, al igual que las restricciones. La modificación pedida es simple, pues es una pequeña variación del procedimiento *Rey2* que va recorriendo el árbol de expansión contando el número de soluciones:

```
PROCEDURE Rey3(k:CARDINAL;x,y:INTEGER);
   VAR orden,costerecorrido:CARDINAL; u,v:INTEGER;
BEGIN
   orden:=0;
   REPEAT
      INC(orden);
      u:= x + mov_x[orden]; v:= y + mov_y[orden];
      IF (1<=u)AND(u<=n)AND(1<=v)AND(v<=n)AND(tablero[u,v]=0) THEN
         tablero[u,v]:= k;
         IF k<n*n THEN
            Rey3(k+1,u,v)
         ELSE
            costerecorrido:=CalcularCoste(tablero);
            IF costerecorrido<costeminimo THEN
```

```
                costeminimo:=costerecorrido;
                mejorsolucion:=tablero
             END
          END;
          tablero[u,v]:=0
       END
    UNTIL (orden=8);
END Rey3;
```

En este ejemplo utilizamos las variables globales *costeminimo* y *mejorsolucion*:

```
VAR costeminimo:CARDINAL; mejorsolucion:TABLERO;
```

que almacenan la mejor solución encontrada hasta el momento. La primera será inicializada en el cuerpo principal del programa, y la segunda de ellas no hace falta inicializar:

```
...
costeminimo:=MAX(CARDINAL);
MovimientosPosibles();
FOR i:=1 TO n DO
   FOR j:=1 TO n DO
      tablero[i,j]:=0
   END
END;
tablero[x0,y0]:=1; (* x0,y0 es la casilla inicial *)
Rey3(2,x0,y0);
...
```

Por su parte, la función *CalcularCoste* es la que determina el peso de un recorrido dado:

```
PROCEDURE CalcularCoste(VAR t:TABLERO):CARDINAL;
   VAR i,j,coste:CARDINAL;
BEGIN
   coste:=0;
   FOR i:=1 TO n DO
      FOR j:=1 TO n DO
         coste:=coste+t[i,j]*i*j
      END
   END;
   RETURN coste;
END CalcularCoste;
```

Existe una pequeña variación a este algoritmo que consiste en ir acarreando el coste del recorrido en curso a lo largo del recorrido del árbol, lo que supondría ahorrarse la llamada (de orden $O(n^2)$) a la función *CalcularCoste* cada vez que se

alcance una hoja del árbol de expansión (esto es, una posible solución). Para implementarlo, basta con incluir como un parámetro más del procedimiento *Rey3* una variable que lleve acumulado el peso del recorrido hasta el momento. Cada vez que se escoja una nueva opción se incrementará tal valor, y cada vez que se cancele una anotación se decrementará en las unidades correspondientes.

Por otro lado, también es posible plantearse si existe un algoritmo más eficiente que el de Vuelta Atrás para resolver este problema. Observando la forma en la que se define la función de peso, el algoritmo debe tratar siempre de visitar las casillas de mayores coordenadas primero, para que el número de orden en que son visitadas, que es un factor multiplicativo en la función, sea lo menor posible.

Así, nos encontramos delante de un típico algoritmo ávido, que escogería siempre como siguiente casilla (x,y) a mover de entre las posibles a aquella aún no visitada y cuyo producto de coordenadas xy sea máximo. La complejidad de este algoritmo es de orden $O(n^2)$, mucho más eficiente que el de Vuelta Atrás. Sin embargo, la demostración de que siempre encuentra la solución no es sencilla.

6.5 LAS PAREJAS ESTABLES

Supongamos que tenemos n hombres y n mujeres y dos matrices M y H que contienen las preferencias de los unos por los otros. Más concretamente, la fila $M[i,\cdot]$ es una ordenación (de mayor a menor) de las mujeres según las preferencias del i-ésimo hombre y, análogamente, la fila $H[i,\cdot]$ es una ordenación (de mayor a menor) de los hombres según las preferencias de la i-ésima mujer.

El problema consiste en diseñar un algoritmo que encuentre, si es que existe, un emparejamiento de hombres y mujeres tal que todas las parejas formadas sean *estables*. Diremos que una pareja (h,m) es estable si no se da ninguna de estas dos circunstancias:

1) Existe una mujer m' (que forma la pareja (h',m')) tal que el hombre h la prefiere sobre la mujer m y además la mujer m' también prefiere a h sobre h'.
2) Existe un hombre h'' (que forma la pareja (h'',m'')) tal que la mujer m lo prefiere sobre el hombre h y además el hombre h'' también prefiere a m sobre la mujer m''.

Solución (😐)

Para este problema vamos a disponer de una n-tupla X que vamos a ir rellenando en cada etapa del algoritmo, y que contiene las mujeres asignadas a cada uno de los hombres. En otras palabras, x_i indicará el número de la mujer asignada al i-ésimo hombre en el emparejamiento. El algoritmo que resuelve el problema trabajará por etapas y en cada etapa k decide la mujer que ha de emparejarse con el hombre k.

Analicemos en primer lugar las restricciones del problema. En una etapa cualquiera k, el k-ésimo hombre escogerá la mujer que prefiere en primer lugar, siempre y cuando esta mujer aún esté libre y la pareja resulte estable. Para saber las mujeres aún libres utilizaremos un vector auxiliar denominado *libre*. Por simetría, aparte de la n-tupla X, también dispondremos de otra n-tupla Y que contiene los hombres asignados a cada mujer, que necesitaremos al comprobar las restricciones.

Por último, también son necesarias dos tablas auxiliares, *ordenM* y *ordenH*. La primera almacena en la posición [*i*,*j*] el orden de preferencia de la mujer *i* por el hombre *j*, y la segunda almacena en la posición [*i*,*j*] el orden de preferencia del hombre *i* por la mujer *j*.

Con todo esto, el procedimiento que resuelve el problema puede ser implementado como sigue:

```
TYPE PREFERENCIAS = ARRAY [1..n],[1,,n] OF CARDINAL;
     ORDEN = ARRAY [1..n],[1,,n] OF CARDINAL;
     SOLUCION = ARRAY [1..n] OF CARDINAL;
     DISPONIBILIDAD = ARRAY [1..n] OF BOOLEAN;

VAR  M,H:PREFERENCIAS;
     ordenM,ordenH:ORDEN;
     X,Y:SOLUCION;
     libre:DISPONIBILIDAD;

PROCEDURE Parejas(hombre:CARDINAL;VAR exito:BOOLEAN);
   VAR mujer,prefiere,preferencias:CARDINAL;
BEGIN
   prefiere:=0; (* recorre las posibles elecciones del hombre *)
   REPEAT
      INC(prefiere);
      mujer:=M[hombre,prefiere];
      IF libre[mujer] AND Estable(hombre,mujer,prefiere) THEN
         X[hombre]:=mujer;
         Y[mujer]:=hombre;
         libre[mujer]:=FALSE;
         IF hombre<n THEN
            Parejas(hombre+1,exito);
            IF NOT exito THEN
               libre[mujer]:=TRUE
            END
         ELSE
            exito:=TRUE;
         END
      END
   UNTIL (prefiere=n) OR exito;
END Parejas;
```

La función *Estable* queda definida como:

```
PROCEDURE Estable(h,m,p:CARDINAL):BOOLEAN;
```

```
   VAR mejormujer,mejorhombre,i,limite:CARDINAL;s:BOOLEAN;
BEGIN
   s:=TRUE; i:=1;
   WHILE (i<p) AND s DO (* es estable respecto al hombre? *)
      mejormujer:=M[h,i];
      INC(i);
      IF NOT(libre[mejormujer])THEN
         s:=ordenM[mejormujer,h]>ordenM[mejormujer,Y[mejormujer]];
      END
   END;
   i:=1; limite:=H[m,h]; (* es estable respecto a la mujer? *)
   WHILE(i<limite) AND s DO
      mejorhombre:=H[m,i];
      INC(i);
      IF mejorhombre<h THEN
      s:=ordenH[mejorhombre,m]>ordenH[mejorhombre,X[mejorhombre]];
      END
   END;
   RETURN s
END Estable;
```

El problema se resuelve mediante la inicialización apropiada de las matrices de preferencias y una invocación a *Parejas*(1,*exito*). Tras su ejecución, las variables *X* e *Y* contendrán la asignaciones respectivas siempre que la variable *exito* lo indique.

6.6 EL LABERINTO

Una matriz bidimensional $n \times n$ puede representar un laberinto cuadrado. Cada posición contiene un entero no negativo que indica si la casilla es transitable (0) o no lo es (∞). Las casillas [1,1] y [*n*,*n*] corresponden a la entrada y salida del laberinto y siempre serán transitables.

Dada una matriz con un laberinto, el problema consiste en diseñar un algoritmo que encuentre un camino, si existe, para ir de la entrada a la salida.

Solución (😐)

En este problema iremos avanzando por el laberinto en cada etapa, y cada nodo representará el camino recorrido hasta el momento. Por la forma en la que trabaja el esquema general de Vuelta Atrás podemos utilizar una variable global (una matriz) para representar el laberinto e ir apuntando los movimientos que realizamos, indicando en cada casilla el orden en el que ésta ha sido visitada. Al producirse la vuelta atrás nos cuidaremos de liberar las casillas ocupadas por el nodo del que volvemos (marcándolas de nuevo con 0).

Con esto en mente, el algoritmo es sencillo:

```
CONST n = ...;
```

```
TYPE LABERINTO = ARRAY[1..n],[1..n] OF CARDINAL;
VAR lab:LABERINTO;

PROCEDURE Laberinto(k:CARDINAL;VAR fil,col:INTEGER;
                    VAR exito:BOOLEAN);
   VAR orden:CARDINAL; (*indica hacia donde debe moverse *)
BEGIN
   orden:=0; exito:=FALSE;
   REPEAT
      INC(orden);
      fil:=fil + mov_fil[orden];
      col:=col + mov_col[orden];
      IF (1<=fil) AND (fil<=n) AND (1<=col) AND (col<=n) AND
      (lab[fil,col]=0) THEN
         lab[fil,col]:=k;
         IF (fil=n) AND (col=n) THEN exito:=TRUE
         ELSE
            Laberinto(k+1,fil,col,exito);
            IF NOT exito THEN lab[fil,col]:=0 END
         END
      END;
      fil:=fil - mov_fil[orden];
      col:=col - mov_col[orden]
      END
   UNTIL (exito) OR (orden=4)
END Laberinto;
```

Las variables *mov_fil* y *mov_col* contienen los posibles movimientos, y son inicializadas por el procedimiento *MovimientosPosibles* que mostramos a continuación:

```
VAR mov_fil,mov_col:ARRAY [1..4] OF INTEGER;

PROCEDURE MovimientosPosibles;
BEGIN
   mov_fil[1]:=1;  mov_col[1]:=0;  (* sur *)
   mov_fil[2]:=0;  mov_col[2]:=1;  (* este *)
   mov_fil[3]:=0;  mov_col[3]:=-1; (* oeste *)
   mov_fil[4]:=-1; mov_col[4]:=0;  (* norte *)
END MovimientosPosibles;
```

Tal como mencionamos en la introducción de este capítulo, el orden en que se intentan esos movimientos es importante, pues podemos utilizar la información disponible de que la casilla de salida es la [*n*,*n*] para tratar siempre de ir hacia ella. Ésta es la razón por la que se intenta primero el sur, luego el este, y por último el oeste y el norte. Para cambiar la forma en la que se realizan los intentos, y por tanto

el orden en el que se construyen los nodos del árbol de expansión, basta modificar el procedimiento *MovimientosPosibles*, que es invocado una vez al comienzo del programa principal –que también es el que invoca la primera vez a la función *Laberinto*–. Por supuesto, independientemente de esta ordenación, el algoritmo encuentra la salida si es que ésta es alcanzable.

Mayor importancia tendría este orden de expandir los nodos si lo que se desease fuera encontrar no una solución cualquiera, sino la más corta. Para ello el algoritmo que lo realiza está basado en el esquema general que busca todas las soluciones y se queda con la mejor:

```
PROCEDURE Laberinto2(k:CARDINAL;VAR fil,col:INTEGER);
   VAR orden:CARDINAL; (*indica hacia donde debe moverse *)
BEGIN
   orden:=0;
   REPEAT
      INC(orden);
      fil:=fil + mov_fil[orden];
      col:=col + mov_col[orden];
      IF (1<=fil) AND (fil<=n) AND (1<=col) AND (col<=n) AND
         (lab[fil,col]=0) THEN
         lab[fil,col]:=k;
         IF (fil=n) AND (col=n) THEN
            (* almacenamos el mejor recorrido hasta el momento *)
            recorridominimo:=k; solucion:=lab;
         ELSIF k<=recorridominimo THEN (* poda! *)
            Laberinto2(k+1,fil,col);
         END;
         lab[fil,col]:=0;
      END;
      fil:=fil - mov_fil[orden];
      col:=col - mov_col[orden]
   UNTIL (orden=4)
END Laberinto2;
```

En este caso hemos introducido una variante muy importante: el uso de una cota para podar ramas del árbol de expansión. Si bien ésta es una técnica que se utiliza sobre todo en los algoritmos de Ramificación y Poda (y de ahí su nombre), el uso de cotas para podar puede ser aplicado a cualquier tipo de árboles de expansión.

En el problema que nos ocupa calculamos la primera solución y para ella se obtiene un valor. En este caso es el número de movimientos que ha realizado el algoritmo hasta encontrar la salida, que es el valor que queremos minimizar. Pues bien, disponiendo ya de un valor alcanzable podemos "rechazar" todos aquellos nodos cuyos recorridos superen este valor, sean soluciones parciales o totales, pues no nos van a llevar hacia la solución óptima. Estas *podas* ahorran mucho trabajo al algoritmo, pues evitan que éste realice trabajo innecesario.

La variable *recorridominimo*, que es la que hace las funciones de cota, se inicializa a *MAX(CARDINAL)* al principio del programa principal que invoca al procedimiento *Laberinto2*(1,1,1).

Como norma general para los algoritmos de Vuelta Atrás, y puesto que su complejidad es normalmente exponencial, debemos de saber aprovechar toda la información disponible sobre el problema o sus soluciones en forma de restricciones, pues son éstas la clave de su eficiencia. En la mayoría de los casos la diferencia entre un algoritmo Vuelta Atrás útil y otro que, por su tardanza, no pueda utilizarse se encuentra en las restricciones impuestas a los nodos, único parámetro disponible al programador de estos métodos para mejorar la eficiencia de los algoritmos resultantes.

6.7 LA ASIGNACIÓN DE TAREAS

Este problema fue introducido en el capítulo cuatro (apartado 4.11), y básicamente puede ser planteado como sigue. Dadas n personas y n tareas, queremos asignar a cada persona una tarea. El coste de asignar a la persona i la tarea j viene determinado por la posición $[i,j]$ de una matriz dada (TARIFAS). Diseñar un algoritmo que asigne una tarea a cada persona minimizando el coste de la asignación.

Solución (😐)

En primer lugar hemos de decidir cómo representaremos la solución del problema mediante una n-tupla de valores $X = [x_1, x_2, ..., x_n]$. En este ejemplo el valor x_i va a representar la tarea asignada a la i-ésima persona. Empezando por la primera persona, en cada etapa el algoritmo irá avanzando en la construcción de la solución, comprobando siempre que el nuevo valor añadido a ella es compatible con los valores anteriores. Por cada solución que encuentre anotará su coste y lo comparará con el coste de la mejor solución encontrada hasta el momento, que almacenará en la variable global *mejor*. El algoritmo puede ser implementado como sigue:

```
CONST n = ...; (* numero de personas y trabajos *)
TYPE  TARIFAS = ARRAY[1..n],[1..n] OF CARDINAL;
      SOLUCION = ARRAY[1..n] OF CARDINAL;

VAR   X,mejor:SOLUCION;
      minimo:CARDINAL;
      tarifa:TARIFAS;

PROCEDURE Asignacion(k:CARDINAL);
   VAR c:CARDINAL;
BEGIN
   X[k]:=0;
   REPEAT
      INC(X[k]);
```

```
      IF Aceptable(k) THEN
         IF k<n THEN Asignacion(k+1)
         ELSE
            c:=Coste();
            IF minimo>c THEN mejor:=X; minimo:=c END;
         END
      END
   UNTIL X[k]=n
END Asignacion;
```

Siendo los procedimientos *Aceptable* y *Coste* los que respectivamente comprueban las restricciones para este problema y calculan el coste de las soluciones que van generándose. En cuanto a las restricciones, sólo se va a definir una, que asegura que las tareas sólo se asignan una vez. Por otro lado, el coste de una solución coincide con el coste de la asignación:

```
PROCEDURE Aceptable(k:CARDINAL):BOOLEAN;
(* comprueba que esa tarea no ha sido asignada antes *)
   VAR i:CARDINAL;
BEGIN
   FOR i:=1 TO k-1 DO
      IF X[k]= X[i] THEN RETURN FALSE END
   END;
   RETURN TRUE
END Aceptable;


PROCEDURE Coste():CARDINAL;
   VAR suma,i:CARDINAL;
BEGIN
   suma:=0;
   FOR i:=1 TO n DO
      suma:=suma+tarifa[i,X[i]]
   END;
   RETURN suma
END Coste;
```

Para resolver el problema basta con invocar al procedimiento *Asignación* tras obtener los valores de la tabla de tarifas e inicializar la variable *minimo*:

```
...
minimo:=MAX(CARDINAL);
Asignacion(1);
ComunicarSolucion(mejor);
...
```

Al igual que en el problema anterior, existe una modificación a este algoritmo que permite realizar podas al árbol de expansión eliminando aquellos nodos que no van a llevar a la solución óptima. Para implementar esta modificación –siempre interesante puesto que consigue reducir el tamaño del árbol de búsqueda– necesitamos hacer uso de una cota que almacene el valor obtenido por la mejor solución hasta el momento, y además llevar la cuenta en cada nodo del coste acumulado hasta ese nodo. Si el coste acumulado es mayor que el valor de la cota, no merece la pena continuar explorando los hijos de ese nodo, pues nunca nos llevarán a una solución mejor de la que teníamos.

Como la cota la tenemos disponible ya (es la variable *minimo* del algoritmo anterior), lo que haremos es ir calculando de forma acumulada en vez de al llegar a una solución, y así podremos realizar la poda cuanto antes:

```
PROCEDURE Asignacion2(k:CARDINAL;costeacum:CARDINAL);
   VAR coste:CARDINAL;
BEGIN
   X[k]:=0;
   REPEAT
      INC(X[k]);
      coste:=costeacum+tarifa[k,X[k]];
      IF Aceptable(k) AND (coste<=minimo) THEN
         IF k<n THEN Asignacion2(k+1,coste)
         ELSE mejor:=X; minimo:=coste
         END
      END
   UNTIL X[k]=n
END Asignacion2;
```

También hacer una última observación sobre el problema de la asignación en general. Este problema siempre posee solución puesto que siempre existen asignaciones válidas. De esta forma no nos tenemos que preocupar de si el algoritmo acaba con éxito o no.

6.8 LA MOCHILA (0,1)

El problema de la mochila (0,1) –originalmente descrito en el apartado 4.8– ha sido discutido en los dos últimos capítulos, y hemos visto que no posee solución mediante un algoritmo ávido, aunque sí la tiene utilizado Programación Dinámica. Nos planteamos aquí dar una solución al problema utilizando Vuelta Atrás.

Recordemos el enunciado del problema. Dados n elementos e_1, e_2, ..., e_n con pesos p_1, p_2, ..., p_n y beneficios b_1, b_2, ..., b_n, y dada una mochila capaz de albergar hasta un máximo peso M (capacidad de la mochila), queremos encontrar cuáles de los n elementos hemos de introducir en la mochila de forma que la suma de los beneficios de los elementos escogidos sea máxima, sujeto a la restricción de que tales elementos no pueden superar la capacidad de la mochila.

Solución (😐/✍)

Éste es uno de los problemas cuya resolución más sencilla se obtiene utilizando la técnica de Vuelta Atrás, puesto que basta expresar la solución al problema como una n-tupla de valores $X = [x_1, x_2, ..., x_n]$ donde x_i tomará los valores 1 ó 0 dependiendo de que el i-ésimo objeto sea introducido o no. El árbol de expansión resultante es, por tanto, trivial.

Sin embargo, y puesto que se trata de un problema de optimización, podemos aplicar una poda para eliminar aquellos nodos que no conduzcan a una solución óptima. Para ello vamos utilizaremos una función (*Cota*) que describiremos más adelante.

Como su versión recursiva no plantea mayores dificultades por tratarse de un mero ejercicio de aplicación del esquema general, este problema lo resolveremos mediante un algoritmo iterativo:

```
CONST n = ...; (* numero de elementos *)
      M = ...; (* capacidad de la mochila *)

TYPE REGISTRO = RECORD peso,beneficio:REAL END;
      ELEMENTOS = ARRAY[1..n] OF REGISTRO;
      MOCHILA   = ARRAY[1..n] OF CARDINAL;

PROCEDURE Mochila(elem:ELEMENTOS;capacidad:REAL;VAR X:MOCHILA;
                        VAR peso_final,beneficio_final:REAL);
   VAR peso_en_curso,beneficio_en_curso:REAL;
       sol:MOCHILA; (* solucion en curso *)
       k:CARDINAL;
BEGIN
   peso_en_curso:=0.0;
   beneficio_en_curso:=0.0;
   beneficio_final:=-1.0;
   k:=1;
   LOOP
      WHILE  (k<=n) AND (peso_en_curso+elem[k].peso<=capacidad) DO
         peso_en_curso:=peso_en_curso+elem[k].peso;
         beneficio_en_curso:=beneficio_en_curso+elem[k].beneficio;
         sol[k]:=1;
         INC(k)
      END;
      IF k>n THEN
         beneficio_final:=beneficio_en_curso;
         peso_final:=peso_en_curso;
         k:=n;
         X:=sol;
      ELSE
         sol[k]:=0
      END;
```

```
      WHILE Cota(elem,beneficio_en_curso,peso_en_curso,k,capacidad)<=
                beneficio_final DO
         WHILE (k<>0) AND (sol[k]<>1) DO DEC(k) END;
         IF k=0 THEN  EXIT  END;
         sol[k]:=0;
         peso_en_curso:=peso_en_curso-elem[k].peso;
         beneficio_en_curso:=beneficio_en_curso-elem[k].beneficio
      END;
      INC(k)
   END
END Mochila;
```

La función *Cota* es la que va a permitir realizar la poda del árbol de expansión para aquellos nodos que no lleven a la solución óptima. Para ello vamos a considerar que los elementos iniciales están todos ordenados de forma decreciente por su ratio beneficio/peso. De esta forma, supongamos que nos encontramos en el paso *k*-ésimo, y que disponemos de un beneficio acumulado B_k. Por la manera en como hemos ido construyendo el vector, sabemos que

$$B_k = \sum_{i=1}^{k} sol[i] * elem[i].beneficio\ .$$

Para calcular el valor máximo que podríamos alcanzar con ese nodo (B_M), vamos a suponer que rellenáramos el resto de la mochila con el mejor de los elementos que nos quedan por analizar. Como los tenemos dispuestos en orden decreciente de ratio beneficio/peso, este mejor elemento será el siguiente (k+1). Este valor, aunque no tiene por qué ser alcanzable, nos permite dar una cota superior del valor al que podemos "aspirar" si seguimos por esa rama del árbol:

$$B_M = B_k + \left(capacidad - \sum_{i=1}^{k} sol[i] * elem[i].peso \right) \frac{elem[k+1].beneficio}{elem[k+1].peso}$$

Con esto:

```
PROCEDURE Cota(e:ELEMENTOS;b:REAL;p:REAL;k:CARDINAL;cap:CARDINAL):REAL;
   VAR i: CARDINAL; ben_ac,peso_ac:REAL; (* acumulados *)
BEGIN
   ben_ac:=b; peso_ac:=p;
   FOR i:=k+1 TO n DO
      peso_ac:=peso_ac+e[i].peso;
      IF peso_ac<cap THEN ben_ac:=ben_ac+e[i].beneficio
      ELSE
         RETURN (ben_ac+(1.0-(peso_ac-cap)/e[i].peso)*(e[i].beneficio))
      END
   END;
   RETURN (beneficio_acumulado)
END Cota;
```

El tipo de poda que hemos realizado al árbol de expansión de este ejercicio está más cerca de las técnicas de poda que se utilizan en los algoritmos de Ramificación y Poda que de las típicas restricciones definidas para los algoritmos Vuelta Atrás.

Aparte de las diferencias existentes entre ambos tipos de algoritmos en cuanto a la forma de recorrer el árbol (mucho más flexible en la técnica de Ramificación y Poda) y la utilización de estructuras globales en Vuelta Atrás (frente a los nodos "autónomos" de la Ramificación y Poda), no existe una frontera clara entre los procesos de poda de unos y otros. En cualquier caso, repetimos la importancia de la poda, esencial para convertir en tratables estos algoritmos de orden exponencial.

6.9 LOS SUBCONJUNTOS DE SUMA DADA

Sea W un conjunto de enteros no negativos y M un número entero positivo. El problema consiste en diseñar un algoritmo para encontrar todos los posibles subconjuntos de W cuya suma sea exactamente M.

Solución (☺)

En primer lugar podemos suponer sin pérdida de generalidad (ni de eficiencia, porque a la postre el algoritmo resultante es de complejidad exponencial) que el conjunto viene dado por un vector de enteros no negativos y que ya se encuentra ordenado de forma creciente. Con esto en mente, la solución al problema podremos expresarla como una n-tupla $X = [x_1, x_2, ..., x_n]$ en donde x_i podrá tomar los valores 1 ó 0 indicando que el entero i forma parte de la solución o no. El algoritmo trabaja por etapas y en cada etapa ha de decidir si el k-ésimo entero del conjunto interviene o no en la solución.

A modo de ejemplo, supongamos el conjunto $W = \{2,3,5,10,20\}$ y sea M=15. Existen dos posibles soluciones, dadas por las 5-tuplas [1,1,0,1,0] y [0,0,1,1,0].

Definamos en primer lugar las restricciones del problema. Llamaremos v_i al valor del i-ésimo elemento. En una etapa k cualquiera podemos considerar que una condición para que pueda encontrarse solución es que se cumpla que:

$$\sum_{i=1}^{k} v_i x_i + \sum_{i=k+1}^{n} v_i \geq M$$

es decir, en una etapa k cualquiera, la suma de todos los elementos que se han considerado hasta esa etapa más el valor de todos los que faltan por considerar al menos ha de ser igual al valor M dado, porque si no, si ni siquiera introduciendo todos se llega a alcanzar este valor, significa que por este camino no hay solución. Es más, como sabemos que los elementos están en orden no decreciente podemos afirmar que:

$$\sum_{i=1}^{k} v_i x_i + v_{k+1} \leq M \ ,$$

es decir, que si al valor conseguido hasta la etapa k le añadimos el siguiente elemento (que es el menor) y ya se supera el valor de M, esto significa que no será posible alcanzar la solución por este camino.

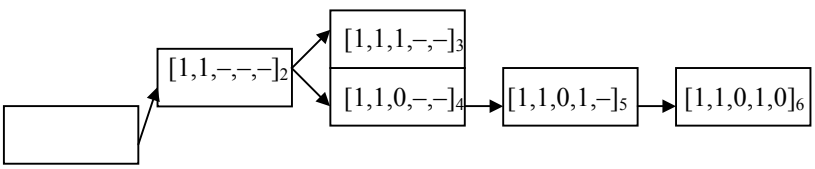
Estas dos restricciones nos van a permitir reducir la búsqueda al evitar caminos que no conducen a solución. En el algoritmo, llamaremos:

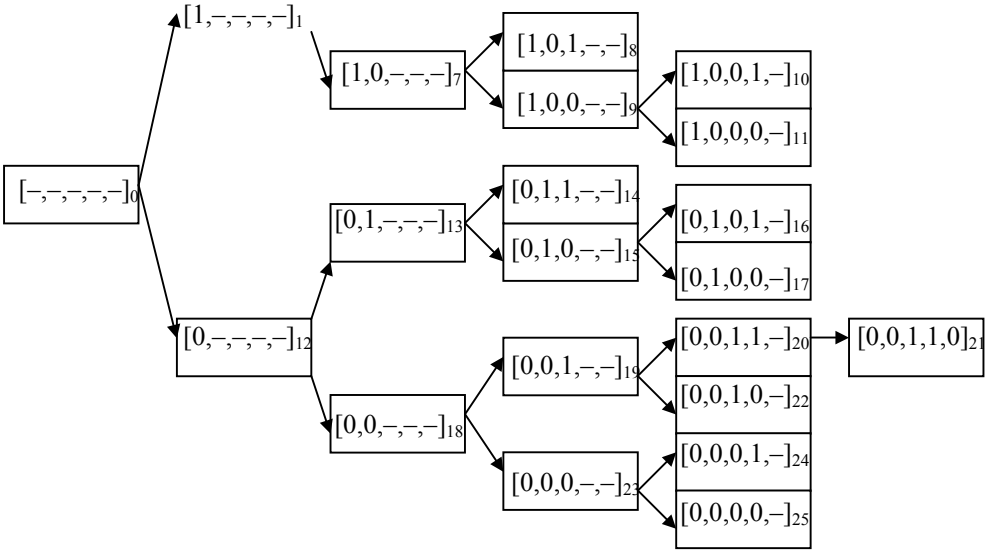
$$s = \sum_{j=1}^{k-1} v_j x_j \quad \text{y} \quad r = \sum_{j=k}^{n} v_j \, .$$

Con esto, el procedimiento que encuentra todos los posibles subconjuntos es el siguiente:

```
CONST n = ...; (* numero de elementos *)
      M = ...; (* cantidad dada *)

TYPE CONJUNTO = ARRAY [1..n] OF CARDINAL;
     SOLUCION = ARRAY [1..n] OF CARDINAL;

VAR  v:CONJUNTO;
     X:SOLUCION;

PROCEDURE Subconjuntos(s,k,r:CARDINAL);
BEGIN
   X[k]:=1;
   IF s+v[k]=M THEN
      ComunicarSolucion(k)
   ELSIF k=n THEN
      RETURN
   ELSIF s+v[k]+v[k+1]<=M THEN
      Subconjuntos(s+v[k],k+1,r-v[k])
   END;
   IF(s+r-v[k]>=M) AND(s+v[k+1]<=M) THEN
      X[k]:=0;
      Subconjuntos(s,k+1,r-v[k])
   END
END Subconjuntos;
```

Observemos cómo se va construyendo el árbol de expansión. En cada etapa k hemos de decidir, caso de que no se haya alcanzado la solución, si es posible añadir este elemento k al subconjunto y continuar por el hijo izquierdo, o no considerarlo y continuar por el derecho. Repitiendo el mismo proceso en cada etapa hasta o bien alcanzar la solución y en tal caso comunicarla o bien determinar la imposibilidad de continuar por esa rama. En ambos casos es necesario la vuelta atrás retrocediendo al nivel anterior. Para el ejemplo inicial del conjunto $W = \{2,3,5,10,20\}$ y $M = 15$, el árbol que va construyendo el algoritmo es:





cuyas soluciones son [1,1,0,1,0] y [0,0,1,1,0].

Para finalizar, el algoritmo ha de invocarse inicialmente desde el programa principal como *Subconjuntos*(*s*,1,*r*), donde *s* = 0 y *r* es igual a la suma de todos los elementos del conjunto.

6.10 CICLOS HAMILTONIANOS. EL VIAJANTE DE COMERCIO

Dado un grafo conexo, se llama *Ciclo Hamiltoniano* a aquel ciclo que visita exactamente una vez cada vértice del grafo y vuelve al punto de partida. El problema consiste en detectar la presencia de ciclos Hamiltonianos en un grafo dado.

Solución

Suponiendo como hemos hecho hasta ahora que los vértices del grafo están numerados desde 1 hasta *n*, la solución al problema puede expresarse como una *n*-tupla de valores $X = [x_1, x_2, ..., x_n]$, donde x_i representa el *i*-ésimo vértice del ciclo Hamiltoniano. El algoritmo que resuelve el problema trabajará por etapas, decidiendo en cada etapa qué vértice del grafo de los aún no considerados puede formar parte del ciclo. Así, el algoritmo que resuelve el problema puede ser implementado como sigue:

```
CONST n = ...; (* numero de vertices *)
TYPE SOLUCION = ARRAY[1..n] OF CARDINAL;
     GRAFO  = ARRAY[1..n],[1..n] OF BOOLEAN;
VAR g:GRAFO; X:SOLUCION; existe:BOOLEAN;
PROCEDURE Hamiltoniano1(k:CARDINAL; VAR existe:BOOLEAN);
(* comprueba si existe un ciclo Hamiltoniano *)
BEGIN
```

```
   LOOP
      NuevoVertice(k);
      IF X[k] = 0 THEN EXIT END;
      IF k = n THEN existe:=TRUE
      ELSE Hamiltoniano1(k+1,existe)
      END
   END
END Hamiltoniano1;
```

Siendo el procedimiento *NuevoVertice* el que busca el siguiente vértice libre que pueda formar parte del ciclo y lo almacena en el vector solución *X*:

```
PROCEDURE NuevoVertice(k: CARDINAL);
   VAR j:CARDINAL; s:BOOLEAN;
BEGIN
   LOOP
      X[k]:=(X[k]+1) MOD (n+1);
      IF X[k]=0 THEN RETURN END;
      IF g[X[k-1],X[k]] THEN
         j:=1; s:=TRUE;
         WHILE (j<=k-1) AND s DO
            s:=(X[j]<>X[k]); INC(j)
         END;
         IF (j=k)AND((k<n)OR((k=n)AND(g[X[n],1]))) THEN RETURN END
      END
   END
END NuevoVertice;
```

El algoritmo termina cuando encuentra un ciclo o bien cuando ha analizado todas las posibilidades y no encuentra solución. Y respecto al estado de las variables y los parámetros de su invocación inicial, debe realizarse como sigue:

```
...
existe:=FALSE;
X[1]:=1;
Hamiltoniano1(2,existe);
IF existe THEN ComunicarSolucion(X) ELSE ...
...
```

Nos podemos plantear también una modificación al algoritmo para que encuentre todos los ciclos Hamiltonianos si es que hubiera más de uno. La modificación en este caso es simple:

```
PROCEDURE Hamiltoniano2(k:CARDINAL);
(* determina todos los ciclos *)
BEGIN
```

```
   LOOP
      NuevoVertice(k);
      IF X[k]=0 THEN RETURN END
      IF k=n THEN ComunicarSolucion(X)
      ELSE Hamiltoniano2(k+1)
      END
   END
END Hamiltoniano2;
```

Asimismo sería interesante generalizar el procedimiento anterior para tratar con grafos ponderados. El problema consistiría en diseñar un algoritmo que, dado un grafo ponderado con pesos positivos, encuentre el ciclo Hamiltoniano de coste mínimo.

En este caso hay que encontrar todos los ciclos Hamiltonianos, siendo necesario calcular el coste de cada solución y actualizar para obtener la óptima (el ciclo con mínimo coste). Lo interesante es que este problema coincide con nuestro viejo conocido el problema del viajante de comercio, ampliamente discutido en capítulos anteriores, y cuya solución mediante Vuelta Atrás es la que sigue:

```
PROCEDURE Hamiltoniano3(k:CARDINAL);
(* calcula el ciclo Hamiltoniano de minimo coste *)
BEGIN
   LOOP
      NuevoVertice2(k);
      IF X[k]=0 THEN RETURN END;
      IF k=n THEN
         coste:=CalcularCoste(X);
         IF coste<minimo THEN
            minimo:=coste;
            XMIN:=X
         END
      ELSE Hamiltoniano3(k+1)
      END
   END
END Hamiltoniano3;
```

La función *CalcularCoste* es la que obtiene la suma de los elementos de la *n*-tupla solución construida. Por otro lado, también se hace uso de dos variables globales para almacenar el coste mínimo y el camino:

```
VAR minimo:CARDINAL; XMIN:SOLUCION;
```

Por su parte, el procedimiento *NuevoVertice2* trabajará en este caso con un grafo ponderado, y por tanto los elementos de su matriz de adyacencia g serán enteros no negativos:

```
PROCEDURE NuevoVertice2(k:CARDINAL);
```

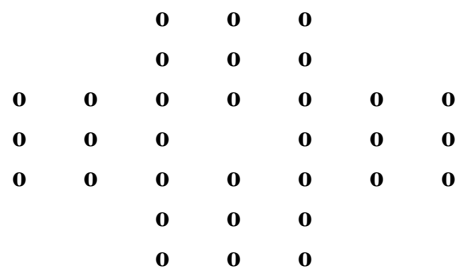
```
   VAR j:CARDINAL; s,vuelta:BOOLEAN;
BEGIN
   LOOP
      X[k]:=(X[k]+1) MOD (n+1);
      IF X[k]=0 THEN RETURN END;
      IF g[X[k-1],X[k]]<>MAX(CARDINAL) THEN
         j:=1; s:=TRUE;
         WHILE (j<=k-1) AND s DO
            s:=(X[j]<>X[k]); INC(j)
         END;
         vuelta:=g[X[n],1]<>MAX(CARDINAL);
         IF (j=k) AND ((k<n)OR((k=n)AND vuelta)) THEN RETURN END
      END
   END
END NuevoVertice2.
```

Este algoritmo debe ser invocado desde el programa principal tras inicializar la variable *minimo* y el primer elemento del vector *X* con el vértice desde donde parte el ciclo.

```
...
minimo:=MAX(CARDINAL);
X[1]:=1;
Hamiltoniano3(2);
IF minimo<MAX(CARDINAL) THEN ComunicarSolucion(XMIN)
...
```

6.11 EL CONTINENTAL

En el solitario de mesa llamado *Continental*, treinta y dos piezas se colocan en un tablero de treinta y tres casillas tal y como indica la siguiente figura, quedando vacía únicamente la casilla central:



Una pieza sólo puede moverse saltando sobre una de sus vecinas y cayendo en una posición vacía, al igual que en el juego de las *damas*, aunque aquí no están permitidos los saltos en diagonal. La pieza sobre la que se salta se retira del tablero.

El problema consiste en diseñar un algoritmo que encuentre una serie de movimientos (saltos) que, partiendo de la configuración inicial expuesta en la figura, llegue a una situación en donde sólo quede una pieza en el tablero, que ha de estar en la posición central.

Solución (😐)

Representaremos el tablero como una matriz bidimensional en la que los elementos pueden tomar los valores *novalida*, *libre* u *ocupada*, dependiendo de que esa posición no sea válida (no correspoda a una de las posición que forman la "cruz"), o bien siendo válida exista ficha o no.

La solución vendrá dada en una tupla de valores $X = [x_1, x_2, ..., x_m]$ en donde x_i representa un movimiento (salto) del juego. En este valor se almacenará la posición de la ficha que va a efectuar el movimiento, la posición hacia donde da el salto y la posición de la ficha "comida". El valor m representa el número de saltos (movimientos) efectuados para alcanzar la solución. Puesto que en cada movimiento ha de comerse obligatoriamente una ficha, sabemos que m podrá tomar a lo sumo el valor 31. Con esto, el algoritmo que resuelve el problema es:

```
CONST m = 31;  (* numero maximo de movimientos *)
      n = 7;   (* tamano del tablero *)
TYPE ESTADO   = (libre,ocupada,novalida);(* tipo de casilla *)
     TABLERO  = ARRAY[1..n],[1..n] OF ESTADO;
     PAR      = RECORD x,y:INTEGER END; (* coordenadas *)
     SALTO    = RECORD origen,destino,comido:PAR END;
     SOLUCION = ARRAY [1..m] OF SALTO;

PROCEDURE Continental(VAR k:CARDINAL;VAR t:TABLERO;
                      VAR encontrado:BOOLEAN;VAR sol:SOLUCION);
   VAR i,j:CARDINAL;
BEGIN
   IF Fin(k,t) THEN encontrado:=TRUE
   ELSE
      FOR i:=1 TO n DO
         FOR j:=1 TO n DO
            IF Valido(i,j,1,t,encontrado) THEN(* a la izquierda *)
               INC(k);
               sol[k].origen.x:=i;
               sol[k].origen.y:=j;
               sol[k].destino.x:=i;
               sol[k].destino.y:=j-2;
               sol[k].comido.x:=i;
               sol[k].comido.y:=j-1;
               NuevaTabla(t,i,j,1); (* actualiza el tablero *)
               Continental(k,t,encontrado,sol)
            END;
            IF Valido(i,j,2,t,encontrado) THEN (* hacia arriba *)
```

```
            INC(k);
            sol[k].origen.x:=i;
            sol[k].origen.y:=j;
            sol[k].destino.x:=i-2;
            sol[k].destino.y:=j;
            sol[k].comido.x:=i-1;
            sol[k].comido.y:=j;
            NuevaTabla(t,i,j,2);(* actualiza el tablero *)
            Continental(k,t,encontrado,sol)
         END;
         IF Valido(i,j,3,t,encontrado) THEN (* a la derecha *)
            INC(k);
            sol[k].origen.x:=i;
            sol[k].origen.y:=j;
            sol[k].destino.x:=i;
            sol[k].destino.y:=j+2;
            sol[k].comido.x:=i;
            sol[k].comido.y:=j+1;
            NuevaTabla(t,i,j,3);(* actualiza el tablero *)
            Continental(k,t,encontrado,sol)
         END;
         IF Valido(i,j,4,t,encontrado) THEN (* hacia abajo *)
            INC(k);
            sol[k].origen.x:=i;
            sol[k].origen.y:=j;
            sol[k].destino.x:=i+2;
            sol[k].destino.y:=j;
            sol[k].comido.x:=i+1;
            sol[k].comido.y:=j;
            NuevaTabla(t,i,j,4);(* actualiza el tablero *)
            Continental(k,t,encontrado,sol)
         END;
      END
   END;
   IF NOT encontrado THEN (* cancelar anotacion *)
      RestaurarTabla(t,k,sol);
      AnularSalida(sol,k);
      DEC(k)
   END
  END
END Continental;
```

La función *Fin* determina si se ha llegado al final del juego, esto es, si sólo queda una ficha y ésta se encuentra en el centro del tablero, y la función *Valido* comprueba si una ficha puede moverse o no:

```
PROCEDURE Valido(i,j,mov:CARDINAL;VAR t:TABLERO;e:BOOLEAN):BOOLEAN;
BEGIN
   IF mov=1 THEN (* izquierda *)
      RETURN ((j-1>0) AND (t[i,j]=ocupada) AND(t[i,j-1]=ocupada)
                AND (j-2>0) AND (t[i,j-2]=libre) AND (NOT e))
   ELSIF mov=2 THEN (* arriba *)
      RETURN ((i-1>0) AND (t[i-1,j]=ocupada) AND(t[i,j]=ocupada)
                AND (i-2>0) AND (t[i-2,j]=libre) AND (NOT e))
   ELSIF mov=3 THEN (* derecha *)
      RETURN ((j+1<8) AND (t[i,j+1]=ocupada) AND(t[i,j]=ocupada)
                AND (j+2<8) AND (t[i,j+2]=libre) AND (NOT e))
   ELSIF mov=4 THEN (* abajo *)
      RETURN ((i+1<8) AND (t[i+1,j]=ocupada) AND(t[i,j]=ocupada)
                AND (i+2<8) AND (t[i+2,j]=libre) AND (NOT e))
   END
END Valido;
```

Un aspecto interesante de este problema es que pone de manifiesto la importancia que tiene el orden de generación de los nodos del árbol de expansión. Si en vez de seguir la secuencia de búsqueda utilizada en la anterior implementación (izquierda, arriba, derecha y abajo) se intentan los movimientos en otro orden, el tiempo de ejecución del algoritmo pasa de varios segundos a más de dos horas.

6.12 HORARIOS DE TRENES

El problema de los horarios de los trenes fue enunciado en el capítulo anterior (apartado 5.15). Una compañía de ferrocarriles que sirve n estaciones $S_1,...,S_n$ trata de mejorar su servicio al cliente mediante terminales de información. Dadas una estación origen S_o y una estación destino S_d, un terminal debe ofrecer (inmediatamente) la información sobre el horario de los trenes que hacen la conexión entre S_o y S_d y que minimizan el tiempo de trayecto total.

Necesitamos por tanto implementar un algoritmo que realice esta tarea a partir de la tabla con los horarios, suponiendo que las horas de salida de los trenes coinciden con las de sus llegadas (es decir, que no hay tiempos de espera) y que, naturalmente, no todas las estaciones están conectadas entre sí por líneas directas; así, en muchos casos hay que hacer transbordos aunque se supone que tardan tiempo cero en efectuarse. Nos planteamos en este apartado solucionarlo mediante un algoritmo de Vuelta Atrás.

Solución (😐)

En primer lugar es necesario expresar la solución del problema mediante una n-tupla de valores $X = [x_1, x_2, ..., x_n]$. En este caso, cada x_i va a representar el

número de estación que compone el trayecto más corto. La *n*-tupla estará rellena hasta la posición *k*-ésima, siendo *k* el número de estaciones que debe recorrer, dándose además que $x_1 = S_o$ y $x_k = S_d$. Por comodidad y sin ninguna pérdida de generalidad supondremos que las estaciones están numeradas del 1 al *n*.

En cada etapa iremos probando estaciones, con la restricción de que no pasemos dos veces por la misma y que la que añadamos nueva en cada paso esté conectada con la anterior.

Como además se trata de un problema de optimización utilizaremos una restricción adicional, como es la comprobación de que el tiempo acumulado hasta el momento por una solución en proceso no supere el tiempo alcanzado por una solución ya conocida. Estas ideas dan lugar al siguiente algoritmo:

```
CONST n = ...; (* numero de estaciones *)

TYPE SOLUCION = ARRAY [1..n] OF CARDINAL;
     HORARIOS = ARRAY [1..n],[1..n] OF CARDINAL;

VAR estacionorigen,estaciondestino,minimo:CARDINAL;
    tablatiempos:HORARIOS;
    X,solucionoptima:SOLUCION;

PROCEDURE Estaciones(k:CARDINAL;tiempoacum:CARDINAL);
BEGIN
   X[k]:=0;
   REPEAT
      INC(X[k]);
      IF Aceptable(k) THEN
         tiempoacum:=tiempoacum+tablatiempos[X[k-1],X[k]];
         IF tiempoacum<=minimo THEN
            IF X[k]=estaciondestino THEN (* hemos llegado *)
               solucionoptima:=X; minimo:=tiempoacum
            ELSIF k<n THEN
               Estaciones(k+1,tiempoacum)
            END
         END
      END
   UNTIL X[k]=n
END Estaciones;
```

Las variables *estacionorigen* y *estaciondestino* son las que el usuario elige, y la matriz *tablatiempos* determina el tiempo de conexión entre cada par de estaciones, pudiendo ser $tablatiempos[i,j] = \infty$ si no existe conexión entre las estaciones *i* y *j*. Por su parte, la función *Aceptable* comprueba las restricciones definidas anteriormente para el problema:

```
PROCEDURE Aceptable(k:CARDINAL):BOOLEAN;
   VAR i:CARDINAL;
```

```
BEGIN
   FOR i:=1 TO k-1 DO (* no puede haber estaciones repetidas *)
      IF X[i]=X[k] THEN RETURN FALSE END
   END;
   (* la nueva estacion ha de ser alcanzable desde la anterior *)
   RETURN tablatiempos[X[k-1],X[k]]<MAX(CARDINAL);
END Aceptable;
```

Obsérvese cómo en este caso la solución puede estar compuesta por menos de n valores significativos. De esta forma, de la n-tupla solución X construida sólo hemos de considerar los k primeros elementos, siendo k el primer índice para el que $X[k] = estaciondestino$.

Por otro lado, el programa principal, además de pedir al usuario las estaciones origen y destino y de dar valores a la tabla de tiempos entre estaciones, debe inicializar la variable *minimo* y el vector con la estacion origen, e invocar al procedimiento que realiza el proceso de Vuelta Atrás:

```
...
minimo:=MAX(CARDINAL);
X[1]:=estacionorigen;
Estaciones(2,0);
IF minimo<MAX(CARDINAL) THEN ComunicarSolucion(solucion)
ELSE ...
```

Al final, la variable global *solucion* contendrá el recorrido óptimo y la variable *minimo* el tiempo total de ese trayecto, a menos que el problema no tenga solución (como ocurre por ejemplo en el caso de que la estación destino no sea alcanzable desde la estación origen) en cuyo caso la variable *minimo* seguirá valiendo ∞.

6.13 LA ASIGNACIÓN DE TAREAS EN PARALELO

Supongamos que necesitamos realizar n tareas en una máquina multiprocesador, pero que sólo m procesadores pueden trabajar en paralelo. Sea t_i el tiempo de ejecución de la i-ésima tarea ($1 \le i \le n$).

El problema consiste en implementar un algoritmo que determine en qué procesador y en qué orden hay que ejecutar los trabajos, de forma que el tiempo total de ejecución sea mínimo.

Solución (☺)

Para resolver este problema expresaremos su solución mediante una n-tupla de valores $X = [x_1, x_2, ..., x_n]$ donde cada valor x_i representa el procesador que realiza la i-ésima tarea. Minimizar el tiempo total de ejecución significará encontrar la solución que tenga menor el máximo de los tiempos de cada procesador.

Las restricciones explícitas establecen el rango de valores que pueden tomar los componentes x_i de la n-tupla solución, que en este caso han de ser números comprendidos entre 1 y m.

Por otro lado no definiremos ninguna restricción implícita puesto que a priori cualquier *n*-tupla formada por valores que cumplan las restricciones explícitas es suceptible de ser solución. Con todo esto, el algoritmo que resuelve el problema puede ser planteado como sigue:

```
CONST m = ...; (* numero de procesadores *)
      n = ...; (* numero de tareas *)
TYPE TIEMPOS  = ARRAY [1..n] OF CARDINAL;
     SOLUCION = ARRAY [1..n] OF CARDINAL;
     VALOR    = ARRAY [1..m] OF CARDINAL;
VAR tiempos:TIEMPOS; (* tiempo de ejecucion de cada tarea *)
    mejor: CARDINAL;
    X,solucion:SOLUCION;
    valor:VALOR; (* tiempo empleado por cada procesador *)

PROCEDURE Procesador(k:CARDINAL);
   VAR t,j:CARDINAL;
BEGIN
   FOR j:=1 TO m DO (* probamos con todos los procesadores *)
      X[k]:=j; valor[j]:=valor[j]+tiempos[k];
      IF k<n THEN Procesador(k+1);
      ELSE
         t:=CalcularTiempo();
         IF mejor>t THEN mejor:=t; solucion:=X END
      END;
      valor[j]:=valor[j]-tiempos[k];
   END
END Procesador;
```

La función *CalcularTiempo* es la que calcula el tiempo total máximo para una tupla solución:

```
PROCEDURE CalcularTiempo():CARDINAL;
   VAR i,maximo: CARDINAL;
BEGIN
   maximo:=valor[1];
   FOR i:=2 TO m DO
      IF valor[i]>maximo THEN maximo:=valor[i] END
   END;
   RETURN maximo
END CalcularTiempo;
```

El programa principal ha de inicializar la variable *mejor* y ha de invocar al procedimiento *Procesador*:

```
...
mejor:=MAX(CARDINAL);
```

```
Procesador(1);
ComunicarSolucion(solucion);
...
```

6.14 EL COLOREADO DE MAPAS

Dado un grafo conexo y un número $m > 0$, llamamos *colorear* el grafo a asignar un número i ($1 \leq i \leq m$) a cada vértice, de forma que dos vértices adyacentes nunca tengan asignados números iguales. Deseamos implementar un algoritmo que coloree un grafo dado.

Solución (☺)

El nombre de este problema proviene de un problema clásico, el del coloreado de mapas en el plano. Para resolverlo se utilizan grafos puesto que un mapa puede ser representado por un grafo conexo. Cada vértice corresponde a un país y cada arco entre dos vértices indica que los dos países son vecinos. Desde el siglo XVII ya se conoce que con cuatro colores basta para colorear cualquier mapa planar, pero sin embargo existen situaciones en donde no nos importa el número de colores que se utilicen.

Para implementar un algoritmo de Vuelta Atrás, la solución al problema puede expresarse como una n-tupla de valores $X = [x_1, x_2, ..., x_n]$ donde x_i representa el color del i-ésimo vértice. El algoritmo que resuelve el problema trabajará por etapas, asignando en cada etapa k un color (entre 1 y m) al vértice k-ésimo.

En primer lugar, y para un grafo con n vértices y con m colores, el algoritmo que encuentra una solución al problema es el siguiente:

```
CONST n = ...; (* numero de vertices *)
      m = ...; (* numero maximo de colores  *)

TYPE  GRAFO=ARRAY[1..n],[1..n]OF BOOLEAN; (* matriz adyacencia *)
      SOLUCION = ARRAY [1..n] OF CARDINAL;

VAR   g:GRAFO; X:SOLUCION; exito:BOOLEAN

PROCEDURE Colorear1(k:CARDINAL); (* busca una posible solucion *)
BEGIN
   X[k]:=0;
   REPEAT
      INC(X[k]);
      IF Aceptable(k) THEN
         IF k<n THEN Colorear1(k+1)
         ELSE exito:=TRUE
         END
      END
   UNTIL (exito) OR (X[k]=m)
```

```
END Colorear1;
```

La función *Aceptable* es la que comprueba las restricción definida para este problema, que consiste en que dos países vecinos (vértices adyacentes) no pueden tener el mismo color:

```
PROCEDURE Aceptable(k: CARDINAL): BOOLEAN;
   VAR j: CARDINAL;
BEGIN
   FOR j:=1 TO k-1 DO
      IF (g[k,j]) AND (X[k]=X[j]) THEN RETURN FALSE END
   END;
   RETURN TRUE
END Aceptable;
```

El programa principal del algoritmo ha de invocar a la función *Colorear1* como sigue:

```
...
exito:=FALSE;
Colorear1(1);
IF exito THEN ComunicarSolucion(X)
...
```

Supongamos ahora que lo que deseamos es obtener todas las formas distintas de colorear un grafo. Entonces el algoritmo anterior podría ser modificado de la siguiente manera:

```
PROCEDURE Colorear2(k:CARDINAL);
(* busca todas las soluciones *)
BEGIN
   X[k]:=0;
   REPEAT
      INC(X[k]);
      IF Aceptable(k) THEN
         IF k<n THEN Colorear2(k+1)
         ELSE ComunicarSolucion(X)
         END
      END
   UNTIL (X[k]=m)
END Colorear2;
```

Por último, vamos a mostrar el algoritmo para colorear un grafo con el mínimo número de colores:

```
PROCEDURE Colorear3(k:CARDINAL);
(* busca la solucion optima *)
   VAR numcolores:CARDINAL;
```

```
BEGIN
   X[k]:=0;
   REPEAT
      INC(X[k]);
      IF Aceptable(k) THEN
         IF k<n THEN Colorear3(k+1)
         ELSE
            numcolores:=NumeroColores(X);
            IF minimo>numcolores THEN
               mejor:=X;
               minimo:=numcolores
            END
         END
      END
   UNTIL (X[k]=m)
END Colorear3;
```

La función *NumeroColores* es la que calcula el número de colores utilizado en una solución. Por la forma en la que hemos ido construyendo las soluciones no queda garantizado que los colores utilizados posean números consecutivos, de forma que es necesario comprobar todos los colores para saber cuales han sido usados en una solución concreta:

```
PROCEDURE NumeroColores(X:SOLUCION):CARDINAL;
   VAR i,j,suma:CARDINAL; sigo:BOOLEAN;
BEGIN
   suma:=0;
   FOR j:=1 TO m DO (* recorremos todos los colores *)
      i:=1;
      sigo:=FALSE;
      WHILE (i<n) AND sigo DO
         IF X[i]=j THEN (* encontrado el color j *)
            INC(suma);
            sigo:=FALSE
         END
      END
   END;
   RETURN suma;
END NumeroColores;
```

En estos algoritmos es importante hacer notar que la constante *m* que indica el número máximo de colores a utilizar ha de ser mayor o igual a cuatro, pues se sabe que con cuatro colores basta siempre que el grafo corresponda a un mapa. Ahora bien, conviene también observar que no todo grafo conexo representa a un mapa planar; por ejemplo un grafo de cinco vértices completamente conexo, es decir, que tenga todos sus vértices conectados entre sí, no puede corresponder a un mapa en el plano.

6.15 RECONOCIMIENTO DE GRAFOS

Dadas dos matrices de adyacencia, el problema consiste en determinar si ambas representan al mismo grafo, salvo nombres de los vértices.

Solución (🖋)

Nuestro punto de partida son dos matrices cuadradas L_1 y L_2 que representan las matrices de adyacencia de dos grafos g_1 y g_2. Queremos ver si g_1 y g_2 son iguales, salvo por la numeración en sus vértices.

Podemos suponer sin pérdida de generalidad que la dimensión de ambas matrices coincide. Si no, al estar suponiendo que los vértices de los grafos están numerados de forma consecutiva a partir de 1, ya podríamos decidir que ambos grafos son distintos por tener diferente número de vértices. Sea entonces n la dimensión de ambas matrices.

Supondremos además, por ser el caso más general, que los grafos son ponderados y dirigidos, y por ello definimos el tipo de las matrices de adyacencia como sigue:

```
CONST n = ...; (* numero de vertices *)
TYPE MATRIZ = ARRAY [1..n],[1..n] OF CARDINAL;
```

Para resolver este problema lo que necesitamos es realizar una aplicación, si es posible, entre los vértices del primer grafo y los del segundo. Por tanto, podemos representar la solución como una n-tupla de valores $X = [x_1, x_2, ..., x_n]$, donde x_i va a indicar el vértice de g_2 que corresponde al i-ésimo vértice de g_1.

En cada etapa el algoritmo, empezando por el vértice 1 de g_1, va a ir construyendo la solución, tratando de asignar un vértice de g_2 a cada uno de g_1.

Una vez tenemos expresada de esta forma la solución podemos definir las restricciones que aplicaremos al problema, puesto que si no el árbol de expansión constaría de n^n nodos, demasiados para que el algoritmo sea operativo.

En primer lugar, no se pueden repetir vértices, es decir, los elementos de la n-tupla X han de ser todos distintos (una permutación de los n vértices de g_2). Además, el vértice de g_2 indicado por x_k ha de tener el mismo número de arcos (entrantes y salientes de él) que el correspondiente vértice k de g_1.

Por otro lado, para que al añadir el elemento k-ésimo a una solución parcial sea k-prometedora, las conexiones (arcos) entre el nuevo elemento x_k y los ya asignados en la solución parcial X ha de coincidir con las existentes en los correspondientes vértices de g_1. Esto hace que el algoritmo que resuelve el problema pueda ser implementado como sigue:

```
PROCEDURE GrafosIguales(k:CARDINAL);
   VAR vertice:CARDINAL; (*vertice que indica la opcion en curso*)
BEGIN
   vertice:=0;
```

```
   REPEAT
      INC(vertice);
      X[k]:=vertice;
      IF Valido(k) THEN
         IF k<n THEN
            GrafosIguales(k+1)
         ELSE
            exito:=TRUE
         END
      END
   UNTIL (X[k]=n) OR exito;
END GrafosIguales;
```

La "inteligencia" del algoritmo la suministra la función *Valido*, que es la que comprueba las restricciones anteriormente citadas:

```
PROCEDURE Valido(k:CARDINAL):BOOLEAN;
   VAR i:CARDINAL;
BEGIN
   FOR i:=1 TO k-1 DO (* no pueden repetirse elementos *)
      IF X[i]=X[k] THEN
         RETURN FALSE
      END
   END;
   IF NumArcos(L1,k)<>NumArcos(L2,X[k]) THEN(* mismo num. arcos *)
      RETURN FALSE
   END;
   FOR i:=1 TO k-1 DO (* mismas conexiones *)
      IF (L2[X[i],X[k]]<>L1[i,k]) OR (L2[X[k],X[i]]<>L1[k,i]) THEN
         RETURN FALSE
      END
   END;
   RETURN TRUE;
END Valido;
```

La función *NumArcos* es la que, dada una matriz de adyacencia de un grafo dirigido y ponderado como los que estamos considerando, y uno de sus vértices, devuelve el número de arcos que salen y entran de él:

```
PROCEDURE NumArcos(VAR L:MATRIZ;k:CARDINAL):CARDINAL;
   VAR i,suma:CARDINAL;
BEGIN
   suma:=0;
   FOR i:=1 TO n DO
      IF ((i<>k) AND (L[i,k]<MAX(CARDINAL))) THEN
         INC(suma)
```

```
      END;
      IF ((i<>k) AND (L[k,i]<MAX(CARDINAL))) THEN
         INC(suma)
      END;
   END;
   RETURN suma;
END NumArcos;
```

6.16 SUBCONJUNTOS DE IGUAL SUMA

Dado un conjunto de n enteros, necesitamos decidir si puede ser descompuesto en dos subconjuntos disjuntos cuyos elementos sumen la misma cantidad.

Solución (☺)

Para resolver el problema almacenaremos el conjunto en un vector de enteros y representaremos la solución mediante una n-tupla de valores $X = [x_1, x_2, ..., x_n]$ donde cada componente x_i puede tomar los valores 1 ó 2 indicando que el i-ésimo elemento pertenece al subconjunto 1 o al subconjunto 2 respectivamente.

En cuanto a las restricciones implícitas, la primera condición que exigiremos al conjunto de enteros para que pueda ser descompuesto en dos subconjuntos que sumen igual es que la suma de los elementos del conjunto sea un número par. Esto lo comprobaremos en el programa principal, y antes de invocar por primera vez al procedimiento recursivo que realiza el algoritmo Vuelta Atrás.

En cada etapa k intentaremos colocar el k-ésimo elemento del conjunto en uno de los dos subconjuntos posibles, lo que da lugar al siguiente procedimiento:

```
CONST n = ...; (* numero de elementos del conjunto *);

TYPE CONJUNTO = ARRAY[1..n] OF INTEGER;
     SOLUCION = ARRAY[1..n] OF CARDINAL;

VAR X:SOLUCION; numeros:CONJUNTO;
    suma:ARRAY[1..2] OF INTEGER; (* suma acumulada de
                                    los subconjuntos *)




PROCEDURE DosSubconjuntos(k:CARDINAL);
   VAR j:CARDINAL;
BEGIN
   FOR j:=1 TO 2 DO (* cada una de las dos posibilidades *)
      X[k]:=j;
      suma[j]:=suma[j]+numeros[k];
      IF k<n THEN
```

```
         DosSubconjuntos(k+1)
      ELSIF suma[1]=suma[2] THEN
         ComunicarSolucion;
      END;
      suma[j]:=suma[j]-numeros[k] (* cancelar anotacion *)
   END
END DosSubconjuntos;
```

En este problema podemos también definir una restricción en forma de cota que permita realizar la poda de aquellos nodos del árbol de expansión que sepamos que no conducen a una solución. La idea consiste en sumar al principio todos los elementos del conjunto, que en cualquier caso hemos de hacer para ver si es un número par. Con esta suma, que almacenaremos en la variable global *sumatotal*, podemos dejar de explorar aquellos nodos del árbol de expansión que verifiquen que la suma de uno de los dos subconjuntos que están construyendo sea mayor que la mitad de la suma total:

```
PROCEDURE DosSubconjuntos2(k:CARDINAL);
   VAR j:CARDINAL;
BEGIN
   FOR j:=1 TO 2 DO
      X[k]:=j;
      suma[j]:=suma[j]+numeros[k];
      IF suma[j]<=(sumatotal DIV 2) THEN (* poda *)
         IF k<n THEN DosSubconjuntos2(k+1)
         ELSIF suma[1]=suma[2] THEN ComunicarSolucion;
         END
      END;
      suma[j]:=suma[j]-numeros[k] (* cancelar anotacion *)
   END
END DosSubconjuntos2;
```

De esta forma conseguimos incrementar las restricciones del problema, lo que contribuye a una menor expansión del número de nodos y por tanto a una mayor eficiencia del algoritmo resultante.

6.17 LAS MÚLTIPLES MOCHILAS (0,1)

Dados n elementos, cada uno con un beneficio b_i y un peso p_i asociado ($1 \le i \le n$), y m mochilas, cada una con una capacidad k_j ($1 \le j \le m$), el problema de las Múltiples Mochilas (0,1) puede describirse como la asignación de los elementos a las mochilas de forma que se maximice el beneficio total de todos los elementos asignados sin superar la capacidad de las mochilas, y teniendo en cuenta que cada elemento puede ser asignado a una mochila o a ninguna, y que un elemento aportará beneficio sólo si éste es introducido en una mochila.

Nos planteamos diseñar un algoritmo Vuelta Atrás para la resolución del problema de las Múltiples Mochilas (0,1) para cualquier conjunto de elementos y mochilas.

Solución (☺)

La solución al problema puede expresarse como una n-tupla $X = [x_1, x_2, ..., x_n]$ en donde x_i representa la mochila en donde es introducido el i-ésimo elemento, o bien x_i puede valer cero si el elemento no se introduce en ninguna mochila. En cada etapa k el algoritmo irá construyendo la tupla solución, intentando decidir en qué mochila introduce el k-ésimo elemento, o si lo deja fuera.

En este caso las restricciones consisten en comprobar que no se supera la capacidad de la mochila indicada. Esto da lugar al siguiente programa:

```
CONST n = ...; (* numero de elementos *)
      m = ...; (* numero de mochilas *)
TYPE MOCHILAS = ARRAY[1..m] OF INTEGER;
     SOLUCION = ARRAY[1..n] OF INTEGER
     PAR = RECORD peso,beneficio:INTEGER END;
     ELEMENTOS = ARRAY[1..n] OF PAR;
VAR cap:ARRAY[1..m]OF CARDINAL;(*capacidad libre de las mochilas *)
     ben,benoptimo:CARDINAL;
     X,soloptima:SOLUCION;
     elem:ELEMENTOS;

PROCEDURE MochilaMultiple(k:CARDINAL);
   VAR j:CARDINAL;
BEGIN
   FOR j:=0 TO m DO
      IF (j=0) OR (cap[j]>=elem[k].peso) THEN (* restricciones *)
         X[k]:=j;
         IF j>0 THEN (* hacer anotacion *)
            cap[j]:=cap[j]-elem[k].peso;
            ben:=ben+elem[k].beneficio
         END;
         IF k<n THEN MochilaMultiple(k+1)
         ELSIF ben>benoptimo THEN
            benoptimo:=ben; soloptima:=X (* actualizar solucion *)
         END;
         IF j>0 THEN (* cancelar anotacion *)
            cap[j]:=cap[j]+elem[k].peso;
            ben:=ben-elem[k].beneficio
         END
      END
   END
END MochilaMultiple;
```

Como puede observarse, este ejemplo es una muestra clara de las ventajas que ofrece el disponer de un esquema general de diseño de algoritmos Vuelta Atrás, pues permite resolver los problemas de forma sencilla, unificada y general.

Capítulo 7

RAMIFICACIÓN Y PODA

7.1 INTRODUCCIÓN

Este método de diseño de algoritmos es en realidad una variante del diseño Vuelta Atrás estudiado en el capítulo anterior. Sin embargo, su particular importancia y extenso uso hace que nosotros le dediquemos un capítulo aparte.

Esta técnica de diseño, cuyo nombre en castellano proviene del término inglés *Branch and Bound*, se aplica normalmente para resolver problemas de optimización. Ramificación y Poda, al igual que el diseño Vuelta Atrás, realiza una enumeración parcial del espacio de soluciones basándose en la generación de un árbol de expansión.

Una característica que le hace diferente al diseño anterior es la posibilidad de generar nodos siguiendo distintas estrategias. Recordemos que el diseño Vuelta Atrás realiza la generación de descendientes de una manera sistemática y de la misma forma para todos los problemas, haciendo un recorrido en profundidad del árbol que representa el espacio de soluciones. El diseño Ramificación y Poda en su versión más sencilla puede seguir un recorrido en anchura (estrategia LIFO) o en profundidad (estrategia FIFO), o utilizando el cálculo de funciones de coste para seleccionar el nodo que en principio parece más prometedor (estrategia de mínimo coste o LC).

Además de estas estrategias, la técnica de Ramificación y Poda utiliza cotas para podar aquellas ramas del árbol que no conducen a la solución óptima. Para ello calcula en cada nodo una cota del posible valor de aquellas soluciones alcanzables desde ése. Si la cota muestra que cualquiera de estas soluciones tiene que ser necesariamente peor que la mejor solución hallada hasta el momento no necesitamos seguir explorando por esa rama del árbol, lo que permite realizar el proceso de poda.

Definimos *nodo vivo* del árbol a un nodo con posibilidades de ser ramificado, es decir, que no ha sido podado. Para determinar en cada momento que nodo va a ser expandido y dependiendo de la estrategia de búsqueda seleccionada, necesitaremos almacenar todos los nodos vivos en alguna estructura que podamos recorrer. Emplearemos una pila para almacenar los nodos que se han generado pero no han sido examinados en una búsqueda en profundidad (LIFO). Las búsquedas en amplitud utilizan una cola (FIFO) para almacenar los nodos vivos de tal manera que van explorando nodos en el mismo orden en que son creados. La estrategia de mínimo coste (LC) utiliza una *función de coste* para decidir en cada momento qué nodo debe explorarse, con la esperanza de alcanzar lo más rápidamente posible una

solución más económica que la mejor encontrada hasta el momento. Utilizaremos en este caso una estructura de montículo (o cola de prioridades) para almacenar los nodos ordenados por su coste.

Básicamente, en un algoritmo de Ramificación y Poda se realizan tres etapas. La primera de ellas, denominada de *Selección*, se encarga de extraer un nodo de entre el conjunto de los nodos vivos. La forma de escogerlo va a depender directamente de la estrategia de búsqueda que decidamos para el algoritmo. En la segunda etapa, la *Ramificación*, se construyen los posibles nodos hijos del nodo seleccionado en el paso anterior. Por último se realiza la tercera etapa, la *Poda*, en la que se eliminan algunos de los nodos creados en la etapa anterior. Esto contribuye a disminuir en lo posible el espacio de búsqueda y así atenuar la complejidad de estos algoritmos basados en la exploración de un árbol de posibilidades. Aquellos nodos no podados pasan a formar parte del conjunto de nodos vivos, y se comienza de nuevo por el proceso de selección. El algoritmo finaliza cuando encuentra la solución, o bien cuando se agota el conjunto de nodos vivos.

Para cada nodo del árbol dispondremos de una función de coste que nos estime el valor óptimo de la solución si continuáramos por ese camino. De esta manera, si la cota que se obtiene para un nod, que por su propia construcción deberá ser mejor que la solución real (o a lo sumo, igual que ella), es peor que una solución ya obtenida por otra rama, podemos podar esa rama pues no es interesante seguir por ella. Evidentemente no podremos realizar ninguna poda hasta que hayamos encontrado alguna solución. Por supuesto, las funciones de coste han de ser crecientes respecto a la profundidad del árbol, es decir, si h es una función de coste entonces $h(n) \leq h(n')$ para todo n' nodo descendiente de n.

En consecuencia, y a la vista de todo esto, podemos afirmar que lo que le da valor a esta técnica es la posibilidad de disponer de distintas estrategias de exploración del árbol y de acotar la búsqueda de la solución, que en definitiva se traduce en eficiencia. La dificultad está en encontrar una buena función de coste para el problema, buena en el sentido de que garantice la poda y que su cálculo no sea muy costoso. Si es demasiado simple probablemente pocas ramas puedan ser excluidas. Dependiendo de cómo ajustemos la función de coste mejor algoritmo se deriva.

Inicialmente, y antes de proceder a la poda de nodos, tendremos que disponer del coste de la mejor solución encontrada hasta el momento que permite excluir de futuras expansiones cualquier solución parcial con un coste mayor. Como muchas veces no se desea esperar a encontrar la primera solución para empezar a podar, un buen recurso para los problemas de optimización es tomar como mejor solución inicial la obtenida con un algoritmo ávido, que como vimos no encuentra siempre la solución óptima, pero sí una cercana a la óptima.

Por último, sólo comentar una ventaja adicional que poseen estos algoritmos: la posibilidad de ejecutarlos en paralelo. Debido a que disponen de un conjunto de nodos vivos sobre el que se efectúan las tres etapas del algoritmo antes mencionadas, nada impide tener más de un proceso trabajando sobre este conjunto, extrayendo nodos, expandiéndolos y realizando la poda. El disponer de algoritmos paralelizables (y estos algoritmos, así como los de Divide y Vencerás lo son) es muy importante en muchas aplicaciones en las que es necesario abordar los problemas de forma paralela para resolverlos en tiempos razonables, debido a su complejidad intrínseca.

Sin embargo, todo tiene un precio, sus requerimientos de memoria son mayores que los de los algoritmos Vuelta Atrás. Ya no se puede disponer de una estructura global en donde ir construyendo la solución, puesto que el proceso de construcción deja de ser tan "ordenado" como antes. Ahora se necesita que cada nodo sea autónomo, en el sentido que ha de contener toda la información necesaria para realizar los procesos de bifurcación y poda, y para reconstruir la solución encontrada hasta ese momento.

7.2 CONSIDERACIONES DE IMPLEMENTACIÓN

Uno de las dificultades que suele plantear la técnica de Ramificación y Poda es la implementación de los algoritmos que se obtienen. Para subsanar este problema, en esta sección presentamos una estructura general de tales algoritmos, basada en tres módulos (en el sentido de Modula-2) principales:

1. De un lado dispondremos del módulo que contiene el esquema de funcionamiento general de este tipo de algoritmos.
2. Por otro se encuentra el módulo que maneja la estructura de datos en donde se almacenan los nodos que se van generando, y que puede tratarse de una pila, una cola o un montículo (según se siga una estrategia LIFO, FIFO o LC).
3. Finalmente, necesitamos un módulo que describa e implemente las estructuras de datos que conforman los nodos.

El primero de los tres módulos no se modifica a lo largo del capítulo, pues es válido para todos los algoritmos que siguen la técnica de Ramificación y Poda, y lo presentamos a continuación:

```
MODULE Esquema;
FROM IO IMPORT WrStr, WrCard, WrLn;
FROM Estruc IMPORT Estructura,Crear,Anadir,Extraer,EsVacia,
                   Tamano,Destruir;
FROM Nodos IMPORT nodo,NodoInicial,MAXHIJOS,Expandir,EsAceptable,
           EsSolucion,h,Eliminar,NoHaySolucion,Imprimir,PonerCota;
VAR   numgenerados,        (* numero total de nodos generados *)
      numanalizados,       (* numero total de nodos analizados *)
      numpodados:CARDINAL; (* numero total de nodos podados *)

PROCEDURE RyP_una():nodo; (* encuentra la primera solucion *)
   VAR E:Estructura; (* estructura para almacenar los nodos *)
       n:nodo; (* nodo vivo en curso *)
       hijos:ARRAY [1..MAXHIJOS] OF nodo; (* hijos de un nodo *)
       numhijos,i,j:CARDINAL;
BEGIN
   E:=Crear();   (* inicializamos las estructuras *)
   n:=NodoInicial(); Anadir(E,n,h(n)); (*h es la funcion de coste*)
   WHILE NOT EsVacia(E) DO
      n:=Extraer(E); INC(numanalizados);
      numhijos:=Expandir(n,hijos); INC(numgenerados,numhijos);
      Eliminar(n);
```

```
      FOR i:=1 TO numhijos DO
         IF EsAceptable(hijos[i]) THEN
            IF EsSolucion(hijos[i]) THEN (* Eureka! *)
               FOR j:=1 TO numhijos DO (*eliminamos resto de hijos*)
                  IF i<>j THEN Eliminar(hijos[j]) END;
               END;
               Destruir(E);
               RETURN hijos[i]  (* devolvemos la solucion *)
            ELSE
               Anadir(E,hijos[i],h(hijos[i]))
            END;
         ELSE
            Eliminar(hijos[i]); INC(numpodados)
         END;
      END;
   END;
   Destruir(E);
   RETURN NoHaySolucion();
END RyP_una;

(* programa principal del esquema *)
   VAR n:nodo;
BEGIN
   numgenerados:=0;  numanalizados:=0; numpodados:=0;
   n:=RyP_una();
   WrStr("Nodos Generados:  "); WrCard(numgenerados,4); WrLn();
   WrStr("Nodos Analizados: "); WrCard(numanalizados,4); WrLn();
   WrStr("Nodos Podados:    "); WrCard(numpodados,4); WrLn();
END Esquema.
```

Como podemos ver, además de encontrar una solución, el programa calcula tres datos, el número de nodos generados, el número de nodos analizados, y el número de nodos podados, los cuales permiten analizar el algoritmo y poder comparar distintas estrategias y funciones LC.

a) El primero de ellos (*numgenerados*) nos da información sobre el trabajo que ha tenido que realizar el algoritmo hasta encontrar la solución. Mientras más pequeño sea este valor, menos parte del árbol de expansión habrá tenido que construir, y por tanto más rápido será el proceso.

b) El segundo valor (*numanalizados*) nos indica el número de nodos que el algoritmo ha tenido que analizar, para lo cual es necesario extraerlos de la estructura y comprobar si han de ser podados y, si no, expandirlos. Éste es el valor más importante de los tres, pues indica el número de nodos del árbol de expansión que se recorren efectivamente. En consecuencia, es deseable que este valor sea pequeño.

c) Por último, el número de nodos podados nos da una indicación de la efectividad de la función de poda y las restricciones impuestas al problema. Mientras mayor sea este valor, más trabajo ahorramos al algoritmo.

Disponer de esta forma fácil y modular de cambiar las estrategias de selección de los nodos vivos (mediante el módulo "Estruc") junto con los valores de estos tres parámetros nos permitirá analizar el algoritmo de Ramificación y Poda de una forma sencilla, cómoda y eficaz, y en consecuencia escoger la mejor de las estrategias para un problema dado.

Si lo que deseamos es encontrar no sólo una solución al problema sino todas, observamos que es posible conseguirlo con una pequeña variación del esquema anterior:

```
PROCEDURE RyP_todas(VAR todas:ARRAY OF nodo):CARDINAL;
(* encuentra todas las soluciones del problema y
   devuelve el numero de soluciones que hay *)
VAR E:Estructura; (* estructura para almacenar los nodos *)
    n:nodo; (* nodo vivo en curso *)
    hijos:ARRAY [1..MAXHIJOS] OF nodo; (* hijos de un nodo*)
    numhijos,i,j,numsol:CARDINAL;
BEGIN
   E:=Crear();
   n:=NodoInicial();
   Anadir(E,n,h(n));
   numsol:=0;
   WHILE NOT EsVacia(E) DO (* analiza todo el arbol *)
      n:=Extraer(E); INC(numanalizados);
      numhijos:=Expandir(n,hijos); INC(numgenerados,numhijos);
      Eliminar(n);
      FOR i:=1 TO numhijos DO
         IF EsAceptable(hijos[i]) THEN
            IF EsSolucion(hijos[i]) THEN (* Eureka! *)
               todas[numsol]:=hijos[i]; INC(numsol)
            ELSE
               Anadir(E,hijos[i],h(hijos[i]))
            END;
         ELSE
            Eliminar(hijos[i]); INC(numpodados)
         END;
      END;
   END;
   Destruir(E);
   RETURN numsol;
END RyP_todas;
```

También vamos a considerar una tercera versión del algoritmo para cuando necesitemos encontrar la mejor de entre todas las soluciones de un problema:

```
PROCEDURE RyP_lamejor():nodo;
VAR E:Estructura; (* estructura para almacenar los nodos *)
    n,solucion:nodo;
    hijos:ARRAY [1..MAXHIJOS] OF nodo; (* hijos de un nodo*)
    numhijos,i,j,valor,valor_solucion:CARDINAL;
BEGIN
   E:=Crear(); n:=NodoInicial(); Anadir(E,n,h(n));
   solucion:=NoHaySolucion(); valor_solucion:=MAX(CARDINAL);
   PonerCota(valor_solucion);
   WHILE NOT EsVacia(E) DO
      n:=Extraer(E); INC(numanalizados);
      numhijos:=Expandir(n,hijos); INC(numgenerados,numhijos);
      Eliminar(n);
      FOR i:=1 TO numhijos DO
         IF EsAceptable(hijos[i]) THEN
            IF EsSolucion(hijos[i]) THEN (* Eureka! *)
               valor:=Valor(hijos[i]);
               IF valor<valor_solucion THEN
                  Eliminar(solucion); solucion:=hijos[i];
                  valor_solucion:=valor; PonerCota(valor);
               END;
            ELSE
               Anadir(E,hijos[i],h(hijos[i]))
            END;
         ELSE
            Eliminar(hijos[i]); INC(numpodados)
         END;
      END;
   END;
   Destruir(E);
   RETURN solucion;
END RyP_lamejor;
```

Una vez disponemos del esquema de este tipo de algoritmos, vamos a definir el interfaz de los tipos abstractos de datos que representan los nodos y la estructura de datos para almacenarlos. En primer lugar, el módulo Nodos ha de implementar los siguientes procedimientos y funciones:

```
DEFINITION MODULE Nodos;
CONST MAXHIJOS = ...; (* numero maximo de hijos de un nodo *)
TYPE nodo;
PROCEDURE NodoInicial():nodo; (* raiz del arbol *)
PROCEDURE Expandir(n:nodo;VAR hijos:ARRAY OF nodo):CARDINAL;
```

```
PROCEDURE EsAceptable(n:nodo):BOOLEAN;
PROCEDURE EsSolucion(n:nodo):BOOLEAN;
PROCEDURE h(n:nodo):CARDINAL;
PROCEDURE PonerCota(c:CARDINAL);
PROCEDURE Valor(n:nodo):CARDINAL;
PROCEDURE Eliminar(VAR n:nodo);
PROCEDURE NoHaySolucion():nodo;
PROCEDURE Imprimir(n:nodo);
END Nodos.
```

- De ellas, *NodoInicial* es la que devuelve el nodo que constituye la raíz del árbol de expansión para el problema. A partir de este nodo se origina el árbol, expandiendo progesivamente los nodos mediante la siguiente función.
- *Expandir* es la función que construye los nodos hijos de un nodo dado, y devuelve el número de hijos que ha generado. Esta función es la que realiza el proceso de ramificación del algoritmo.
- La función *EsAceptable* es la que realiza la poda, y dado un nodo vivo decide si seguir analizándolo o bien rechazarlo.
- *EsSolucion* es una función que decide cuándo un nodo es una hoja del árbol, esto es, una posible solución al problema original. Obsérvese que tal solución no ha de ser necesariamente la mejor, sino una cualquiera de ellas.
- Por su parte, la función *h* es la que implementa la función de coste de la estrategia LC, y que se utiliza como prioridad en la estructura de montículo.
- La función *Valor* devuelve el valor asociado a un nodo, y se utiliza para comparar soluciones con la cota superior encontrada hasta el momento.
- La función *PonerCota* permite establecer la cota superior del problema. Usualmente la función que realiza la poda utiliza este dato para podar aquellos nodos cuyo valor (calculado mediante la función *Valor*) sea superior a la cota ya obtenida de una solución. El código de esta función va a ser común para los ejemplos desarrollados en este capítulo, y por lo tanto lo incluimos aquí:

  ```
  PROCEDURE PonerCota(c:CARDINAL);
  BEGIN
     cota:=c;
  END PonerCota;
  ```

  siendo *cota* una variable global (aunque privada) del módulo "Nodos" que se utiliza para almacenar la cota inferior del problema alcanzada hasta ese momento por alguna solución. Nótese que hablamos de cota inferior puesto que el esquema presentado permite resolver problemas de minimización. En el ejemplo de la mochila presentado en este capítulo se discute la solución de los problemas de maximización.
- La función *NoHaySolucion* es la que devuelve un nodo con valor especial, necesario para indicar que no se encuentra solución al problema.
- Por último, las funciones *Eliminar* e *Imprimir* son las que destruyen e imprimen un nodo, respectivamente.

En cuanto al tipo abstracto de datos que representa la estructura donde se almacenan los nodos, su interfaz es el siguiente:

```
DEFINITION MODULE Estruc;
   FROM Nodos IMPORT nodo;
   TYPE Estructura;
PROCEDURE Crear():Estructura;
PROCEDURE Anadir(VAR h:Estructura;n:nodo;prioridad:CARDINAL);
PROCEDURE Extraer(VAR h:Estructura):nodo;
PROCEDURE EsVacia(h:Estructura):BOOLEAN;
PROCEDURE Tamano(h:Estructura):CARDINAL;
PROCEDURE Destruir(VAR h:Estructura);
END Estruc.
```

Hemos llamado a este tipo abstracto *Estructura* porque puede corresponder a una cola, una pila o un montículo invertido (en la raíz del árbol se encuentra el menor elemento puesto que tratamos con problemas de minimización) dependiendo de la estrategia que queramos implementar en nuestro algoritmo de Ramificación y Poda (FIFO, LIFO o LC). No consideramos necesario incluir su implementación, pues ni aporta nada nuevo a la técnica ni presenta ninguna dificultad especial.

Utilizando este esquema conseguimos reducir la programación de los algoritmos de Ramificación y Poda a la implementación de las funciones que conforman los nodos, lo que reduce notablemente la dificultad de implementación de este tipo de algoritmos. Por consiguiente, para la resolución de los problemas planteados en este capítulo será suficiente dar una implementación del módulo "Nodos" de cada uno de ellos.

7.3 EL PUZZLE (n^2–1)

Este juego es una generalización del Puzzle-15 ideado por Sam Loyd en 1878. Disponemos de un tablero con n^2 casillas y de n^2–1 piezas numeradas del uno al n^2–1. Dada una ordenación inicial de todas las piezas en el tablero, queda sólo una casilla vacía, a la que denominaremos "hueco". Nuestro objetivo es transformar, mediante movimientos legales de la fichas, dicha disposición inicial de las piezas en una disposición final ordenada, en donde en la casilla [i,j] se encuentra la pieza numerada (i–1)*n + j y en la casilla [n,n] se encuentra el hueco.

Los únicos movimientos permitidos son los de las piezas adyacentes (horizontal y verticalmente) al hueco, que pueden ocuparlo; al hacerlo, dejan el hueco en la posición en donde se encontraba la pieza antes del movimiento.

Otra forma de abordar el problema es considerar que lo que se mueve es el hueco, pudiendo hacerlo hacia arriba, abajo, izquierda o derecha (siempre sin salirse del tablero). Al moverse, su casilla es ocupada por la pieza que ocupaba la casilla a donde se ha "movido" el hueco. Por ejemplo, para el caso n = 3 se muestra a continuación una disposición inicial junto con la disposición final:

Disposición Inicial Disposición Final

1	5	2
4	3	
7	8	6

1	2	3
4	5	6
7	8	

Es posible resolver el problema mediante Ramificación y Poda utilizando dos funciones de coste diferentes:

a) La primera calcula el número de piezas que están en una posición distinta de la que les corresponde en la disposición final.

b) La segunda se basa en la suma de las distancias de Manhattan desde la posición de cada pieza a su posición en la disposición final. La distancia de Manhattan entre dos puntos del plano de coordenadas (x_1,y_1) y (x_2,y_2) viene dada por la expresión $|x_1 - x_2| + |y_1 - y_2|$.

Se pide resolver este problema utilizando ambas funciones de coste, y comparar los resultados que se obtienen para ambas funciones.

Solución (𝒢)

Para resolver este problema es necesario en primer lugar construir su árbol de expansión, y para ello hemos de plantearlo como una secuencia de decisiones, una en cada nivel del árbol.

Por tanto, partiendo de una disposición del tablero, consideraremos como posibles decisiones a tomar los cuatro movimientos del hueco (arriba, abajo, izquierda y derecha) siempre que éstos sean válidos, es decir, siempre que no caigan fuera del tablero.

Así por ejemplo, partiendo de la disposición inicial mostrada en el enunciado del problema tenemos tres opciones válidas:

1	5	
4	3	2
7	8	6

1	5	2
4	3	6
7	8	

1	5	2
4		3
7	8	6

A partir de esta idea vamos a construir el módulo de implementación asociado a los nodos que es, según hemos comentado en la introducción de este capítulo, lo único que necesitamos para resolver el problema.

Una primera aproximación a la solución del problema consiste en definir cada uno de los nodos como un tablero, es decir:

```
CONST dim = ...; (* dimension del puzzle *)
TYPE puzzle = ARRAY [1..dim],[1..dim] OF CARDINAL;
TYPE nodo = POINTER TO puzzle;
```

Sin embargo, esto no va a ser suficiente puesto que no disponemos de "historia" sobre los movimientos ya realizados, lo cual nos llevaría posiblemente a ciclos en donde repetiríamos indefinidamente movimientos de forma circular.

Aparte de esta razón, también hemos de recordar que los nodos utilizados en la técnica de Ramificación y Poda han de ser autónomos, es decir, han de contener toda la información necesaria para recuperar a partir de ellos la solución construida hasta el momento. En nuestro caso la solución ha de estar compuesta por una sucesión de tableros que muestran la serie de movimientos a realizar para llegar a la disposición final.

Por tanto, la definición de nodos que vamos a utilizar es:

```
CONST dim = ...; (* dimension del puzzle *)
TYPE puzzle = ARRAY [1..dim],[1..dim] OF CARDINAL;
TYPE nodo = POINTER TO RECORD p:puzzle; sig:nodo; END;
```

Otra posibilidad sería la de utilizar una lista global con todos aquellos tableros que ya han sido considerados, y que fuera utilizada durante el proceso de bifurcación de todos los nodos para comprobar que no se generan duplicados. De esta forma también se eliminarían los ciclos. La diferencia de propuesta es que esa lista sería global a todos los nodos, mientras que en la primera cada nodo tiene la lista de los nodos que él ha generado. En este problema haremos uso de la primera de las opciones.

Una vez disponemos de la representación de los valores del tipo abstracto de datos, implementaremos sus operaciones. En primer lugar, la función *NodoInicial* habrá de contener la disposición inicial del tablero:

```
PROCEDURE NodoInicial():nodo;
   VAR x:nodo;
BEGIN
   NEW(x);
   x^.p[1,1]:=1; x^.p[1,2]:=5; x^.p[1,3]:=2;
   x^.p[2,1]:=4; x^.p[2,2]:=3; x^.p[2,3]:=0;
   x^.p[3,1]:=7; x^.p[3,2]:=8; x^.p[3,3]:=6;
   x^.sig:=NIL;
   RETURN x;
END NodoInicial;
```

La estrategia de ramificación está a cargo de la función *Expandir*:

```
PROCEDURE Expandir(n:nodo;VAR hijos:ARRAY OF nodo):CARDINAL;
   VAR i,j,nhijos:CARDINAL;
       p:nodo;
BEGIN
   BuscaHueco(n,i,j); (* primero buscamos el "hueco" *)
   nhijos:=0;
   (* y ahora vemos a donde lo podemos "mover" *)
   IF i<dim THEN (* abajo *)
```

```
      INC(nhijos);
      Copiar(n,p);
      p^.p[i,j]:=p^.p[i+1,j]; p^.p[i+1,j]:=0;
      hijos[nhijos-1]:=p;
   END;
   IF j<dim THEN (* derecha *)
      INC(nhijos);
      Copiar(n,p);
      p^.p[i,j]:=p^.p[i,j+1]; p^.p[i,j+1]:=0;
      hijos[nhijos-1]:=p;
   END;
   IF (i-1)>0 THEN (* arriba *)
      INC(nhijos);
      Copiar(n,p);
      p^.p[i,j]:=p^.p[i-1,j]; p^.p[i-1,j]:=0;
      hijos[nhijos-1]:=p;
   END;
   IF (j-1)>0 THEN (* izquierda *)
      INC(nhijos);
      Copiar(n,p);
      p^.p[i,j]:=p^.p[i,j-1]; p^.p[i,j-1]:=0;
      hijos[nhijos-1]:=p;
   END;
   RETURN nhijos;
END Expandir;
```

Una de las primeras dudas que nos asaltan tras implementar esta función es si el orden en el que se bifurque va a influir en la eficiencia del algoritmo, tal como sucedía en algunos problemas de Vuelta Atrás. Realmente, el orden de ramificación sí es importante cuando el árbol de expansión se recorre siguiendo una estrategia "ciega" (FIFO o LIFO). Sin embargo, puesto que en este problema vamos a utilizar una estrategia LC, el orden en el que se generen los nodos (y se inserten en la estructura) no va a tener una influencia de peso en el comportamiento final del algoritmo.

Esto también lo hemos probado de forma experimental, pues una vez obtenidos los resultados finales decidimos cambiar este orden, y generar nodos moviendo el hueco en el sentido inverso a las agujas del reloj y comenzando por arriba (a priori es la peor manera, pues el hueco ha de tratar de ir hacia la posición $[n,n]$). Los resultados obtenidos de esta forma no presentan ningún cambio sustancial respecto a los anteriores, lo que corrobora el hecho de que en este caso el orden de generación de los hijos no es influyente. En cualquier caso, esta afirmación es cierta para este problema pero no tiene por qué ser válida para cualquier otro; obsérvese que en este caso el número de hijos que genera cada nodo es pequeño (a lo más cuatro). Para problemas en los que el número de hijos que expande cada nodo es grande, sí que puede tener influencia el orden de generación de los mismos.

Volviendo a la función *Expandir*, su implementación hace uso de varios procedimientos auxiliares, que presentamos a continuación:

```
PROCEDURE BuscaHueco(n:nodo;VAR i,j:CARDINAL);
(* busca el hueco en un tablero *)
   VAR a,b:CARDINAL;
BEGIN
   FOR a:=1 TO dim DO
      FOR b:=1 TO dim DO
         IF n^.p[a,b]=0 THEN
            i:=a; j:=b; RETURN
         END
      END
   END
END BuscaHueco;
```

También necesita una función para copiar un nodo y añadirle el tablero que va a contener el siguiente movimiento:

```
PROCEDURE Copiar(VAR n1,n2:nodo);
   VAR i,j:CARDINAL; nuevo:nodo;
BEGIN
   NEW(n2);
   Duplicar(n1,nuevo);
   FOR i:=1 TO dim DO
      FOR j:=1 TO dim DO
         n2^.p[i,j]:=n1^.p[i,j]
      END
   END;
   n2^.sig:=nuevo;
END Copiar;
```

Esta función utiliza la que duplica un nodo dado:

```
PROCEDURE Duplicar(VAR n1,n2:nodo);
   VAR i,j:CARDINAL;
BEGIN
   NEW(n2);
   FOR i:=1 TO dim DO
      FOR j:=1 TO dim DO
         n2^.p[i,j]:=n1^.p[i,j]
      END
   END;
   n2^.sig:=n1^.sig;
   IF n1^.sig<>NIL THEN Duplicar(n1^.sig,n2^.sig) END
END Duplicar;
```

Continuando con la implementación del módulo “Nodos”, también es necesario implementar la función que realiza la poda. En este caso, vamos a podar aquellos nodos cuyo último movimiento haga aparecer un ciclo:

```
PROCEDURE EsAceptable(n:nodo):BOOLEAN;
(* mira si ese movimiento ya lo ha hecho antes *)
   VAR aux:nodo;
BEGIN
   aux:=n^.sig;
   WHILE aux<>NIL DO
      IF SonIguales(n,aux) THEN RETURN FALSE END;
      aux:=aux^.sig;
   END;
   RETURN TRUE;
END EsAceptable;
```

A su vez, esta función utiliza otra que permite decidir cuándo dos tableros son iguales:

```
PROCEDURE SonIguales(n1,n2:nodo):BOOLEAN;
VAR i,j:CARDINAL;
BEGIN
   FOR i:=1 TO dim DO
      FOR j:=1 TO dim DO
         IF n1^.p[i,j]<>n2^.p[i,j] THEN RETURN FALSE END
      END
   END;
   RETURN TRUE;
END SonIguales;
```

Una función que también es necesario implementar es la que define la función de coste. Para este problema vamos a implementar dos, una para cada una de las estrategias mencionadas en el enunciado. La primera de ellas va a contar el número de piezas que se encuentran fuera de su sitio:

```
PROCEDURE h(n:nodo):CARDINAL;
(* cuenta el numero de piezas fuera de su posicion final *)
   VAR i,j,cuenta:CARDINAL;
BEGIN
   cuenta:=0;
   FOR i:=1 TO dim DO FOR j:=1 TO dim DO
        IF n^.p[i,j]<>((j+(i-1)*dim)MOD(dim*dim)) THEN INC(cuenta) END
   END END;
   RETURN cuenta;
END h;
```

La segunda corresponde a la suma de las distancias de Manhattan de la posición de cada pieza a su casilla final, y que hace uso de una función que calcula el valor absoluto de la diferencia de dos números naturales:

```
PROCEDURE ValAbs(a,b:CARDINAL):CARDINAL;
(* valor absoluto de la diferencia de sus argumentos: |a-b| *)
BEGIN
   IF a>b THEN RETURN a-b ELSE RETURN b-a END;
END ValAbs;

PROCEDURE h2(n:nodo):CARDINAL;
(* calcula la suma de las distancias de Manhattan *)
   VAR i,j,x,y,cuenta:CARDINAL;
BEGIN
   cuenta:=0;
   FOR i:=1 TO dim DO
      FOR j:=1 TO dim DO
         IF n^.p[i,j] = 0 THEN
            x:=dim; y:=dim
         ELSE
            x:=((n^.p[i,j]-1) DIV dim)+1;
            y:=((n^.p[i,j]-1) MOD dim)+1;
         END;
         cuenta:=cuenta+ValAbs(x,i)+ValAbs(y,j);
      END
   END;
   RETURN cuenta;
END h2;
```

También es preciso implementar una función para determinar cuándo un nodo es solución. En nuestro caso consiste en decidir cuándo un tablero coincide con la disposición final del juego y para esto es suficiente comprobar que su función de coste vale cero (cualquiera de las dos funciones vistas):

```
PROCEDURE EsSolucion(n:nodo):BOOLEAN;
BEGIN
   RETURN h(n)=0;
END EsSolucion;
```

También es necesario implementar la función *NoHaySolucion*, que devuelve un valor especial para indicar que el problema no admite solución:

```
PROCEDURE NoHaySolucion():nodo;
BEGIN
   RETURN NIL;
END NoHaySolucion;
```

Obsérvese que esto puede ocurrir puesto que no todas las disposiciones iniciales de un puzzle permiten llegar a la disposición final, como por ejemplo ocurre para la siguiente disposición inicial:

1	3	2
4	5	6
7	8	

Por último, la función *Eliminar* es la que va a devolver al sistema los recursos ocupados por un nodo, actuando como destructor del tipo abstracto de datos:

```
PROCEDURE Eliminar(VAR n:nodo);
BEGIN
   IF n<>NIL THEN
      IF n^.sig<>NIL THEN Eliminar(n^.sig) END;
      DISPOSE(n);
   END;
END Eliminar;
```

El procedimiento *Imprimir* no plantea mayores problemas, y su implementación va a depender de lo que el usuario desee consultar sobre un nodo. En cuanto a las funciones *Valor* y *PonerCota*, como nuestro problema consiste en encontrar la primera solución, no tienen relevancia alguna.

Una vez implementado el módulo "Nodos" es el momento de analizar su comportamiento. Para ello haremos uso de los resultados que nos da el programa principal que contiene el esquema, y que mostramos en las siguientes tablas. Cada una de ellas contiene a la izquierda la disposición inicial de partida, y los valores obtenidos utilizando cada una de las dos funciones LC que hemos implementado. Hemos llamado LC_1 a la función de coste que contaba el número de piezas fuera de su sitio, y LC_2 a la otra.

Disposición Inicial

1	5	2
4	3	
7	8	6

Resultados Obtenidos

	LC_1	LC_2
Núm. nodos generados	19	23
Núm. nodos analizados	7	8
Núm. nodos podados	5	6

1	3	5
7		2
8	4	6

	LC_1	LC_2
Núm. nodos generados	38	48
Núm. nodos analizados	13	17
Núm. nodos podados	11	15

4	1	5
7		2
8	3	6

	LC_1	LC_2
Núm. nodos generados	47	194
Núm. nodos analizados	17	71
Núm. nodos podados	15	69

Como puede apreciarse, la primera función de coste se comporta de forma más eficaz que la segunda.

7.4 EL VIAJANTE DE COMERCIO

Analicemos una vez más el problema del viajante de comercio, presentado ya en el capítulo cuatro, y cuyo enunciado reza como sigue. Se conocen las distancias entre un cierto número de ciudades. Un viajante debe, a partir de una de ellas, visitar cada ciudad exactamente una vez y regresar al punto de partida habiendo recorrido en total la menor distancia posible. Más formalmente, dado un grafo *g* conexo y ponderado, y dado uno de sus vértices v_0, queremos encontrar el ciclo Hamiltoniano de coste mínimo que comienza y termina en v_0.

Solución ()

El problema del viajante de comercio admite numerosas estrategias de ramificación y poda, y casi cada autor que describe el problema emplea una distinta, o incluso varias. Nosotros utilizaremos la primera de las tres estrategias descritas en [HOR78] para solucionar el problema.

Comenzaremos analizando la construcción del árbol de expansión para el problema. En primer lugar, hemos de plantear la solución como una secuencia de decisiones, una en cada paso o etapa.

Para ello, nuestra solución estará formada por un vector que va a indicar el orden en el que deberán ser visitados los vértices. Cada elemento del vector contendrá un número entre 1 y *N*, siendo *N* el número de vértices del grafo que define el problema. Es preciso indicar aquí que utilizaremos una representación del grafo en donde los vértices están numerados consecutivamente comenzando por 1, y los arcos vienen definidos mediante una matriz de adyacencia, no necesariamente simétrica en este caso, aunque sí de elementos no negativos.

De esta forma, inicialmente el vector solución estará compuesto por un solo elemento, el 1 (que es el vértice origen), y en cada paso *k* tomaremos la decisión de

qué vértice incluimos en el recorrido. Por tanto, los valores que puede en principio tomar el elemento en posición k del vector ($1 \leq k \leq N$) estarán comprendidos entre 1 y N pero sin poder repetirse, esto es, no puede haber dos elementos iguales en el vector. Por tanto, cada nodo podrá generar hasta $N–k$ hijos. Este mecanismo es el que va construyendo el árbol de expansión para el problema.

Teniendo en cuenta las consideraciones realizadas en la introducción de este capítulo, será suficiente realizar el módulo que implementa el tipo abstracto de datos que representa los nodos, pues el resto del programa es fijo para estos algoritmos.

Respecto a la información que debe contener cada uno de ellos, hemos de conseguir que cada nodo sea "autónomo", esto es, que cada uno contenga toda la información relevante para poder realizar los procesos de bifurcación, poda y reconstrucción de la solución encontrada hasta ese momento. En consecuencia, al menos ha de contar con el nivel en donde se encuentra y con el vector solución construido hasta el momento. Por otro lado, también debe llevar información para realizar la poda. En este sentido vamos a incluir una *matriz de costes reducida* .

Diremos que una fila (columna) de una matriz está *reducida* si contiene al menos un elemento cero, y el resto de los elementos son no negativos. Una matriz se dice *reducida* si y sólo si todas sus filas y columnas están reducidas. Por ejemplo, dada la matriz de adyacencia:

∞	15	7	4	20
1	∞	16	6	5
8	20	∞	4	10
4	7	14	∞	3
10	35	15	4	∞

podemos calcular su matriz reducida restando respectivamente 4, 1, 4, 3 y 4 a cada fila, y luego 4 y 3 a las columnas 2 y 3, obteniendo la matriz:

∞	7	0	0	16
0	∞	12	5	4
4	12	∞	0	6
1	0	8	∞	0
6	27	8	0	∞

En total hemos restado un valor de 23 (4 + 1 + 4 + 3 + 4 + 4 + 3), que es lo que denominaremos el *coste* de la matriz.

De esta forma, dada una matriz de adyacencia de un grafo ponderado podemos obtener su matriz reducida calculando los mínimos de cada una de las filas y restándoselos a los elementos de esas filas, haciendo después lo mismo con las columnas.

Respecto a la interpretación del *coste*, pensemos que restando una cantidad t a una fila o a una columna decrementamos en esa cantidad el coste de los recorridos del grafo. Por tanto, un camino mínimo lo seguirá siendo tras una operación de sustracción de filas o columnas. En cuanto a la cantidad total sustraída al reducir una matriz, ésta será una cota inferior del coste total de sus recorridos. En consecuencia, para el ejemplo anterior hemos obtenido que 23 es una cota inferior para la solución al problema del viajante.

Esto es justo lo que vamos a utilizar como función de coste LC para podar nodos del árbol de expansión. Así, a cada nodo le vamos a asociar una matriz reducida y un coste acumulado. Para ver cómo trabajamos con ellos, supongamos que M es la matriz reducida asociada al nodo n, y sea n' el hijo de n que se obtiene incluyendo el arco $\{i,j\}$ en el recorrido.

- Si n' es una hoja del árbol, esto es, una posible solución, su coste va a venir dado por el coste que llevaba n acumulado más $M[i,j]+M[j,1]$, que es lo que completa el recorrido. Esta cantidad coincide además con el coste de tal recorrido.
- Por otro lado, si n' no es una hoja, su matriz de costes reducida M' vamos a calcularla a partir de los valores de M como sigue:
  a) En primer lugar, hay que sustituir todos los elementos de la fila i y de la columna j por ∞. Esto elimina el posterior uso de aquellos caminos que parten del vértice i y de los que llegan al vértice j.
  b) En segundo lugar, debemos asignar $M'[j,1]=\infty$, eliminando la posibilidad de acabar el recorrido en el siguiente paso (recordemos que n' no era una hoja).
  c) Reducir entonces la matriz M', y ésta es la matriz que asignamos al nodo n'.

  Como coste para n' vamos a tomar el coste de n más el coste de la reducción de M' más, por supuesto, el valor de $M[i,j]$. Es importante señalar en este punto que la reducción no se realiza teniendo en cuenta los elementos con valor ∞, no obteniéndose coste alguno en aquellas filas o columnas cuyos elementos tomen todos tal valor.

Con todo esto, comenzaremos definiendo el tipo *nodo* que utilizaremos en la implementación del algoritmo de Ramificación y Poda que resuelve el problema. Vamos a utilizar entonces la siguiente estructura de datos:

```
CONST N = ...; (* numero de vertices del grafo *)
TYPE solucion = ARRAY[1..N] OF CARDINAL;
TYPE mat_ady = ARRAY[1..N],[1..N] OF CARDINAL;
TYPE nodo = POINTER TO RECORD
                         coste:CARDINAL; (* coste acumulado *)
                         matriz:mat_ady; (* matriz reducida *)
                         k:CARDINAL;     (* nivel *)
                         s:solucion
                       END;
```

Además del vector solución y el nivel, los otros componentes del registro indican el coste acumulado hasta el momento, así como la matriz reducida asociada al nodo.

Necesitaremos también una variable global al módulo para almacenar la cota superior alcanzada por la mejor solución hasta el momento:

```
VAR cota:CARDINAL;
```

Esta variable será inicializada en el cuerpo principal del módulo "Nodos":

```
BEGIN (* Nodos *)
   cota:=MAX(CARDINAL);
END Nodos.
```

Veamos ahora las funciones de este módulo. En primer lugar la función *NodoInicial* habrá de contener el nodo raíz del árbol de expansión:

```
PROCEDURE NodoInicial():nodo;
    VAR n:nodo; i,j:CARDINAL; m:mat_ady;
BEGIN
    (* aqui se introduce la matriz de adyacencia del grafo *)
    FOR i:=1 TO N DO m[i,i]:=MAX(CARDINAL) END;
                m[1,2]:=15; m[1,3]:=7 ; m[1,4]:=4 ; m[1,5]:=20;
    m[2,1]:=1 ;             m[2,3]:=16; m[2,4]:=6 ; m[2,5]:=5 ;
    m[3,1]:=8 ; m[3,2]:=20;             m[3,4]:=4 ; m[3,5]:=10;
    m[4,1]:=4 ; m[4,2]:=7 ; m[4,3]:=14;             m[4,5]:=3 ;
    m[5,1]:=10; m[5,2]:=35; m[5,3]:=15; m[5,4]:=4 ;
    (* ahora, generamos el primer nodo *)
    NEW(n);
    FOR i:=2 TO N DO n^.s[i]:=0 END;
    n^.matriz:=m;
    n^.coste:=Reducir(n^.matriz);
    n^.s[1]:=1; (* incluimos el primer vertice *)
    n^.k:=1;
    RETURN n;
END NodoInicial;
```

Como podemos observar, se introduce ya en la solución el vértice origen, y la matriz que se asocia a este nodo es la reducida de la original. El procedimiento que se encarga de la reducción es el siguiente:

```
PROCEDURE Reducir(VAR m:mat_ady):CARDINAL;
(* devuelve el coste total reducido a la matriz *)
    VAR i,j,coste,minimo:CARDINAL;
BEGIN
    coste:=0;
    FOR i:=1 TO N DO (* primero por filas *)
```

```
            minimo:=CosteFil(m,i);
            IF (minimo>0) AND (minimo<MAX(CARDINAL)) THEN
                QuitarFil(m,i,minimo); INC(coste,minimo)
            END
        END;
        FOR j:=1 TO N DO (* despues por columnas *)
            minimo:=CosteCol(m,j);
            IF (minimo>0) AND (minimo<MAX(CARDINAL)) THEN
                QuitarCol(m,j,minimo); INC(coste,minimo)
            END
        END;
        RETURN coste;
    END Reducir;
```

Para lograr su objetivo, se basa en los procedimientos que calculan el mínimo de una fila y se lo restan a los elementos de tal fila, y los análogos para las columnas:

```
    PROCEDURE CosteFil(m:mat_ady;i:CARDINAL):CARDINAL;
        VAR j,c:CARDINAL;
    BEGIN
        c:=m[i,1];
        FOR j:=2 TO N DO
            IF m[i,j]<c THEN c:=m[i,j] END;
        END;
        RETURN c
    END CosteFil;

    PROCEDURE CosteCol(m:mat_ady;j:CARDINAL):CARDINAL;
        VAR i,c:CARDINAL;
    BEGIN
        c:=m[1,j];
        FOR i:=2 TO N DO
            IF m[i,j]<c THEN c:=m[i,j] END;
        END;
        RETURN c
    END CosteCol;

    PROCEDURE QuitarFil(VAR m:mat_ady;i:CARDINAL;minimo:CARDINAL);
        VAR j:CARDINAL;
    BEGIN
        FOR j:=1 TO N DO
            IF m[i,j]<MAX(CARDINAL) THEN m[i,j]:=m[i,j]-minimo END;
        END;
    END QuitarFil;
```

```
PROCEDURE QuitarCol(VAR m:mat_ady;j:CARDINAL;minimo:CARDINAL);
    VAR i:CARDINAL;
BEGIN
    FOR i:=1 TO N DO
        IF m[i,j]<MAX(CARDINAL) THEN m[i,j]:=m[i,j]-minimo END;
    END;
END QuitarCol;
```

Por otro lado, la estrategia de ramificación está a cargo de la función *Expandir*. Cada nodo puede generar, como hemos dicho antes, a lo sumo *N–k* hijos, que son los correspondientes a los vértices aún no incluidos en el recorrido.

```
PROCEDURE Expandir(n:nodo;VAR hijos:ARRAY OF nodo):CARDINAL;
    VAR nk,i,j,l,coste,nhijos:CARDINAL;
    p:nodo;
BEGIN
    nhijos:=0;
    nk:=n^.k+1;
    i:=n^.s[nk-1];
    IF nk>N THEN  (* caso especial *)
        RETURN nhijos
    END;
    FOR j:=1 TO N DO
        IF NoEsta(n^.s,nk-1,j) THEN
            INC(nhijos);
            Copiar(n,p);
            p^.s[nk]:=j;
            IF nk=N THEN (* recorrido completo *)
                INC(p^.coste,n^.matriz[i,j]+n^.matriz[j,1])
            ELSE
              FOR l:=1 TO N DO
                  p^.matriz[i,l]:=MAX(CARDINAL);
                  p^.matriz[l,j]:=MAX(CARDINAL);
              END;
              p^.matriz[j,1]:=MAX(CARDINAL);
              INC(p^.coste,Reducir(p^.matriz)+n^.matriz[i,j]);
            END;
            INC(p^.k);
            hijos[nhijos-1]:=p;
        END
    END;
    RETURN nhijos;
END Expandir;
```

Esta función hace uso de un procedimiento que permite duplicar un nodo:

```
PROCEDURE Copiar(VAR n1,n2:nodo);
    VAR i,j:CARDINAL;
BEGIN
    NEW(n2);
    n2^.s:=n1^.s; n2^.matriz:=n1^.matriz;
    n2^.coste:=n1^.coste; n2^.k:=n1^.k;
END Copiar;
```

Y también de otra función para determinar si un vértice del grafo está ya incluido o no en el recorrido:

```
PROCEDURE NoEsta(s:solucion;k,j:CARDINAL):BOOLEAN;
    VAR i:CARDINAL;
BEGIN
    FOR i:=1 TO k DO
        IF s[i]=j THEN RETURN FALSE END
    END;
    RETURN TRUE;
END NoEsta;
```

Es necesario implementar la función que realiza la poda. En este caso, vamos a podar aquellos nodos cuya penalización hasta el momento supere la alcanzada por una solución ya encontrada:

```
PROCEDURE EsAceptable(n:nodo):BOOLEAN;
BEGIN
   RETURN Valor(n)<=cota;
END EsAceptable;
```

Esta función hace uso de otra que es necesario implementar:

```
PROCEDURE Valor(n:nodo):CARDINAL;
BEGIN
   RETURN n^.coste;
END Valor;
```

que devuelve el coste acumulado hasta el momento. Esto tiene sentido pues el objetivo es encontrar la solución de menor coste.

Veamos ahora la función de coste para los nodos. Como nos piden encontrar la solución de menor coste, este valor es el más adecuado para tal función:

```
PROCEDURE h(n:nodo):CARDINAL;
BEGIN
   RETURN n^.coste;
END h;
```

Otra de las funciones que es necesario implementar es la que determina cuándo un nodo es solución. En nuestro caso consiste en decidir cuándo hemos sido capaces de acomodar hasta el *N*-ésimo vértice:

```
PROCEDURE EsSolucion(n:nodo):BOOLEAN;
BEGIN
   RETURN n^.k=N;
END EsSolucion;
```

En cuanto a la función *NoHaySolucion*, que devuelve un valor especial para indicar que el problema no admite solución, sabemos que para este problema eso no ocurrirá nunca si el grafo es conexo, pues siempre existe al menos una solución, que es la que conecta a todos los vértices entre sí.

Por su parte, la función *Eliminar* es la que va a devolver al sistema los recursos ocupados por un nodo, y es la que actúa como "destructor" del tipo abstracto de datos:

```
PROCEDURE Eliminar(VAR n:nodo);
BEGIN
   DISPOSE(n);
END Eliminar;
```

Con esto finaliza nuestra implementación del módulo "Nodos". El problema queda resuelto escogiendo la función del esquema que encuentra la mejor de todas las soluciones.

Para los valores iniciales del ejemplo, el algoritmo encuentra un recorrido óptimo de coste 29, que es el representado por el vector solución [1, 3, 5, 4, 2], obteniéndose los siguientes valores de exploración del árbol de expansión:

Núm. nodos generados	26
Núm. nodos analizados	12
Núm. nodos podados	14

Nos podemos plantear también lo que ocurriría si hubiésemos escogido una estrategia distinta de la LC, esto es, LIFO o FIFO. Siguiendo nuestro modelo de programación, bastaría con sustituir el módulo de implementación del tipo abstracto de datos "Estruc" acomodándolo a una pila o a una cola. Estos cambios permitirán recorrer el árbol de expansión en profundidad o en anchura, respectivamente.

	LC	LIFO	FIFO
Núm. nodos generados	26	36	64
Núm. nodos analizados	12	18	41
Núm. nodos podados	14	13	18

Como era de esperar por la función de coste definida para el problema, el mejor caso se obtiene cuando la estrategia de búsqueda es LC. Como veremos en otros ejemplos esto no es siempre así, pues para ciertos problemas no existen funciones de coste que permitan agilizar de forma notable la búsqueda por el árbol. De hecho, la búsqueda de buenas funciones de coste para reducir la exploración del árbol de expansión de un problema es una de las partes más delicadas e importantes de su resolución, sobre todo en aquellos casos en donde el árbol sea, por su tamaño, intratable.

7.5 EL LABERINTO

Este problema fue presentado en el apartado 6.6 del capítulo anterior, y consiste en determinar el camino de salida de un laberinto, representado por una matriz que indica las casillas transitables.

Solución (☺)

Cara a resolver este problema utilizando Ramificación y Poda, podemos definir una función LC basándonos en la distancia de Manhattan del punto en donde nos encontramos actualmente hasta la casilla de salida, es decir, el número mínimo estimado de movimientos para alcanzar la salida.

Teniendo en cuenta las consideraciones realizadas en la introducción de este capítulo, será suficiente realizar el módulo que implementa el tipo abstracto de datos que representa los nodos, pues el resto del programa es fijo.

Para ello, comenzaremos definiendo el tipo *nodo*. En primer lugar, deberá ser capaz no sólo de representar el laberinto y en dónde nos encontramos en el momento dado, sino que además deberá contener información sobre el recorrido realizado hasta tal punto. Utilizaremos por tanto las siguientes estructuras:

```
CONST dim = ...;     (* dimension del laberinto *)
TYPE laberinto = ARRAY[1..dim],[1..dim] OF CARDINAL;
TYPE nodo =   POINTER TO RECORD
                 x,y:CARDINAL;
                 l:laberinto
              END;
```

Las coordenadas *x* e *y* indican la casilla en donde nos encontramos, y los valores que vamos a almacenar en la matriz que define el laberinto indican el estado en el que se encuentra cada casilla, pudiendo ser:

a) 0 si la casilla no ha sido visitada,
b) *MAX(CARDINAL)* si la casilla no es transitable, o
c) un valor entre 1 y *dim*dim* que indica el orden en el que la casilla ha sido visitada.

De esta forma conseguimos que cada nodo sea "autónomo", esto es, que cada uno contenga toda la información relevante para poder realizar los procesos de bifurcación, la poda y la reconstrucción de la solución encontrada hasta ese momento. Necesitaremos además una variable global al módulo para almacenar la cota superior alcanzada por la mejor solución hasta el momento:

```
VAR cota:CARDINAL; (* num. movimientos de la mejor solucion *)
```

Esta variable será inicializada en el cuerpo del módulo "Nodos":

```
BEGIN (* Nodos *)
   cota:=MAX(CARDINAL);
END Nodos.
```

Respecto a las funciones de este módulo, en primer lugar la función *NodoInicial* habrá de contener la disposición inicial del laberinto:

```
CONST MURO = MAX(CARDINAL);

PROCEDURE NodoInicial():nodo;
   VAR n:nodo; i,j:CARDINAL;
BEGIN
   NEW(n);
   (* rellenamos a cero el laberinto *)
   FOR i:=1 TO dim DO FOR j:=1 TO dim DO n^.l[i,j]:=0 END END;
   (* situamos la casilla inicial *)
   n^.x:=1; n^.y:=1;
   n^.l[1,1]:=1;
   (* y ahora ponemos los bloques que forman los muros *)
   n^.l[1,5]:=MURO;n^.l[2,3]:=MURO; n^.l[3,2]:=MURO;
   n^.l[3,3]:=MURO;n^.l[3,5]:=MURO; n^.l[4,3]:=MURO;
   n^.l[4,5]:=MURO;n^.l[5,1]:=MURO; n^.l[5,3]:=MURO;
   n^.l[6,5]:=MURO;
   RETURN n;
END NodoInicial;
```

Siendo *MURO* una constante con el valor *MAX(CARDINAL)*. El laberinto representado por esa disposición es el siguiente:

1				X	
		X			

	X	X		X	
		X		X	
X		X			
				X	

La estrategia de ramificación está a cargo de la función *Expandir*. Cada nodo puede generar hasta cuatro hijos, que son los correspondientes a los posibles movimientos que podemos realizar desde una casilla dada (arriba, abajo, izquierda, derecha). Esta función sólo generará aquellos movimientos que sean válidos, esto es, que no se salgan del laberinto, no muevan sobre un muro, o bien sobre una casilla previamente visitada:

```
PROCEDURE Expandir(n:nodo;VAR hijos:ARRAY OF nodo):CARDINAL;
   VAR i,j,nhijos:CARDINAL; p:nodo;
BEGIN
    nhijos:=0;
    i:=n^.x;
    j:=n^.y;
    (* y ahora vemos a donde lo podemos "mover" *)
    IF ((i-1)>0) AND (n^.l[i-1,j]=0) THEN (* arriba *)
        INC(nhijos);
        Copiar(n,p);
        p^.l[i-1,j]:=p^.l[i,j]+1;
        DEC(p^.x);
        hijos[nhijos-1]:=p;
    END;
    IF ((j-1)>0) AND (n^.l[i,j-1]=0) THEN (* izquierda *)
        INC(nhijos);
        Copiar(n,p);
        p^.l[i,j-1]:=p^.l[i,j]+1;
        DEC(p^.y);
        hijos[nhijos-1]:=p;
    END;
    IF (i<dim) AND (n^.l[i+1,j]=0) THEN (* abajo *)
        INC(nhijos);
        Copiar(n,p);
        p^.l[i+1,j]:=p^.l[i,j]+1;
        INC(p^.x);
        hijos[nhijos-1]:=p;
    END;

    IF (j<dim) AND (n^.l[i,j+1]=0) THEN (* derecha *)
        INC(nhijos);
        Copiar(n,p);
```

```
        p^.l[i,j+1]:=p^.l[i,j]+1;
        INC(p^.y);
        hijos[nhijos-1]:=p;
    END;
    RETURN nhijos;
END Expandir;
```

Esta función hace uso de un procedimiento que permite duplicar un nodo:

```
PROCEDURE Copiar(VAR n1,n2:nodo);
   VAR i,j:CARDINAL;
BEGIN
   NEW(n2);
   FOR i:=1 TO dim DO FOR j:=1 TO dim DO
      n2^.l[i,j]:=n1^.l[i,j]
   END END;
   n2^.x:=n1^.x;
   n2^.y:=n1^.y;
END Copiar;
```

Una de las primeras dudas que nos asaltan tras implementar la función *Expandir* es si el orden en el que se bifurque va a influir en la eficiencia del algoritmo, tal como sucedía en algunos problemas de Vuelta Atrás. Realmente, el orden de ramificación sí es importante cuando el árbol de expansión se recorre siguiendo una estrategia "ciega" (FIFO o LIFO). Sin embargo, puesto que en este problema vamos a utilizar una estrategia LC, el orden en el que se generen los nodos (y se inserten en la estructura) no va a tener una influencia de peso en el comportamiento final del algoritmo.

Esto también lo hemos probado de forma experimental, y los resultados obtenidos muestran que el comportamiento del algoritmo no varía sustancialmente cuando se altera el orden en que se generan los nodos. En cualquier caso, esta afirmación es cierta para este problema pero no tiene por qué ser válida para cualquier otro: obsérvese que en este caso el número de hijos que genera cada nodo es pequeño (a lo más cuatro). Para problemas en los que el número de hijos que expande cada nodo es grande sí que puede tener influencia el orden de generación de los mismos.

Por otro lado, también es necesario implementar la función que realiza la poda. En este caso, vamos a podar aquellos nodos cuyo recorrido hasta el momento supere el número de pasos alcanzado por una solución ya encontrada:

```
PROCEDURE EsAceptable(n:nodo):BOOLEAN;
BEGIN
   RETURN Valor(n)<=cota;
END EsAceptable;
```

Esta función hace uso de otra que es necesario implementar:

```
PROCEDURE Valor(n:nodo):CARDINAL;
```

```
BEGIN
   RETURN n^.l[n^.x,n^.y];
END Valor;
```

que devuelve el número de pasos dados hasta el momento en el recorrido.

Veamos ahora la función de coste para los nodos. Tal como nos piden en el enunciado del problema, ésta corresponde a la distancia de Manhattan desde la posición en la que nos encontramos a la casilla final:

```
PROCEDURE h(n:nodo):CARDINAL;
BEGIN
   RETURN (dim-n^.x)+(dim-n^.y);
END h;
```

Otra de las funciones que es necesario implementar es la que determina cuándo un nodo es solución. En nuestro caso consiste en decidir cuándo hemos llegado a la casilla final, y para esto es suficiente comprobar que su función de coste vale cero:

```
PROCEDURE EsSolucion(n:nodo):BOOLEAN;
BEGIN
   RETURN h(n)=0
END EsSolucion;
```

También es necesario implementar la función *NoHaySolucion*, que devuelve un valor especial para indicar que el problema no admite solución:

```
PROCEDURE NoHaySolucion():nodo;
BEGIN
   RETURN NIL;
END NoHaySolucion;
```

Obsérvese que esto puede ocurrir para algunos laberintos, si es que los muros "rodean" completamente la salida. Por su parte, la función *Eliminar* es la que va a devolver al sistema los recursos ocupados por un nodo, y es la que actúa como "destructor" del tipo abstracto de datos:

```
PROCEDURE Eliminar(VAR n:nodo);
BEGIN
   DISPOSE(n);
END Eliminar;
```

Esto finaliza nuestra implementación del módulo "Nodos". El problema queda resuelto escogiendo la función del esquema apropiada según deseemos encontrar una solución, todas, o la mejor.

Para el valor inicial que damos en este ejemplo, los valores obtenidos por el programa son los que a continuación mostramos.

- En el caso de buscar solamente una solución, la primera que se encuentra consta de 13 movimientos y es la siguiente:

1				X	
2		X			
3	X	X		X	
4	5	X		X	
X	6	X	10	11	12
	7	8	9	X	13

Y los valores que se obtienen son:

Núm. nodos generados	17
Núm. nodos analizados	12
Núm. nodos podados	0

- En el caso de buscar la mejor solución, ésta consta de 11 movimientos y es la siguiente:

1	2	3	4	X	
		X	5		
	X	X	6	X	
		X	7	X	
X		X	8	9	10
				X	11

Y los valores que se obtienen en este caso son:

Núm. nodos generados	75
Núm. nodos analizados	62
Núm. nodos podados	11

- En el caso de buscar todas las soluciones, se consigue hallar 8 soluciones distintas, de longitudes 13, 19, 11, 11, 13, 13, 15 y 21 respectivamente, y los valores que se obtienen en este caso son:

Núm. nodos generados	166
Núm. nodos analizados	159
Núm. nodos podados	0

Nos podemos plantear también lo que ocurriría si hubiésemos escogido una estrategia distinta de la LC, esto es, LIFO o FIFO. Siguiendo nuestro modelo de programación, bastaría con sustituir el módulo de implementación del tipo abstracto de datos "Estruc" acomodándolo a una pila o a una cola. Estos cambios permitirán recorrer el árbol de expansión en profundidad o en anchura, respectivamente.

- En el caso de la estrategia LIFO los resultados que se obtienen no varían demasiado respecto a los conseguidos siguiendo nuestra estrategia LC:

	Primera	Mejor	Todas
Núm. nodos generados	15	69	166
Núm. nodos analizados	10	58	159
Núm. nodos podados	0	10	0

Como era de esperar, el valor de la columna "Todas" es igual, puesto que el árbol se rastrea completamente. Los valores de las otras dos columnas son similares a los obtenidos para la estrategia LC; el hecho que sean un poco mejores depende sólo del ejemplo concreto. Para otros ejemplos los valores que se obtienen siguiendo esta estrategia son peores (p.e. para aquellos laberintos con pocas casillas no transitables).

- En el caso de la estrategia FIFO los resultados son los siguientes:

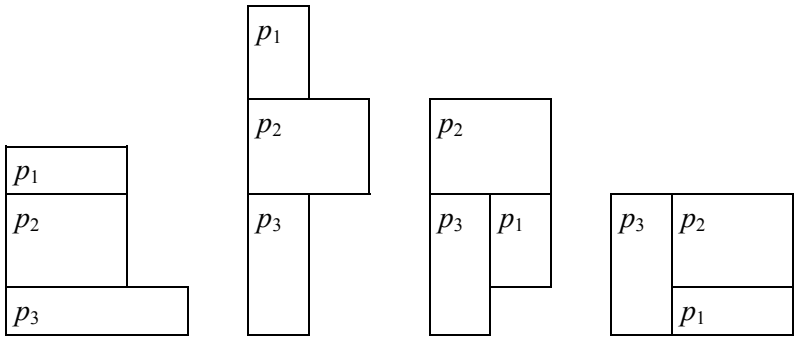
	Primera	Mejor	Todas
Núm. nodos generados	57	69	166
Núm. nodos analizados	48	58	159
Núm. nodos podados	0	10	0

Podemos observar que de nuevo la columna "Todas" consigue los mismos valores, y por la misma razón en este ejemplo, los valores de la columna "Mejor" no cambian. Sin embargo, vemos un empeoramiento notable de los valores en la columna "Primera". El motivo es obvio, pues al recorrer el árbol en anchura necesitamos generar muchos más nodos hasta llegar a la primera hoja solución.

7.6 LA COLOCACIÓN ÓPTIMA DE RECTÁNGULOS

Supongamos que disponemos de n piezas planas rectangulares p_1, p_2, ..., p_n, cada una con un área (a_i,b_i) $(1 \le i \le n)$, que precisamos encajar en un tablero plano rectangular T. El problema consiste en encontrar una disposición de las n piezas de forma que el tablero que necesitamos para contenerlas a todas sea de área mínima.

Por ejemplo, sean las piezas p_1=(1,2), p_2=(2,2) y p_3=(1,3). El siguiente diagrama muestra cuatro disposiciones distintas de las tres piezas:



Como puede observarse, el área de los rectángulos que los recubren en cada uno de los casos es 12 (4x3), 14 (7x2), 10 (5x2) y 9 (3x3).

Solución (☞)

Este problema plantea dos dificultades principales. En primer lugar la de cómo generar el árbol de expansión pues, como veremos más adelante, las formas usuales de planteamiento de cualquier problema de Ramificación y Poda no valen para este caso.

La segunda dificultad es un problema de recursos, pues el árbol que se maneja es muy grande, y por tanto el número de nodos que se genera supera pronto la capacidad del ordenador. Incluso para el ejemplo del enunciado, con sólo tres piezas pequeñas, algunas estrategias agotan enseguida la memoria disponible.

Comenzaremos analizando la construcción del árbol de expansión para el problema. En primer lugar hemos de plantear la solución como una secuencia de decisiones, una en cada paso o etapa. La primera idea que intentamos llevar a cabo es la de ir colocando una pieza en cada paso. Así en la etapa k colocaremos la pieza p_k $(1 \le k \le n)$ adyacente a las que ya tenemos, y para cada una de ellas será suficiente almacenar la posición en donde la hemos colocado. Sin embargo, esta estrategia no es válida puesto que al ir colocando las piezas por orden y cada una junto a las que ya teníamos colocadas, no cubrimos todas las posibilidades. Por ejemplo, de esta forma no podríamos tener la pieza número uno junto a la tercera, y detrás de ésta la segunda. Por este motivo, en cada paso necesitamos explorar todas las piezas aún no colocadas, y no la pieza p_k en concreto.

Por otro lado, para cada una de estas piezas tenemos múltiples opciones, pues podemos colocarlas vertical u horizontalmente (a menos que la pieza sea simétrica) y alrededor del conjunto de piezas que ya tenemos situadas.

En cuanto a nuestra representación de la solución, la forma más sencilla es la de disponer de un tablero en donde marcar las casillas ocupadas. El tablero puede ser implementado mediante una matriz de número naturales, en donde el valor 0 indica una posición libre y un valor $k > 0$ indica que esa posición está ocupada por la pieza p_k.

Con esto en mente, ya podemos atacar la implementación del tipo abstracto de datos que representa los nodos.

```
CONST N = ...; (* numero de piezas *)
CONST LMAX = ...; (* longitud maxima de una pieza (alto o ancho) *)
CONST XMAX = N*LMAX; YMAX=N*LMAX; (* tam. maximo del tablero *)

TYPE tablero = ARRAY[0..XMAX],[0..YMAX] OF CARDINAL;
TYPE nodo = POINTER TO RECORD
      t:tablero; (* tablero asociado al nodo *)
      k:CARDINAL;  (* nivel *)
      xmax,ymax:CARDINAL; (* area acumulada *)
      puestas:ARRAY [1..N] OF BOOLEAN;  (* piezas ya colocadas *)
    END;
```

Además del tablero con la solución construida hasta ese momento y el nivel, las otras componentes del registro indican el área del rectángulo que contiene a las piezas colocadas y un vector que indica qué piezas están ya situadas y cuáles quedan por colocar.

Necesitaremos además dos variables globales al módulo para almacenar la cota superior (el área) alcanzada por la mejor solución hasta el momento y el área de las piezas que debemos colocar:

```
VAR cota:CARDINAL;
VAR piezas:ARRAY[1..N] OF RECORD x,y:CARDINAL END;
```

Estas variables serán inicializadas en el cuerpo principal del módulo “Nodos”:

```
BEGIN (* Nodos *)
    cota:=MAX(CARDINAL);
    piezas[1].x:=1; piezas[1].y:=2;
    piezas[2].x:=2; piezas[2].y:=2;
    piezas[3].x:=1; piezas[3].y:=3;
END Nodos.
```

Veamos ahora las funciones de este módulo. En primer lugar la función *NodoInicial* habrá de contener el nodo raíz del árbol de expansión:

```
PROCEDURE NodoInicial():nodo;
  VAR n:nodo; i,j:CARDINAL;
BEGIN
  NEW(n); n^.k:=0; n^.xmax:=0; n^.ymax:=0; (* origen *)
  FOR i:=1 TO N DO n^.puestas[i]:=FALSE END;
  FOR i:=0 TO XMAX DO FOR j:=0 TO YMAX DO n^.t[i,j]:=0 END END;
  RETURN n;
END NodoInicial;
```

Como podemos observar, inicialmente el tablero se encuentra vacío. Por otro lado, la estrategia de ramificación está a cargo de la función *Expandir*. Cada nodo va a generar un hijo por cada posición posible de cada una de las piezas aún no incluidas en el tablero. Este hecho es el que produce un árbol de expansión tan grande:

```
PROCEDURE Expandir(n:nodo;VAR hijos:ARRAY OF nodo):CARDINAL;
   VAR  i,nhijos:CARDINAL; p:nodo;
        inicial,basura:BOOLEAN; a,b:CARDINAL;
BEGIN
 nhijos:=0;
 inicial:=(n^.xmax=0)AND(n^.ymax=0); (* esta vacio? *)
 FOR i:=1 TO N DO  (* generamos los hijos *)
  IF NOT n^.puestas[i] THEN
   FOR a:=0 TO n^.xmax+1 DO
    FOR b:=0 TO n^.ymax+1 DO
     Copiar(n,p);
     IF ColocarPieza(inicial,p,i,a,b,piezas[i].x,piezas[i].y) THEN
      INC(nhijos); INC(p^.k); hijos[nhijos-1]:=p;
     ELSE Eliminar(p);
     END;
     IF piezas[i].x<>piezas[i].y THEN (* no simetrica *)
      Copiar(n,p);
      IF ColocarPieza(inicial,p,i,a,b,piezas[i].y,piezas[i].x)THEN
       INC(nhijos); INC(p^.k); hijos[nhijos-1]:=p;
      ELSE Eliminar(p);
      END
     END
    END
   END
  END;
 RETURN nhijos;
END Expandir;
```

Esta función hace uso de un procedimiento que permite duplicar un nodo:

```
  PROCEDURE Copiar(VAR n1,n2:nodo);
```

```
BEGIN
    NEW(n2);
    n2^.xmax:=n1^.xmax; n2^.ymax:=n1^.ymax;
    n2^.t:=n1^.t; n2^.puestas:=n1^.puestas;
    n2^.k:=n1^.k;
END Copiar;
```

Y también de otra función para determinar si una pieza puede ser colocada o no en una determinada posición:

```
PROCEDURE ColocarPieza (inicial:BOOLEAN; (* primera pieza?*)
                        VAR n:nodo;      (* nodo vivo *)
                        p:CARDINAL;      (* num. pieza a poner *)
                        x,y:CARDINAL;    (* donde ponerla *)
                        a,b:CARDINAL     (* largo y alto *)
                        ):BOOLEAN;       (* puedo ponerla? *)
   VAR i,j:CARDINAL; conexa:BOOLEAN;
BEGIN
   IF (inicial)AND((x<>0)OR(y<>0)) THEN RETURN FALSE END;
   (* primero miramos que cabe *)
   IF ((x+a-1)>XMAX)OR((y+b-1)>YMAX) THEN RETURN FALSE END;
   (* despues miramos que no pisa a ninguna pieza *)
   FOR i:=x TO x+a-1 DO FOR j:=y TO y+b-1 DO
      IF n^.t[i,j]<>0 THEN RETURN FALSE END
   END END;
   (* despues miramos que sea adyacente a otra *)
   IF NOT inicial THEN
      conexa:=FALSE;
      IF x=0 THEN i:=0 ELSE i:=x-1 END;
      WHILE (i<=Max2(x+a,XMAX)) AND (NOT conexa) DO
         IF (((y>0)AND(n^.t[i,y-1]<>0))OR
            ((y+b<=YMAX)AND(n^.t[i,y+b]<>0))) THEN conexa:=TRUE END;
         INC(i)
      END;
      IF y=0 THEN j:=0 ELSE j:=y-1 END;
      WHILE (j<=Max2(y+b,YMAX)) AND (NOT conexa) DO
         IF (((x>0)AND(n^.t[x-1,j]<>0))OR
            ((x+a<=XMAX)AND(n^.t[x+a,j]<>0))) THEN conexa:=TRUE END;
         INC(j)
      END;
      IF NOT conexa THEN RETURN FALSE END;
   END;

   (* ahora la ponemos en el tablero *)
   FOR i:=x TO x+a-1 DO FOR j:=y TO y+b-1 DO
      n^.t[i,j]:=p
```

```
    END END;
    n^.puestas[p]:=TRUE;
    (* y ajustamos los nuevos bordes del tablero *)
    n^.xmax:=Max2(x+a-1,n^.xmax);
    n^.ymax:=Max2(y+b-1,n^.ymax);
    RETURN TRUE;
END ColocarPieza;
```

La dificultad de este problema reside en las funciones *Expandir* y *ColocarPieza*. Como podemos ver, la primera de ellas genera los nodos hijos de un nodo dado, y recorre el tablero buscando posiciones en donde situar cada una de las piezas aún no colocadas. Por cada pieza puede realizar hasta dos veces esta tarea, según disponga la pieza vertical u horizontalmente. Por su parte, la segunda función es la que decide si una posición es válida o no para colocar una pieza. Entendemos por válida que quepa en el tablero, no "pise" a ninguna otra, y sea adyacente a alguna de las piezas previamente colocadas.

También es necesario implementar la función que realiza la poda. En este caso, vamos a podar aquellos nodos cuya área hasta el momento supere la alcanzada por una solución ya encontrada. Para eso definimos una función de coste para los nodos. Como buscamos la solución de menor área total, el valor que vamos a tomar es el del área acumulada hasta el momento:

```
PROCEDURE h(n:nodo):CARDINAL;
BEGIN
    RETURN Max2((n^.xmax+1)*(n^.ymax+1),AreaTotalPiezas);
END h;
```

donde *AreaTotalPiezas* es el área de todas la piezas, en este caso 9, que es el mejor de los casos posibles.

De las dos funciones siguientes, la primera calcula el valor asociado a una solución, y la segunda es la que va a permitir realizar la poda:

```
PROCEDURE Valor(n:nodo):CARDINAL;
BEGIN
   RETURN (n^.xmax+1)*(n^.ymax+1);
END Valor;

PROCEDURE EsAceptable(n:nodo):BOOLEAN;
BEGIN
   RETURN Valor(n)<=cota;
END EsAceptable;
```

Otra de las funciones que es necesario implementar es la que determina cuándo un nodo es solución. En nuestro caso consiste en decidir cuándo hemos sido capaces de acomodar todas las piezas. Como en cada paso colocamos una, llegaremos a una hoja cuando el nivel del nodo sea *N*:

```
PROCEDURE EsSolucion(n:nodo):BOOLEAN;
BEGIN
   RETURN n^.k=N;
END EsSolucion;
```

En cuanto a la función *NoHaySolucion*, que devuelve un valor especial para indicar que el problema no admite solución, sabemos que para este problema eso no ocurrirá nunca, pues estamos suponiendo que el tablero es lo suficientemente grande para acomodar a todas la piezas.

Por su parte, la función *Eliminar* es la que va a devolver al sistema los recursos ocupados por un nodo, y es la que actúa como "destructor" del tipo abstracto de datos:

```
PROCEDURE Eliminar(VAR n:nodo);
BEGIN
   DISPOSE(n);
END Eliminar;
```

Con esto finaliza nuestra implementación del módulo "Nodos". El problema queda resuelto escogiendo la función del esquema que encuentra la mejor de las soluciones.

Para los valores iniciales dados en el ejemplo, el algoritmo encuentra una disposición óptima de coste 9, que es una de las indicadas en el enunciado del problema. Los valores del árbol de expansión que se obtienen son:

Núm. nodos generados	1081
Núm. nodos analizados	80
Núm. nodos podados	982

Podemos plantearnos también lo que ocurriría si hubiésemos escogido una estrategia distinta de la LC, esto es, LIFO o FIFO. Siguiendo nuestro modelo de programación, bastaría con sustituir el módulo de implementación del tipo abstracto de datos "Estruc" acomodándolo a una pila o a una cola. Estos cambios permitirán recorrer el árbol de expansión en profundidad o en anchura, respectivamente.

	LC	LIFO	FIFO
Núm. nodos generados	1081	681	1081
Núm. nodos analizados	80	59	80

Núm. nodos podados	982	596	982

Como puede observarse, para los valores del ejemplo trabaja mejor la estrategia LIFO, debido también al orden en el que se va generando el árbol de expansión. Éste es un buen ejemplo en donde la función de coste que hemos utilizado para implementar la estrategia LC no da buenos frutos.

También conviene destacar que éste es un ejemplo en donde la poda realiza una gran labor, pero se trata de la poda "a posteriori". Y hemos de señalar que aunque el número de nodos generados sea grande, el trabajo real del algoritmo, que viene dado por el número de nodos analizados, no es excesivo pese al gran número de nodos que se generan.

7.7 LA MOCHILA (0,1)

Recordemos el problema de la Mochila (0,1), enunciado por primera vez en el capítulo 4. Dados n elementos e_1, e_2, ..., e_n con pesos p_1, p_2, ..., p_n y beneficios b_1, b_2, ..., b_n, y dada una mochila capaz de albergar hasta un máximo de peso M (capacidad de la mochila), queremos encontrar cuáles de los n elementos hemos de introducir en la mochila de forma que la suma de los beneficios de los elementos escogidos sea máxima.

Esto es, hay que encontrar valores (x_1, x_2, ..., x_n), donde cada x_i puede ser 0 ó 1, de forma que se maximice el beneficio, dado por la cantidad $\sum_{i=1}^{n} b_i x_i$, sujeta a la restricción $\sum_{i=1}^{n} p_i x_i \leq M$.

En este caso nos planteamos resolver el problema utilizando una estrategia LC.

Solución (😐)

Para construir el árbol de expansión del problema es necesario plantear la solución como una secuencia de decisiones, una en cada etapa o nivel del árbol. Para ello, vamos a representar la solución del problema mediante un vector, en donde en cada posición podrá encontrarse uno de los valores 1 ó 0, indicando si introducimos o no el elemento en cuestión.

Así, comenzando por el primer elemento, iremos recorriéndolos todos y decidiendo en cada paso si incluimos o no el elemento, por lo que cada nodo va a dar lugar a lo sumo a dos hijos. Sin pérdida de generalidad vamos a suponer los elementos ordenados de forma decreciente en cuanto a su ratio beneficio/peso para facilitar el cálculo de la función de coste, tal y como veremos más adelante.

Respecto a la poda, éste es un problema de maximización, mientras que el esquema visto en la introducción del capítulo (*RyP_lamejor()*) está diseñado para problemas de minimización. Pero el cambio es bien sencillo, pues basta con considerar la naturaleza dual de ambos problemas y utilizar el hecho de que para maximizar una función positiva v basta con minimizar la función $v' = -v$.

Comencemos entonces a definir la estructura de datos que contendrá a los nodos. En ellos se ha de almacenar información sobre la solución alcanzada hasta ese momento, y por tanto definimos:

```
CONST N = ...; (* numero de elementos distintos *);
TYPE solucion = ARRAY[1..N] OF CARDINAL;
TYPE nodo = POINTER TO RECORD
                          peso, (* peso acumulado *)
                          beneficio, (* beneficio acumulado *)
                          k:CARDINAL; (* nivel *)
                          s:solucion
                       END;
```

Necesitamos además tres variables globales al módulo "Nodos": una para almacenar la cota superior alcanzada hasta el momento, otra con la capacidad máxima de la mochila, y otra para guardar la tabla con los datos iniciales del problema. Obsérvese que esta tabla es global pues contiene la información sobre los propios elementos.

```
VAR cota:CARDINAL;
VAR capacidad:CARDINAL;
VAR tabla: ARRAY [1..N] OF RECORD beneficio,peso:CARDINAL END;
```

Estas variables serán inicializadas en el cuerpo principal del módulo "Nodos":

```
BEGIN (* Nodos *)
   cota:=MAX(CARDINAL);
   capacidad:=8;
   (* ordenados de forma decreciente por ratio beneficio/peso *)
   tabla[1].beneficio:=10;
   tabla[1].peso:=5;
   tabla[2].beneficio:=5;
   tabla[2].peso:=3;
   tabla[3].beneficio:=6;
   tabla[3].peso:=4;
   tabla[4].beneficio:=3;
   tabla[4].peso:=2;
END Nodos.
```

La función *NodoInicial* ha de generar un nodo vacío inicialmente:

```
PROCEDURE NodoInicial():nodo;
   VAR n:nodo; i:CARDINAL;
BEGIN
    NEW(n);
    FOR i:=1 TO N DO n^.s[i]:=0 END;
```

```
    n^.peso:=0;
    n^.beneficio:=0;
    n^.k:=0;
    RETURN n;
END NodoInicial;
```

La estrategia de ramificación está a cargo de la función *Expandir*. Cada nodo puede generar a lo sumo dos hijos, que corresponden a incluir el elemento o no en la mochila. Sólo serán generados aquellos nodos que sean válidos, esto es, si caben en la mochila, teniendo en cuenta la capacidad utilizada hasta el momento.

```
PROCEDURE Expandir(n:nodo;VAR hijos:ARRAY OF nodo):CARDINAL;
    VAR i,j,peso,plazo,beneficio,nhijos:CARDINAL; p:nodo;
BEGIN
    nhijos:=0;
    i:=n^.k+1;
    IF i>N THEN RETURN nhijos END; (* caso especial *)
    peso:=tabla[i].peso;
    beneficio:=tabla[i].beneficio;
    (* caso 0: no lo metemos *)
    INC(nhijos);
    Copiar(n,p);
    INC(p^.k); (* no se aumenta el peso ni el beneficio *)
    hijos[nhijos-1]:=p;
    (* caso 1: lo metemos *)
    IF n^.peso+peso<=capacidad THEN (* cabe! *)
        INC(nhijos);
        Copiar(n,p);
        p^.s[i]:=1;
        INC(p^.k);
        INC(p^.peso,peso);
        INC(p^.beneficio,beneficio);
        hijos[nhijos-1]:=p;
    END;
    RETURN nhijos;
END Expandir;
```

Esta función hace uso de un procedimiento que permite duplicar un nodo:

```
PROCEDURE Copiar(VAR n1,n2:nodo);
    VAR i:CARDINAL;
BEGIN
    NEW(n2);
    FOR i:=1 TO N DO n2^.s[i]:=n1^.s[i] END;
```

```
    n2^.peso:=n1^.peso;
    n2^.beneficio:=n1^.beneficio;
    n2^.k:=n1^.k;
END Copiar;
```

Por otro lado, es necesario implementar la función que realiza la poda. Comenzaremos primero definiendo la función de coste que vamos a asignar a cada nodo. Para ello vamos a considerar que los elementos iniciales están todos ordenados de forma decreciente por su ratio beneficio/peso. Según esto, cuando nos encontramos en el paso *k*-ésimo disponemos de un beneficio acumulado B_k. Por la forma en como hemos ido construyendo el vector, sabemos que:

$$B_k = \sum_{i=1}^{k} s[i] * tabla[i].beneficio \, .$$

Para calcular el valor máximo que podríamos alcanzar con ese nodo (B_M) procederemos de igual forma a como hicimos en la resolución de este problema utilizando la técnica de Vuelta Atrás (apartado 6.8). Así, vamos a suponer que rellenáramos el resto de la mochila con el mejor de los elementos que nos quedan por analizar. Como los tenemos dispuestos en orden decreciente de ratio beneficio/peso, éste mejor elemento será el siguiente (k+1). Este valor, aunque no tiene por qué ser alcanzable, nos permite dar una cota superior del valor al que podemos "aspirar" si seguimos por esa rama del árbol:

$$B_M = B_k + \left(capacidad - \sum_{i=1}^{k} s[i] * tabla[i].peso \right) \frac{tabla[k+1].beneficio}{tabla[k+1].peso}$$

Esto da lugar a la siguiente función de coste para un nodo dado:

```
PROCEDURE h(n:nodo):CARDINAL;
    VAR mejor:CARDINAL;
BEGIN
    IF EsSolucion(n) THEN RETURN n^.beneficio END;
    mejor:=CARDINAL((REAL(tabla[n^.k+1].beneficio)/
                        REAL(tabla[n^.k+1].peso))+0.5);
    RETURN n^.beneficio+(capacidad-n^.peso)*mejor
END h;
```

Obsérvese el carácter dual del problema de la mochila frente a los que hemos visto con anterioridad. Frente a un problema de minimización como teníamos en los anteriores, aquí nos planteamos la maximización del beneficio conseguido.

En general, los problemas de maximización de una función v se consiguen minimizando la función $-v$. Sin embargo, como es necesario trabajar con números positivos, utilizaremos el hecho de que dada una constante positiva t, el problema de minimizar una función f coincide con el de minimizar la función $f+t$. Uniendo ambas consideraciones, para maximizar nuestra función original v trataremos de

minimizar *MAX(CARDINAL)–v*, que es una función no negativa. Esto hace que definamos la función *Valor* como:

```
PROCEDURE Valor(n:nodo):CARDINAL;
BEGIN
    RETURN MAX(CARDINAL)-h(n);
END Valor;
```

De esta forma podremos podar, al igual que hacíamos en los otros problemas, aquellos nodos cuya penalización hasta el momento supere la alcanzada por una solución ya encontrada:

```
PROCEDURE EsAceptable(n:nodo):BOOLEAN;
BEGIN
   RETURN Valor(n)<=cota;
END EsAceptable;
```

Otra de las funciones que es necesario implementar es la que determina cuándo un nodo es solución. En nuestro caso consiste en decidir cuándo hemos sido capaces de tratar hasta el *N*-ésimo elemento:

```
PROCEDURE EsSolucion(n:nodo):BOOLEAN;
BEGIN
   RETURN n^.k=N;
END EsSolucion;
```

En cuanto a la función *NoHaySolucion*, que devuelve un valor especial para indicar que el problema no admite solución, sabemos que para este problema eso no ocurrirá nunca, pues siempre existe al menos una solución, que es la que representa el vector [0,0,...,0], es decir, siempre podemos no incluir ningún elemento. Ésta es, por ejemplo, la solución a un problema en donde los pesos de los elementos superen la capacidad de la mochila.

Por su parte, la función *Eliminar* es la que va a devolver al sistema los recursos ocupados por un nodo, y es la que actúa como "destructor" del tipo abstracto de datos:

```
PROCEDURE Eliminar(VAR n:nodo);
BEGIN
   DISPOSE(n);
END Eliminar;
```

Con esto finaliza nuestra implementación del módulo "Nodos". El problema queda resuelto escogiendo la función del esquema que encuentra la mejor de todas las soluciones.

7.8 LA MOCHILA (0,1) CON MÚLTIPLES ELEMENTOS

El problema de la Mochila (0,1) con múltiples elementos fue presentado en el apartado 5.17, y es una variación del problema de la Mochila (0,1) en donde en vez de tener n objetos distintos, de lo que disponemos es de n tipos de objetos. En definitiva, se trata de cambiar la restricción de que los números x_i sólo puedan tomar los valores 0 ó 1 por la de que sean enteros no negativos.

Nos piden dar una solución a este problema utilizando Ramificación y Poda, diseñando una función de coste adecuada.

Por otro lado, existe una variación del problema en donde se incorpora la restricción de que existe sólo un número limitado de objetos de cada tipo. Sería interesante modificar el algoritmo anterior para tener en cuenta esta restricción.

Solución (☺)

Este problema está muy ligado al anterior y va a presentar muy pocas diferencias frente a él. En primer lugar, la solución va a seguir estando representada por un vector, pero esta vez no será de ceros y unos, sino que podrá tomar valores enteros positivos. Y en segundo lugar, cada nodo no generará a lo sumo dos hijos, sino que podrá generar varios, tantos como le permita la capacidad de la mochila.

El primer cambio no se ve reflejado en el algoritmo desarrollado en el problema anterior, pues el tipo *nodo* ya permitía almacenar valores positivos mayores que uno. El segundo cambio tiene su reflejo en la función que expande los nodos:

```
PROCEDURE Expandir(n:nodo;VAR hijos:ARRAY OF nodo):CARDINAL;
    VAR i,j,peso,plazo,beneficio,nhijos:CARDINAL;
    p:nodo;
BEGIN
    nhijos:=0;
    (* en cada etapa generamos los nodos hijos *)
    i:=n^.k+1;
    IF i>N THEN RETURN nhijos END; (* caso especial *)
    peso:=tabla[i].peso;
    beneficio:=tabla[i].beneficio;
    (* caso 0: no lo metemos *)
    INC(nhijos);
    Copiar(n,p);
    INC(p^.k); (* no se aumenta el peso ni el beneficio *)
    hijos[nhijos-1]:=p;


    (* resto de los casos: metemos 1, 2, ... unidades *)
    j:=1;
    WHILE n^.peso+(peso*j)<=capacidad DO (* caben j unidades *)
        INC(nhijos);
        Copiar(n,p);
        p^.s[i]:=j;
```

```
            INC(p^.k);
            INC(p^.peso,peso*j);
            INC(p^.beneficio,beneficio*j);
            hijos[nhijos-1]:=p;
            INC(j)
        END;
        RETURN nhijos;
    END Expandir;
```

Las demás funciones del módulo "Nodos" quedan igual.

Respecto a la modificación de limitar el número de objetos de un tipo, en primer lugar necesitamos modificar la estructura de datos que almacena los datos globales sobre los elementos, para incluir la información sobre el número de objetos que disponemos de cada tipo:

```
VAR tabla:ARRAY[1..N]OF RECORD beneficio,peso,unidades:CARDINAL END;
```

y, por supuesto, incluir la inicialización de tales datos en el proceso de inicialización del módulo "Nodos":

```
tabla[1].unidades:=2; tabla[2].unidades:=2;
tabla[3].unidades:=2; tabla[4].unidades:=2;
```

Por otro lado, en la función *Expandir* hace falta tener en cuenta esta limitación:

```
PROCEDURE Expandir(n:nodo;VAR hijos:ARRAY OF nodo):CARDINAL;
    VAR i,j,peso,plazo,beneficio,nhijos:CARDINAL; p:nodo;
BEGIN
    nhijos:=0;
    (* en cada etapa generamos los posibles nodos hijos *)
    i:=n^.k+1;
    IF i>N THEN RETURN nhijos END; (* caso especial *)
    peso:=tabla[i].peso;
    beneficio:=tabla[i].beneficio;
    (* caso 0: no lo metemos *)
    INC(nhijos);
    Copiar(n,p);
    INC(p^.k); (* no se aumenta el peso ni el beneficio *)
    hijos[nhijos-1]:=p;


    (* resto de los casos: metemos 1, 2, ... unidades *)
    j:=1;
    WHILE (n^.peso+(peso*j)<=capacidad)AND(j<=tabla[i].unidades) DO
        (* caben j unidades *)
        INC(nhijos);
        Copiar(n,p);
```

```
            p^.s[i]:=j;
            INC(p^.k);
            INC(p^.peso,peso*j);
            INC(p^.beneficio,beneficio*j);
            hijos[nhijos-1]:=p;
            INC(j)
        END;
        RETURN nhijos;
    END Expandir;
```

Si nos fijamos, la única diferencia entre esta función y la del apartado anterior es la condición del bucle que va generando los hijos. Ahora se pregunta no sólo si cabría un nuevo elemento de ese tipo, sino además si disponemos de él.

7.9 LA ASIGNACIÓN DE TAREAS

El problema de la asignación de tareas puede resolverse también utilizando una técnica de Ramificación y Poda. Recordemos que este problema consiste en, dadas *n* personas y *n* tareas, asignar a cada persona una tarea minimizando el coste de la asignación total, haciendo uso de una matriz de tarifas que determina el coste de asignar a cada persona una tarea.

Deseamos implementar dicho algoritmo utilizando la técnica de Ramificación y Poda y resolver el problema de minimizar el coste total para las dos siguientes matrices de tarifas, en donde las letras representan personas y los números tareas:

	1	2	3	4
a	94	1	54	68
b	74	10	88	82
c	62	88	8	76
d	11	74	81	21

	1	2	3	4	5
a	11	17	8	16	20
b	9	7	12	6	15
c	13	16	15	12	16
d	21	24	17	28	26
e	14	10	12	11	15

Solución (😐)

En primer lugar hemos de construir el árbol de expansión del problema, y para ello es necesario plantear la solución como una secuencia de decisiones, una en cada etapa o nivel del árbol. Una forma fácil de realizar esto es considerando la estructura que va a tener la solución del problema.

En este caso la solución puede ser representada mediante un vector, cuyo *k*-ésimo elemento indica la tarea asignada a la persona *k*. Así, comenzando por la primera persona, en cada paso decidiremos qué tarea le asignamos de entre las que no hayan sido asignadas todavía, lo que implica que cada nodo generará a lo sumo *N–k* nodos hijos.

Teniendo en cuenta las consideraciones realizadas en la introducción de este capítulo, será suficiente realizar el módulo que implementa el tipo abstracto de

datos que representa los nodos, pues el resto del programa es fijo para este tipo de algoritmos.

Respecto a la información que debe contener cada uno de los nodos, hemos de conseguir que cada uno de ellos sea "autónomo", esto es, que contenga toda la información relevante para poder realizar los procesos de bifurcación, poda y reconstrucción de la solución encontrada hasta ese momento. En consecuencia, al menos ha de contar con el nivel en donde se encuentra y con el vector solución construido hasta ese instante. Por otro lado, también debe contener la información que permita realizar la poda. En este sentido vamos a incluir una matriz de tarifas modificada, en donde vamos a ir anulando las opciones que dejan de tener sentido en cada paso. Por ejemplo, si asignamos la tarea 3 a la persona a, ya no tiene sentido asignar la tarea 3 a nadie más.

Por otro lado necesitamos una función de coste LC para podar nodos. Por tratarse de un problema de minimización, dicha función va a representar una cota inferior (teórica, y por lo tanto no necesariamente alcanzable) de la solución del problema. Para ello, calcularemos los mínimos de los elementos de cada columna aún no asignados, puesto que éstas son las mejores tarifas que vamos a poder tener para cada tarea, independientemente de a quien se las asignemos. De hecho, ésta es una cota no necesariamente alcanzable, pues no estamos imponiendo la restricción de que no se puedan repetir trabajadores.

Con todo esto, comenzaremos definiendo el tipo *nodo* que utilizaremos en la implementación del algoritmo de Ramificación y Poda que resuelve el problema. Vamos a utilizar entonces la siguiente estructura de datos:

```
CONST N = ...; (* numero de personas y tareas *)
TYPE solucion = ARRAY[1..N] OF CARDINAL;
TYPE tarifas  = ARRAY[1..N],[1..N] OF CARDINAL;
TYPE nodo = POINTER TO RECORD
                         matriz:tarifas; (* matriz de tarifas *)
                         k:CARDINAL; (* nivel *)
                         s:solucion
                       END;
```

Además del vector solución y el nivel, la otra componente del registro es una matriz de tarifas, pero modificada para reflejar el hecho de que ya hay ciertas tareas asignadas. La forma de reflejar esta circunstancia es mediante la asignación de una valor ∞ a las tarifas de aquellas tareas que no puedan ser asignadas.

Necesitaremos además una variable *cota* global al módulo para almacenar la cota superior alcanzada por la mejor solución hasta el momento. Esta variable será inicializada en el cuerpo principal del módulo "Nodos":

```
BEGIN (* Nodos *)
   cota:=MAX(CARDINAL);
END Nodos.
```

Veamos ahora las funciones de este módulo. En primer lugar la función *NodoInicial* habrá de contener el nodo raíz del árbol de expansión:

```
PROCEDURE NodoInicial():nodo; (* para el ejemplo 1 *)
   VAR n:nodo; i,j:CARDINAL;
BEGIN
   NEW(n);
   FOR i:=1 TO N DO n^.s[i]:=0 END;
   n^.k:=0;
   n^.matriz[1,1]:=94;  n^.matriz[1,2]:=1;   n^.matriz [1,3]:=54;
   n^.matriz[1,4]:=68;  n^.matriz[2,1]:=74;  n^.matriz[2,2]:=10;
   n^.matriz[2,3]:=88;  n^.matriz [2,4]:=82; n^.matriz [3,1]:=62;
   n^.matriz [3,2]:=88; n^.matriz [3,3]:=8 ; n^.matriz [3,4]:=76;
   n^.matriz [4,1]:=11; n^.matriz [4,2]:=74; n^.matriz [4,3]:=81;
   n^.matriz [4,4]:=21;
   RETURN n;
END NodoInicial;
```

Como podemos observar, se asocia la matriz original al nodo origen. Por otro lado, la estrategia de ramificación está a cargo de la función *Expandir*. Cada nodo puede generar, como hemos dicho antes, a lo sumo *N–k* hijos, que son los correspondientes a los nodos aún no incluidos en el recorrido:

```
PROCEDURE Expandir(n:nodo;VAR hijos:ARRAY OF nodo):CARDINAL;
   VAR nk,i,j,l,coste,nhijos:CARDINAL; p:nodo;
BEGIN
   nhijos:=0;
   nk:=n^.k+1;
   i:=n^.s[nk-1];
   IF nk>N THEN RETURN nhijos END; (* caso especial *)
   FOR j:=1 TO N DO
      IF NoEsta(n^.s,nk-1,j) THEN
         INC(nhijos);
         Copiar(n,p);
         p^.s[nk]:=j;
         Quitar(p^.matriz,nk,j);
         INC(p^.k);
         hijos[nhijos-1]:=p;
      END
   END;
   RETURN nhijos;
END Expandir;
```

Esta función hace uso de varios procedimientos que a continuación veremos. El primero de ellos permite duplicar un nodo:

```
PROCEDURE Copiar(VAR n1,n2:nodo);
   VAR i,j:CARDINAL;
BEGIN
```

```
   NEW(n2);
   n2^.s:=n1^.s; n2^.matriz:=n1^.matriz; n2^.k:=n1^.k;
END Copiar;
```

También utiliza otra función para determinar si una tarea está incluida o no en la solución:

```
PROCEDURE NoEsta(s:solucion;k,j:CARDINAL):BOOLEAN;
   VAR i:CARDINAL;
BEGIN
   FOR i:=1 TO k DO
      IF s[i]=j THEN RETURN FALSE END
   END;
   RETURN TRUE;
END NoEsta;
```

Aparte de estas dos funciones, también necesita modificar la matriz de tarifas de un nodo, eliminando las opciones que ya no son válidas. Esto lo realiza mediante el siguiente procedimiento:

```
PROCEDURE Quitar(VAR m:tarifas;i,j:CARDINAL);
    VAR k,temp:CARDINAL;
BEGIN
    temp:=m[i,j]; (* lo guardamos para reponerlo despues *)
    FOR k:=1 TO N DO
        m[i,k]:=MAX(CARDINAL); m[k,j]:=MAX(CARDINAL);
    END;
    m[i,j]:=temp;
END Quitar;
```

Además es necesario implementar la función que realiza la poda. En este caso vamos a implementar una función que asigne un coste a un nodo:

```
PROCEDURE CosteCol(VAR m:tarifas;j:CARDINAL):CARDINAL;
   VAR i,c:CARDINAL;
BEGIN (* calcula el elemento minimo de una columna dada *)
   c:=m[1,j];
   FOR i:=2 TO N DO IF m[i,j]<c THEN c:=m[i,j] END END;
   RETURN c
END CosteCol;
PROCEDURE Coste(VAR m:tarifas):CARDINAL;
(* calcula la suma de los minimos de las columnas *)
   VAR i,j,coste:CARDINAL;
BEGIN
   coste:=0;
   FOR j:=1 TO N DO (* lo hacemos por columnas *)
```

```
      INC(coste,CosteCol(m,j));
   END;
   RETURN coste;
END Coste;

PROCEDURE h(n:nodo):CARDINAL;
(* funcion de coste de la estrategia LC *)
BEGIN
   RETURN Coste(n^.matriz);
END h;
```

Y otra que permita podar aquellos nodos cuyo coste hasta el momento supere el alcanzado por una solución ya encontrada:

```
PROCEDURE EsAceptable(n:nodo):BOOLEAN;
BEGIN
   RETURN Valor(n)<=cota;
END EsAceptable;
```

Esta función hace uso de otra que es necesario implementar:

```
PROCEDURE Valor(n:nodo):CARDINAL;
BEGIN
   RETURN h(n);
END Valor;
```

y que devuelve el coste acumulado hasta el momento. Esto tiene sentido ya que nos piden encontrar la solución de menor coste. Otra de las funciones que es necesario implementar es la que determina cuándo un nodo es solución. En nuestro caso consiste en decidir cuándo hemos conseguido acomodar hasta la *N*-ésima tarea:

```
PROCEDURE EsSolucion(n:nodo):BOOLEAN;
BEGIN
   RETURN n^.k=N;
END EsSolucion;
```

En cuanto a la función *NoHaySolucion*, que devuelve un valor especial para indicar que el problema no admite solución, sabemos que para este problema eso no ocurrirá nunca pues siempre existe al menos una solución, que es la que asigna una tarea a cada persona.

Por su parte, la función *Eliminar* es la que va a devolver al sistema los recursos ocupados por un nodo, y es la que actúa como "destructor" del tipo abstracto de datos:

```
PROCEDURE Eliminar(VAR n:nodo);
```

```
BEGIN
   DISPOSE(n);
END Eliminar;
```

Con esto finaliza nuestra implementación del módulo "Nodos". El problema queda resuelto escogiendo la función del esquema que encuentra la mejor de todas las soluciones.

Para los valores iniciales dados en el primer ejemplo, el algoritmo encuentra un asignación óptima de coste 97, que es la que representa el vector solución [4,2,3,1], esto es, asigna a la persona *a* la tarea 4, a la persona *b* la tarea 2, a la persona *c* la tarea 3 y a la persona *d* la tarea 1. En la exploración del árbol de expansión para estos datos obtenemos los siguientes valores:

Núm. nodos generados	38
Núm. nodos analizados	18
Núm. nodos podados	20

Obsérvese el buen funcionamiento de la estrategia LC, pues con sólo el análisis de 18 nodos consigue descubrir la asignación óptima.

Respecto al segundo ejemplo, el algoritmo encuentra un asignación óptima de coste 60, que es la representada por el vector [1,4,5,3,2], obteniéndose los siguientes valores de exploración del árbol de expansión:

Núm. nodos generados	167
Núm. nodos analizados	84
Núm. nodos podados	83

7.10 LAS *n* REINAS

El problema de las *n* reinas, ya expuesto en el apartado 6.2, consiste en encontrar una disposición de todas ellas en un tablero de ajedrez de tamaño $n \times n$ de forma que ninguna amenace a otra.

Necesitamos resolver este problema utilizando Ramificación y Poda mediante las estrategias FIFO y LIFO, y comparar ambas soluciones.

Solución (☺)

El estudio del árbol de expansión de este problema ya es conocido, y sólo recordaremos que se basa en construir un vector solución formado por *n* enteros positivos, donde el *k*-ésimo de ellos indica la columna en donde hay que colocar la reina de la fila *k* del tablero. En cada paso o etapa disponemos de *n* posibles opciones a priori (las *n* columnas), pero podemos eliminar aquellas columnas que

den lugar a un vector que no sea *k*-prometedor, esto es, que la nueva reina incorporada amenace a las ya colocadas.

Más formalmente, diremos que el vector *s* de *n* elementos es *k*-prometedor (con $1 \leq k \leq n$) si y sólo si para todo par de enteros *i* y *j* entre 1 y *k* se verifica que $s[i] \neq s[j]$ y $|s[i] - s[j]| \neq |i - j|$.

Esto da lugar a un árbol de expansión razonablemente manejable (del orden de 2000 nodos para $n = 8$) y por tanto convierte el problema en "tratable".

Veamos cómo la técnica de Ramificación y Poda aborda dos recorridos distintos de ese árbol, en profundidad y en anchura, y qué resultados obtiene.

Comenzaremos definiendo entonces el tipo abstracto de datos que representa los nodos. Como han de ser autónomos para poder abordar los procesos de ramificación, poda y reconstrucción de la solución con la información contenida en cada uno de ellos, la manera natural de implementarlos es como sigue:

```
CONST N = ...; (* dimension del tablero *)
TYPE solucion = ARRAY[1..N] OF CARDINAL;
TYPE nodo = POINTER TO RECORD
                          k:CARDINAL; s:solucion;
                       END;
```

En el registro, *k* indica el nivel y *s* contiene la solución construida hasta el momento. De esta forma, el nodo inicial que forma la raíz del árbol contendrá una solución vacía:

```
PROCEDURE NodoInicial():nodo;
   VAR n:nodo; i,j:CARDINAL;
BEGIN
   NEW(n);
   FOR i:=1 TO N DO
      n^.s[i]:=0
   END;
   n^.k:=0;
   RETURN n;
END NodoInicial;
```

Respecto al proceso de ramificación, cada nodo puede generar hasta *n* nodos hijos, cada uno con la columna de la reina que ocupa la fila en curso. Lo que ocurre es que descartaremos todos aquellos que no den lugar a un vector solución *k*-prometedor:

```
PROCEDURE Expandir(n:nodo;VAR hijos:ARRAY OF nodo):CARDINAL;
    VAR i,j,nhijos:CARDINAL; p:nodo;
BEGIN
   nhijos:=0;
   i:=n^.k+1;
   IF i>N THEN RETURN nhijos END; (* caso especial *)
```

```
      FOR j:=1 TO N DO
         IF EsKprometedor(n^.s,i-1,j) THEN
            INC(nhijos);
            Copiar(n,p);
            p^.s[i]:=j;
            INC(p^.k);
            hijos[nhijos-1]:=p;
         END
      END;
      RETURN nhijos;
   END Expandir;
```

Las funciones auxiliares de las que hace uso la función *Expandir* son las siguientes:

```
   PROCEDURE Copiar(VAR n1,n2:nodo); (* duplica un nodo *)
   BEGIN
      NEW(n2);
      n2^.s:=n1^.s; n2^.k:=n1^.k;
   END Copiar;

   PROCEDURE EsKprometedor(s:solucion;k,j:CARDINAL):BOOLEAN;
      VAR i:CARDINAL;
   BEGIN
      FOR i:=1 TO k DO
         IF (s[i]=j)OR(ValAbs(s[i],j)=k+1-i) THEN RETURN FALSE END;
      END;
      RETURN TRUE;
   END EsKprometedor;
```

Esta función hace uso de la que calcula el valor absoluto de la diferencia de dos enteros no negativos:

```
   PROCEDURE ValAbs(a,b:CARDINAL):CARDINAL;
   (* valor absoluto de la diferencia de sus argumentos: |a-b| *)
   BEGIN
      IF a>b THEN RETURN a-b
      ELSE RETURN b-a
      END
   END ValAbs;
```

Otra función importante es aquella que determina cuándo un nodo es una hoja del árbol de expansión, esto es, una solución al problema. Para ello, basta ver que el vector solución construido es *n*-prometedor:

```
   PROCEDURE EsSolucion(n:nodo):BOOLEAN;
   BEGIN
```

```
   RETURN n^.k=N;
END EsSolucion;
```

Aparte de estas funciones, el resto de los procedimientos que se definen en el interfaz de este tipo abstracto de datos no presentan mayor dificultad.

En primer lugar, las funciones *EsAceptable*, *Valor*, *PonerCota* y *h* no intervienen en el desarrollo de este problema, pues no existen podas a posteriori, es decir, la poda de nodos se realiza durante el proceso de ramificación, y todos aquellos nodos que se generan son válidos porque o son solución del problema, o conducen a una de ellas. La función *Eliminar* es la que devuelve al sistema los recursos utilizados por un nodo:

```
PROCEDURE Eliminar(VAR n:nodo);
BEGIN
   DISPOSE(n);
END Eliminar;
```

La función *NoHaySolucion* es necesaria en este caso porque hay tableros en donde este problema no tiene solución (p.e. para n=3).

```
PROCEDURE NoHaySolucion():nodo;
BEGIN
   RETURN NIL;
END NoHaySolucion;
```

Una vez implementado este módulo, las estrategias FIFO y LIFO que queramos analizar van a llevarse a cabo mediante el uso de una implementación adecuada del módulo "Estruc". Los resultados que hemos obtenido utilizando una y otra hasta conseguir encontrar la primera solución del problema son los siguientes:

	LIFO	FIFO
Núm. nodos generados	124	1965
Núm. nodos analizados	113	1665
Núm. nodos podados	0	0

Como puede apreciarse en la tabla, el recorrido en profundidad del árbol es el más adecuado, y consigue en este caso recorrer el mismo número de nodos que recorría el algoritmo de Vuelta Atrás hasta encontrar la primera solución. Aquí, la primera solución que se encuentra mediante el uso de la estrategia LIFO es la que representa el vector [8, 4, 1, 3, 6, 2, 7, 5].

Por otro lado, es normal que el recorrido en anchura tenga que analizar tantos nodos, pues hasta no llegar al último nivel del árbol no encuentra la solución. Obsérvese además cómo tiene que analizar todos los nodos hasta el nivel n–1 y

generar casi todos los nodos del nivel *n* antes de encontrar la primera solución, que en este caso es [1, 5, 8, 6, 3, 7, 2, 4].

También es fácil analizar cómo se comportan una y otra estrategia cuando lo que le pedimos es que calculen todas la soluciones y no se detengan al encontrar la primera. Esto se consigue sencillamente utilizando la función *RyP_todas()* del esquema que presentamos al principio del capítulo:

	LIFO	FIFO
Núm. nodos generados	2056	2056
Núm. nodos analizados	1965	1965
Núm. nodos podados	0	0

Para este caso ambas estrategias obtienen los mismos resultados antes de encontrar las 92 soluciones que posee el problema para $n = 8$, pues ambas han de recorrer todo el árbol.

Por último, hacer notar que en ningún caso se podan nodos pues, como hemos señalado anteriormente, la poda se realiza durante el proceso de expansión, y no a posteriori.

7.11 EL FONTANERO CON PENALIZACIONES

Supongamos que un fontanero tiene *N* avisos pendientes, y que cada uno de ellos lleva asociado una duración (los días que tarda en realizarse), un plazo límite, y una penalización en caso de que no se ejecute dentro del plazo límite establecido para él (lo que deja de ganar). Por ejemplo, para el caso de cuatro avisos ($N = 4$) podemos tener los siguientes datos:

	1	2	3	4
Duración	2	1	2	3
Plazo límite	3	4	4	3
Penalización	5	15	13	10

En dicha tabla la duración y los plazos están expresados en días, y la penalización en miles de pesetas. Se pide determinar la fecha de comienzo de cada una de las tareas (cero si se decide no realizarla) de forma que la penalización total sea mínima.

Solución (☺)

Comenzaremos analizando la construcción del árbol de expansión para el problema. En primer lugar, hemos de plantear la solución como una secuencia de decisiones, una en cada paso o etapa.

Para ello, nuestra solución estará formada por un vector, con un elemento para cada una de las tareas. Cada uno de estos elementos contendrá el día de comienzo de la tarea correspondiente. Utilizaremos el valor 0 para indicar que tal tarea no se realiza.

De esta forma, inicialmente el vector estará vacío, y en el paso k-ésimo tomaremos la decisión de si hacemos o no la tarea número k, y en caso de decidir hacerla, cuál será su día de comienzo. Por tanto, los valores que puede en principio tomar el elemento en posición k del vector ($1 \leq k \leq N$) estarán comprendidos entre 0 y ($p - d + 1$), siendo p el plazo y d la duración de tal tarea. Sin embargo, todos esos valores no tienen por qué ser válidos; al incluir una tarea habrá de comprobarse que no se solape con las tareas que ya tenía asignadas el vector. Este mecanismo es el que va construyendo el árbol de expansión para este problema.

Teniendo en cuenta las consideraciones realizadas en la introducción de este capítulo, será suficiente realizar el módulo que implementa el tipo abstracto de datos que representa los nodos, pues el resto del programa es fijo para este tipo de algoritmos.

Para ello, comenzaremos definiendo el tipo *nodo* que representará el vector que hemos mencionado anteriormente. Utilizaremos entonces la siguiente estructura de datos:

```
CONST N = ...; (* numero de tareas *)
TYPE solucion = ARRAY[1..N] OF CARDINAL;
TYPE nodo = POINTER TO RECORD
                          penalizacion,k:CARDINAL; s:solucion
                       END;
```

Además del vector solución, las otras dos componentes del registro indican la penalización acumulada hasta el momento y la etapa en curso (k). De esta forma conseguimos que cada nodo sea "autónomo", esto es, que cada uno contenga toda la información relevante para poder realizar los procesos de bifurcación, poda y reconstrucción de la solución encontrada hasta ese momento.

Necesitaremos además dos variables globales al módulo. Una para almacenar la cota superior alcanzada por la mejor solución hasta el momento y otra para guardar la tabla con los datos iniciales del problema:

```
VAR cota:CARDINAL;
VAR tabla:ARRAY [1..N] OF RECORD
                           duracion,plazo,penalizacion:CARDINAL
                          END;
```

Estas variables serán inicializadas en el cuerpo principal del módulo "Nodos":

```
   BEGIN (* Nodos *)
      cota:=MAX(CARDINAL);
```

```
   tabla[1].duracion:=2; tabla[1].plazo:=3;
   tabla[1].penalizacion:=5;
   tabla[2].duracion:=1; tabla[2].plazo:=4;
   tabla[2].penalizacion:=15;
   tabla[3].duracion:=2; tabla[3].plazo:=4;
   tabla[3].penalizacion:=13;
   tabla[4].duracion:=3; tabla[4].plazo:=3;
   tabla[4].penalizacion:=10;
END Nodos.
```

Veamos ahora las funciones de este módulo. En primer lugar la función *NodoInicial* habrá de generar un nodo vacío:

```
PROCEDURE NodoInicial():nodo;
   VAR n:nodo; i:CARDINAL;
BEGIN
   NEW(n);
   FOR i:=1 TO N DO n^.s[i]:=0 END;
   n^.penalizacion:=0; n^.k:=0;
   RETURN n;
END NodoInicial;
```

La estrategia de ramificación está a cargo de la función "*Expandir*". Cada nodo puede generar, como hemos dicho antes, a lo sumo $(p - d + 2)$ hijos, que son los correspondientes a no realizar la tarea o realizarla comenzando en los días 1, 2, ..., $(p - d + 1)$. Esta función sólo generará aquellos nodos que sean válidos, esto es, que sean compatibles con las tareas asignadas previamente.

```
PROCEDURE Expandir(n:nodo;VAR hijos:ARRAY OF nodo):CARDINAL;
   VAR i,j,penalizacion,plazo,duracion,nhijos:CARDINAL; p:nodo;
BEGIN
   nhijos:=0;
   (* en cada etapa generamos los valores de la siguiente tarea *)
   i:=n^.k+1;
   IF i>N THEN RETURN nhijos END; (* caso especial *)
   penalizacion:=tabla[i].penalizacion;
   plazo:=tabla[i].plazo;
   duracion:=tabla[i].duracion;
   (* caso 0: no hacemos esa tarea *)
   INC(nhijos);
   Copiar(n,p);
   INC(p^.k);
   INC(p^.penalizacion,penalizacion);
   hijos[nhijos-1]:=p;
   (* resto de los casos *)
   FOR j:=1 TO (plazo-duracion+1) DO
```

```
      (* comprobamos que es compatible con el resto de tareas *)
         IF EsCompatible(n,i,j,duracion) THEN
            INC(nhijos);
            Copiar(n,p);
            p^.s[i]:=j; INC(p^.k);(* aqui no hay penalizacion *)
            hijos[nhijos-1]:=p;
         END;
      END;
      RETURN nhijos;
   END Expandir;
```

Esta función hace uso de un procedimiento que permite duplicar un nodo:

```
   PROCEDURE Copiar(VAR n1,n2:nodo);
      VAR i,j:CARDINAL;
   BEGIN
      NEW(n2);
      FOR i:=1 TO N DO n2^.s[i]:=n1^.s[i] END;
      n2^.penalizacion:=n1^.penalizacion;
      n2^.k:=n1^.k;
   END Copiar;
```

Y también de una función que decide si la decisión a tomar es compatible con las asignaciones previamente almacenadas en el vector:

```
   PROCEDURE EsCompatible(n:nodo;nivel,comienzo,duracion:CARDINAL)
                                                              :BOOLEAN;
      VAR i,fin,com:CARDINAL;
   BEGIN
      FOR i:=1 TO nivel-1 DO
         com:=n^.s[i];
         fin:=com+tabla[i].duracion-1;
         IF com<>0 THEN
          IF NOT((com>(comienzo+duracion-1))OR(fin<comienzo)) THEN
            RETURN FALSE
          END;
         END;
      END;
      RETURN TRUE;
   END EsCompatible;
```

Por otro lado, también es necesario implementar la función que realiza la poda. En este caso, vamos a podar aquellos nodos cuya penalización hasta el momento supere la alcanzada por una solución ya encontrada:

```
   PROCEDURE EsAceptable(n:nodo):BOOLEAN;
   BEGIN
```

```
   RETURN Valor(n)<=cota;
END EsAceptable;
```

Esta función hace uso de otra que es necesario implementar:

```
PROCEDURE Valor(n:nodo):CARDINAL;
BEGIN
   RETURN n^.penalizacion;
END Valor;
```

que devuelve la penalización de la solución construida hasta el momento. Esto tiene sentido pues nos piden encontrar la solución de menor penalización.

Veamos ahora la función de coste para los nodos. Como buscamos la solución de menor penalización, este valor se presenta como un buen candidato para tal función:

```
PROCEDURE h(n:nodo):CARDINAL;
BEGIN
   RETURN n^.penalizacion;
END h;
```

Otra de las funciones que es necesario implementar es la que determina cuándo un nodo es solución. En nuestro caso consiste en decidir cuándo hemos sido capaces de acomodar hasta la tarea *N*-ésima:

```
PROCEDURE EsSolucion(n:nodo):BOOLEAN;
BEGIN
   RETURN n^.k=N;
END EsSolucion;
```

En cuanto a la función *NoHaySolucion*, que devuelve un valor especial para indicar que el problema no admite solución, sabemos que para este problema eso no ocurrirá nunca, pues siempre existe al menos una solución, que es la que representa el vector [0, 0, ..., 0], es decir, siempre podemos no hacer ninguna tarea, cuya penalización coincide con la suma de las penalizaciones de todas las tareas. Esta sería, por ejemplo, la solución al problema siguiente:

	1	2	3	4
Duración	4	5	6	7
Plazo límite	3	4	4	3
Penalización	5	15	13	10

en donde ninguna tarea puede realizarse por ser sus duraciones mayores a sus plazos límite.

Por su parte, la función *Eliminar* es la que va a devolver al sistema los recursos ocupados por un nodo, y es la que actúa como "destructor" del tipo abstracto de datos:

```
PROCEDURE Eliminar(VAR n:nodo);
BEGIN
   DISPOSE(n);
END Eliminar;
```

Con esto finaliza nuestra implementación del módulo "Nodos". El problema queda resuelto escogiendo la función del esquema que encuentra la mejor de todas las soluciones.

Para los valores iniciales dados en el enunciado, el algoritmo encuentra seis soluciones óptimas de penalización 15, que son:

[0, 1, 2, 0]
[0, 1, 3, 0]
[0, 2, 3, 0]
[0, 3, 1, 0]
[0, 4, 1, 0]
[0, 4, 2, 0]

Obteniéndose los siguientes valores de exploración del árbol de expansión:

Núm. nodos generados	51
Núm. nodos analizados	30
Núm. nodos podados	16

Nos podemos plantear también lo que ocurriría si hubiésemos escogido una estrategia distinta de la LC, esto es, LIFO o FIFO. Siguiendo nuestro esquema, bastaría con sustituir el módulo de implementación del tipo abstracto de datos "Estruc" acomodándolo a una pila o a una cola.

Estos cambios permitirán recorrer el árbol de expansión en profundidad o en anchura, respectivamente. Los valores que se obtienen para las tres estrategias son los siguientes:

	LC	LIFO	FIFO
Núm. nodos generados	51	48	58
Núm. nodos analizados	30	27	36
Núm. nodos podados	16	12	11

Como era de esperar, el peor caso es para la búsqueda en anchura. Además, para estos datos vemos cómo la estrategia LIFO mejora sensiblemente la LC; el hecho

de que sea un poco mejor depende sólo de los datos del problema. Para otros ejemplos los valores que se obtienen siguiendo esta estrategia son peores.

Por último, cabe preguntarse qué ocurre si deseamos buscar no la mejor, sino todas las posibles soluciones de este problema. Para este ejemplo los valores que se obtienen son:

Núm. nodos generados	58
Núm. nodos analizados	36
Núm. nodos podados	0

Por supuesto, estos valores se obtienen independientemente de la estrategia seguida (FIFO, LIFO o LC). Además, es curioso observar cómo los dos primeros valores coinciden con los obtenidos para la estrategia FIFO en la búsqueda de la mejor solución. La razón es bien sencilla, pues si vamos recorriendo el árbol de expansión en anchura necesitaremos recorrerlo entero para dar con la mejor solución, ya que todas las soluciones se encuentran siempre en el nivel N, y para llegar a él esta estrategia necesita haber construido completamente todos los niveles anteriores.