Contenidos

Capítulo 1

ALGORÍTMICA

1.1 Definición de algoritmo

1.1.1 Definición informal

Un **algoritmo** es un conjunto de reglas para resolver un problema, pero este conjunto de reglas debe estar completamente definido sin dejar lugar a dudas en su interpretación (concepto de **definibilidad**). Un conjunto de reglas de este tipo, para ser un algoritmo, debe poder implementarse en un lenguaje de programación por medio de un programa, debe tener unos datos de entrada (aunque el número de datos de entrada puede ser cero), y producir unos datos de salida; y debe realizarse, para cada conjunto de datos de entrada, un número finito de pasos en un tiempo finito (concepto de **finitud**) para producir el conjunto de datos de salida (la salida puede ser que no encuentre solución). Para los mismos datos de entrada pueden producirse o no siempre los mismos datos de salida; en el caso en que se produzcan siempre los mismos datos de salida el algoritmo se llama **determinista**, y en caso contrario **no determinista**. Un algoritmo sería una función $f : E \to S$, pero si es no determinista puede que no sea posible definir la función, lo que pasa en Programación Concurrente y Paralela, donde en muchos casos la salida puede depender no sólo de la entrada sino también de factores temporales.

A pesar de la finitud de un algoritmo puede que éste no sea efectivo por no poder ejecutarse en un tiempo finito lo suficientemente pequeño como para poder ser tomado como finito según una medida humana. Como ejemplo se puede ver el problema de las torres de Hanoi:

Ejemplo 1.1 *Problema de las torres de Hanoi.* Se tienen tres varillas verticales que llamaremos `origen`, `destino` y `pivote`, y una serie de discos de distintos tamaños y con un orificio central. Los discos se encuentran en la varilla `origen` y pretendemos pasarlos a la varilla `destino`. Los discos están inicialmente de menor a mayor tamaño (de arriba a abajo) en la varilla `origen` y deben quedar finalmente en la misma posición en la varilla `destino`. Se puede usar la varilla `pivote` para hacer movimientos intermedios, consistiendo cada movimiento en cambiar un único disco de una varilla a otra (obviamente el disco en la parte superior de un montón de discos), y no pudiendo en ningún momento haber un disco por encima de otro de menor tamaño en una misma varilla.

El problema se puede resolver recursivamente de la forma:

Hanoi(*origen*,*destino*,*pivote*,*discos*):
  si $discos \neq 1$
    Hanoi(*origen*,*pivote*,*destino*,$discos - 1$)
    moveruno(*origen*,*destino*)
    Hanoi(*pivote*,*destino*,*origen*,$discos - 1$)
  en otro caso
    moveruno(*origen*,*destino*)

El tiempo de ejecución en el caso base (cuando hay un único disco) será constante $t(1) = a$; y en el otro caso será $t(n) = 2t(n-1) + b$, con b constante. Expandiendo la recurrencia llegamos a $t(n) = 2^{n-1}(a+b)-b$, con lo que para 60 discos y suponiendo que las constantes a y b tienen un valor de $1\mu s$ (una operación básica tarda en ejecutarse un micro segundo) el tiempo necesario para resolver el problema sería de aproximadamente ¡365 siglos!.

La demostración de que una regla de cálculo es un algoritmo (cumple las características aquí especificadas) no es fácil. Incluso en los casos más sencillos puede ser bastante complicado.

1.1.2 Definición de Knuth

Una definición más formal de algoritmo se puede encontrar en el libro de Knuth [12]:

Definición 1.1 Un **método de cálculo** es una cuaterna (Q, I, W, f) donde:
Q es un conjunto que contiene a I y W, y
$f : Q \to Q$ con $f(w) = w$ para todo w perteneciente a W.
Q es el **conjunto de estados del cálculo**,
I es el **conjunto de estados de entrada**,
W es el **conjunto de estados de salida** y
f es la **regla de cálculo**.

Definición 1.2 Una **secuencia de cálculo** es $x_0, x_1, x_2, \ldots$ donde $x_0 \in I$ y $\forall\ k \geq 0$: $f(x_k) = x_{k+1}$. La secuencia de cálculo acaba en n pasos si n es el menor entero con $x_n \in W$.

Veamos cómo esta definición formal de algoritmo (el método de cálculo) cumple las propiedades que hemos dicho de manera informal que debe de cumplir un algoritmo:

- Se cumplirá el concepto de finitud si toda secuencia de cálculo a la que pueda dar lugar el método de cálculo es finita.
- Existe un conjunto de posibles entradas, que es I en la definición.
- Existe un conjunto de posibles salidas, que es W en la definición.
- En cuanto a la definibilidad, el algoritmo está definido con el método de cálculo, pero, dado un método de cálculo, ¿se puede hacer un programa? ¿en qué lenguaje? ¿en un tiempo razonable?
- El método de cálculo será eficiente si el valor de n en cada secuencia de cálculo es un valor "razonable".

Ejemplo 1.2 Como ejemplo veremos el **algoritmo de Euclides**:
Dados m y n números enteros positivos con $m > n$ se trata de calcular el $m.c.d(m, n)$ haciendo:

1. Hallar $r = resto(\frac{m}{n})$
2. Si $r = 0$ acabar y $m.c.d(m, n) = n$
3. Si $r \neq 0$ hacer $m = n,\ n = r$ y volver a 1

En este algoritmo tenemos:

- $Q = \{(n)/\ n \in N\} \bigcup \{(m,n)/m,n \in N\} \bigcup \{(m,n,r,1),(m,n,r,2),(m,n,r,3)/\ m,n \in N, y\ r \in N \bigcup \{0\}\}$
- $I = \{(m,n)/m,n \in N\}$
- $W = \{(n)/n \in N\}$
- $f(n) = (n)$
  $f(m,n) = (m,n,0,1)$ (suponemos que 0 es el valor inicial y final de r)
  $f(m,n,r,1) = (m,n,resto(\frac{m}{n}),2)$
  $f(m,n,r,2) = \left\{ \begin{array}{ccc} (n) & si & r = 0 \\ (m,n,r,3) & si & r \neq 0 \end{array} \right.$
  $f(m,n,r,3) = (n,r,r,1)$
  donde 1, 2 y 3 indican el paso del algoritmo en que estamos, los pares (m,n) indican las entradas y (n) las soluciones.

Para calcular el Máximo común divisor de 6 y 4 la secuencia de cálculo sería: (6,4), (6,4,0,1), (6,4,2,2), (6,4,2,3), (4,2,2,1), (4,2,0,2), (2).

Una definición formal, como la de método de cálculo, nos puede ayudar a deteminar de manera científica y rigurosa si un algoritmo acaba, si es eficiente, ... (las propiedades de las que hemos hablado), pero ¿nos sirve para diseñar programas reales y analizarlos? En el ejemplo que hemos visto hemos hecho el programa (en seudocódigo, con `goto`, ...) y después hemos comprobado la definición formal; pero esto no es lo que nos interesa. En el curso se analizarán y diseñarán algoritmos de una manera más informal pero más cercana a la realidad de la programación, quedando los aspectos más científicos para otras asignaturas, como Modelos Abstractos de Cálculo.

1.2 Algorítmica

1.2.1 Programación

Un algoritmo puede representarse en un lenguaje como el que usamos normalmente, pero este tipo de lenguajes para la comunicación humana deja un gran margen a la intuición y a la posibilidad de múltiples interpretaciones, lo que no debe ocurrir con un algoritmo, por lo que la expresión final de un algoritmo debe realizarse en un lenguaje de programación (implementado o no) por medio de un programa, pues se supone que el ordenador capaz de utilizar este programa está totalmente desprovisto de intuición (aunque con técnicas de Inteligencia Artificial se pueda simular), con lo que el conjunto de reglas que constituyen el algoritmo no dará lugar a dudas en su interpretación siempre que quien quiera que lo lea haga un mínimo esfuerzo. Lo que usaremos será un seudolenguaje que no definiremos pero que debe ser lo suficientemente claro para que podamos entenderlo, y debe corresponder a la idea que tenemos de un lenguaje imperativo.

La representación de un algoritmo se puede hacer en un lenguaje natural, en un seudolenguaje, y en un lenguaje de programación, y la evolución del algoritmo hasta su representación como programa se puede representar según la figura 1.1, en donde se ve que el algoritmo y la estructura de los datos evolucionan al mismo tiempo hasta llegar al programa ejecutable y a la estructura de datos sobre la que actúa el programa.

1.2.2 Algorítmica

La algorítmica estudia técnicas para la realización de algoritmos eficientes. Diremos que un algoritmo es más eficiente que otro cuando gasta menos recursos, como pueden ser tiempo y memoria, en la resolución de un problema; pero la comparación entre algoritmos no es tan sencilla de hacer, pues puede ocurrir que un algoritmo sea más eficiente que otro para unos determinados conjuntos de datos de entrada, y menos eficiente para otros, por lo que se puede estudiar la eficiencia de un algoritmo en el caso más favorable, en el peor caso y en término medio y, aunque puede parecer que en término medio es el mejor tipo de análisis, puede ocurrir también que tengamos

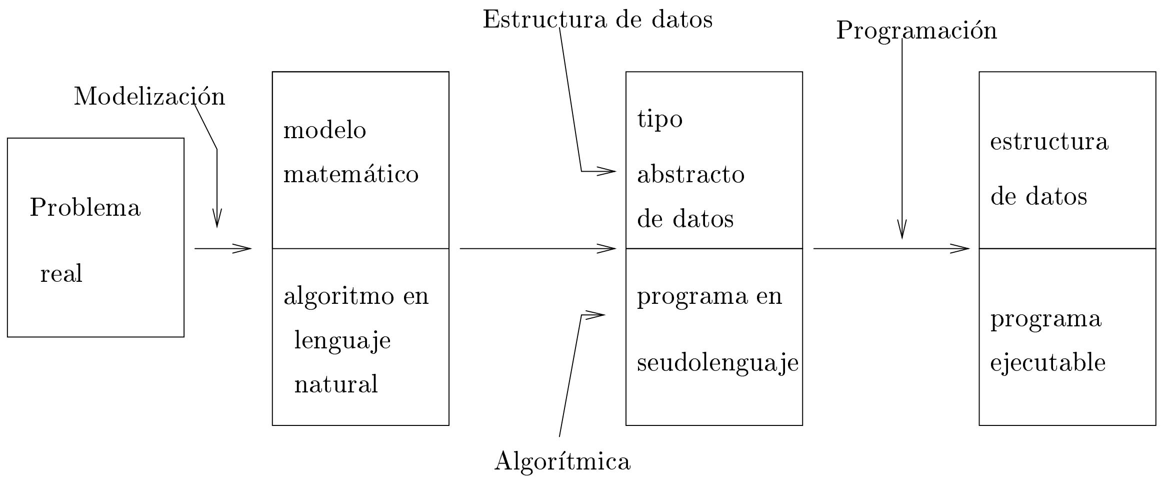


Figura 1.1: Proceso de programación

unos determinados datos de entrada con una probabilidad mayor que otros, por lo que la media debería ser ponderada y se presenta la dificultad adicional del cálculo de las probabilidades. Por tanto, la Algorítmica estudiará, además de técnicas de diseño de algoritmos, técnicas para medir la eficiencia de los algoritmos, de manera que se pueda decidir qué algoritmo es mejor utilizar para la resolución de un problema concreto.

1.2.3 Análisis de algoritmos

Como ejemplo de comparación de algoritmos podemos estudiar la ordenación de números según los dos algoritmos siguientes escritos en seudolenguaje y acompañando a cada instrucción que consideramos elemental un par de números para poder referirnos a ella en el estudio:

Algoritmo 1.1 ordenar1:

1.1 fin=false
1.2 mientras no fin
1.3   fin=true
1.4   para $i = 1, 2, \ldots, n-1$
1.5   si $x_i > x_{i+1}$
1.6   cambiar
1.7   fin=false

Algoritmo 1.2 ordenar2:

2.1 para $i = 1, 2, \ldots, n-1$

2.2 para $j = i+1, i+2, \dots, n$
2.3 si $x_i > x_j$
2.4 cambiar

Estudiamos el número de veces que cada uno de los algoritmos pasa por una sentencia de las que hemos considerado elemental para los conjuntos de entradas: 1,2,3,4 y 4,3,2,1. El estudiar los algoritmos en casos tan particulares como estos (para un tamaño de la entrada e incluso para una entrada fija) puede ser una buena decisión si no se sabe abordar el problema desde el principio de una manera más general; pero hay que tener en cuenta que lo que nos interesa es el estudio general del algoritmo y no el análisis de un caso particular.

Para la entrada 1,2,3,4 ordenar1 pasará por:
1.1 1.2 1.3 1.4 1.5 para $i = 1$
1.4 1.5 para $i = 2$
1.4 1.5 para $i = 3$
1.2 (salida del bucle)
Para 1,2,3,4 ordenar2 pasará por:
2.1 2.2 2.3 para $i = 1,\ j = 2$
2.2 2.3 para $i = 1,\ j = 3$
2.2 2.3 para $i = 1,\ j = 4$
2.1 2.2 2.3 para $i = 2,\ j = 3$
2.2 2.3 para $i = 2,\ j = 4$
2.1 2.2 2.3 para $i = 3,\ j = 4$
Para la entrada 4,3,2,1 ordenar1 pasará por:
1.1 1.2 1.3 1.4 1.5 1.6 1.7 (quedará 3 4 2 1)
1.4 1.5 1.6 1.7 (quedará 3 2 4 1)
1.4 1.5 1.6 1.7 (quedará 3 2 1 4)
1.2 1.3 1.4 1.5 1.6 1.7 (quedará 2 3 1 4)
1.4 1.5 1.6 1.7 (quedará 2 1 3 4)
1.4 1.5
1.2 1.3 1.4 1.5 1.6 1.7 (quedará 1 2 3 4)
1.4 1.5
1.4 1.5
1.2 1.3 1.4 1.5
1.4 1.5
1.4 1.5
1.2 (salida del bucle)
Para 4,3,2,1 ordenar2 pasará por:
2.1 2.2 2.3 2.4 (quedará 3 4 2 1)
2.2 2.3 2.4 (quedará 2 4 3 1)
2.2 2.3 2.4 (quedará 1 4 3 2)
2.1 2.2 2.3 2.4 (quedará 1 3 4 2)

2.2 2.3 2.4 (quedará 1 2 4 3)
2.1 2.2 2.3 2.4 (quedará 1 2 3 4)

Como se puede ver, los dos algoritmos son bastante ineficientes y se han usado sólo para hacer su comparación. El ordenar1 ordena 1,2,3,4 ejecutando 10 sentencias elementales, mientras que ordenar2 lo hace en 15, lo que puede hacer suponer que ordenar1 es mejor algoritmo que ordenar2; pero para ordenar 4,3,2,1 ordenar1 necesita 46 sentencias elementales y ordenar2 21, de donde parece que ordenar2 es mejor que ordenar1. Para poder compararlos habría que hacer una media para todas las posibles entradas ordenadas de todas las maneras posibles, pero puede que esto tampoco sea lo más adecuado pues es mucho más probable que necesitemos ordenar un número pequeño de números (entre unos cientos y unos miles) que un billón de números. Lo que se hace normalmente es comparar los algoritmos dependiendo del tamaño de la entrada. No siempre es fácil determinar qué se considera tamaño de la entrada, pero en el caso de ordenar números puede ser la cantidad de datos a ordenar.

Hemos comparado los dos algoritmos para un cierto tamaño de las entradas y una cierta distribución de éstas. Para hacer una comparación más general empezaremos suponiendo una entrada ordenada de n números. En este caso el algoritmo 1.1 tarda un tiempo:

$$t_n = 1 + t_w + 1 \tag{1.1}$$

siendo t_w el tiempo de ejecución del bucle mientras y suponiendo que una asignación y una comparación tardan una unidad de tiempo en ejecutarse (el primer 1 corresponde a la asignación inicial; y el segundo 1 corresponde a la última comprobación del bucle mientras, en la que no se entra al cuerpo del bucle), y

$$t_w = 2 + t_f \tag{1.2}$$

siendo t_f el tiempo de ejecución del bucle "para" que se ejecuta $n-1$ veces y en cada una de ellas tiene lugar una asignación a i (la i varía de 1 a $n-1$) y una comparación. No se realiza ningún cambio ni la asignación pues estamos suponiendo la entrada ordenada. Además, por estar la entrada ordenada sólo se pasa una vez por el cuerpo del while. De este modo:

$$t_n = 2 + t_w = 4 + (n-1)2 = 2n + 2 \tag{1.3}$$

Y el segundo algoritmo tendrá un tiempo:

$$t_n = \sum_{i=1}^{n-1}\left(1 + \sum_{j=i+1}^{n} 2\right) = \sum_{i=1}^{n-1}(1 + 2(n-i)) = n - 1 + \sum_{i=1}^{n-1} 2(n-i) = n^2 - 1 \tag{1.4}$$

Las gráficas correspondientes a estas fórmulas se muestran en la figura 1.2, donde se ve que el algoritmo 1.1 es mejor que el 1.2 (en el caso más favorable) porque su crecimiento es menor.

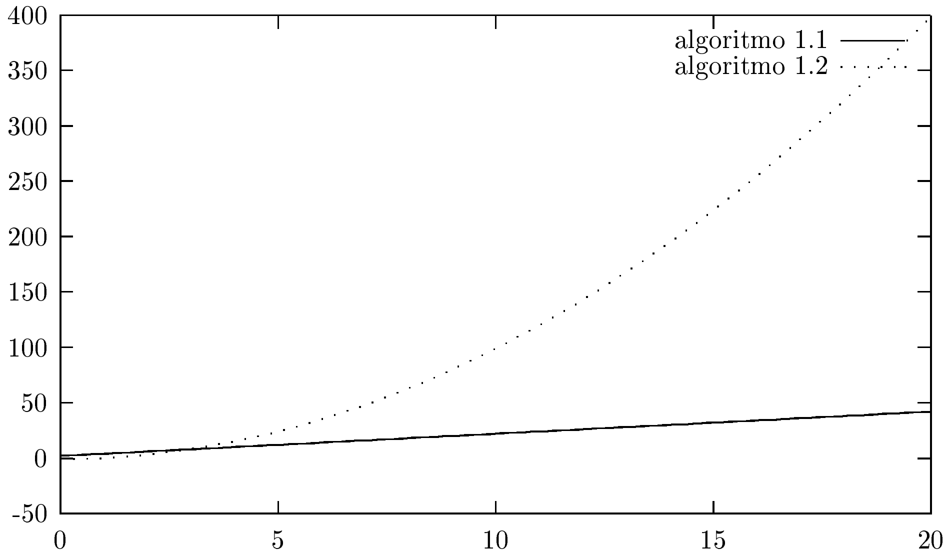


Figura 1.2: Tiempos de ejecución en el caso más favorable de los dos algoritmos

Analizamos los algoritmos en el caso más desfavorable (entrada ordenada inversamente):

En ordenar1 tendremos:

$$t_n = 2 + \sum_{j=1}^{n} \left(2 + \sum_{i=1}^{n-1} 2 + \sum_{i=1}^{n-j} 2 \right) = 3n^2 - n + 2 \tag{1.5}$$

y para ordenar2:

$$t_n = \sum_{i=1}^{n-1} \left(1 + \sum_{j=i+1}^{n} 3 \right) = \frac{3}{2}n^2 - \frac{n}{2} - 1 \tag{1.6}$$

El tipo de análisis que hemos hecho con los dos métodos de ordenación anteriores tiene una serie de inconvenientes: hemos considerado que todas las sentencias tardan lo mismo en ejecutarse, lo que no es verdad; tampoco en todas las máquinas y con todos los compiladores idénticas sentencias tardan lo mismo; no hemos tenido en cuenta el tipo de dato que tienen los x_i; ni sabemos si los datos que queremos ordenar van a estar ya ordenados, inversamente ordenados o arbitrariamente ordenados. Por todo esto, lo que haremos al analizar un algoritmo no será calcular su tiempo de ejecución, sino hacer una estimación del tiempo pero teniendo en cuenta que éste dependerá de una amplia gama de factores.

Además, no habrá reglas generales para analizar algoritmos, aunque veremos las técnicas más usuales, y una comparación más precisa dependerá del sistema en que queramos compararlos (máquina, compilador, ...).

Capítulo 2

ANÁLISIS DE ALGORITMOS

2.1 Introducción

El análisis de algoritmos tiene sentido como paso previo a la realización de un programa, o a posteriori.

Como paso previo tiene sentido como cualquier otro estudio previo para evitar la pérdida de tiempo ante la máquina; para determinar si un algoritmo es bueno y merece la pena hacer un programa para ese algoritmo o pensar otro mejor; y para poder determinar entre dos algoritmos o programas dados cuál es preferible.

Como estudio a posteriori tiene sentido para comparar dos programas o para encontrar la causa del mal funcionamiento de un programa.

2.1.1 Tiempo de ejecución

En el análisis de algoritmos se puede estudiar el tiempo de ejecución y la ocupación de memoria. Lo que se intentará es que los programas se ejecuten en poco tiempo y necesitando poca memoria (normalmente el uso de poca memoria implica una mejora en el tiempo de ejecución). El tiempo se mide en unidades de tiempo desde que empieza la ejecución del programa hasta que se obtienen los resultados. No calcularemos un valor exacto del tiempo de ejecución pues no tendremos un programa sino un algoritmo, y el tiempo real de ejecución dependerá del programa que se haga (por tanto de la experiencia del programador), de la máquina en que se ejecute, del compilador utilizado, de los tipos de datos, de los usuarios trabajando en el sistema, etc... Lo que se intentará calcular es la forma en que crece la función tiempo de ejecución en función del tamaño de la entrada. Por ejemplo, si tenemos varios números a ordenar dependerá de la cantidad de números; si calculamos el factorial de un número dependerá del valor de éste; si sumamos enteros largos dependerá de la cantidad de enteros a sumar y de las longitudes de los enteros, por lo que hay veces que el tiempo es función de dos parámetros o más. Muchas veces ni siquiera seremos capaces de calcular la forma en que crece el tiempo de ejecución, por lo que podemos intentar calcular el tiempo en el caso más favorable o en el más desfavorable, encontrando de este modo una cota inferior y otra superior del tiempo de ejecución. Otras veces nos puede interesar obtener el tiempo promedio sumando los tiempos de cada una de las posibles entradas por la probabilidad de que aparezca la entrada:

$$t_p(n) = \sum_{\sigma \in S} t(\sigma)\, p(\sigma) \tag{2.1}$$

donde S es el conjunto de todas las posibles entradas. El tiempo promedio no siempre se calcula así, sino que algunas veces es preferible calcularlo contando las operaciones afectada cada operación por la probabilidad con que se ejecuta. De este modo se tiene una aproximación del tiempo de ejecución y no unas cotas como con el tiempo en el caso más favorable y más desfavorable. Para cualquiera de los tiempos tratados nos puede ocurrir que por las características del algoritmo no seamos capaces de determinar su valor, y en estos casos podemos intentar a su vez encontrarles una cota inferior y otra superior. Este sentido tendrán las notaciones asintóticas que estudiaremos posteriormente.

Errores comunes Hay una serie de fallos usuales de los que es conveniente huir (¡sobre todo en los exámenes!):

- El caso más favorable no es tener el menor tamaño posible en la entrada, sino, dado un tamaño de entrada fijo, cuál es la distribución de los datos que necesita de un menor tiempo de ejecución.
- El tiempo promedio no es la media de los tiempos más favorable y más desfavorable.
- No siempre los casos más favorables y más desfavorables son los que aparentan. Por ejemplo, en una ordenación rápida (Quick-sort) el caso más favorable puede no ser el de tener los datos ordenados sino todo lo contrario: ese caso puede ser el más desfavorable.

2.1.2 Ocupación de memoria

En cuanto a la ocupación de memoria se puede decir que es la cantidad de memoria necesaria para poder ejecutar el programa y, por tanto, será la mayor cantidad de memoria que se ocupa durante la ejecución del programa.

Todo lo dicho sobre los tiempos de ejecución, la imposibilidad de calcular su valor exacto y las necesidades de calcular cotas, sirve también para la ocupación de memoria.

Ejemplo 2.1 Como ejemplo de ocupación de memoria vemos el caso de la función factorial:

fact(n):
  si $n \neq 1$
    return(fact($n-1$)n)
  en otro caso
    return 1

Si hacemos la llamada fact(10) necesitamos espacio para $n = 10$ y otro para almacenar el valor que devuelve fact (se ocupa en realidad más espacio pues se necesitan índices de regreso, etc...), al hacer la llamada con $n = 9$ se necesitarán otras dos posiciones para el valor de n y el valor a retornar por la función, y así sucesivamente. Cuando hagamos fac(1) tendremos diez espacios para diez enes distintas y otros diez para los diez valores a devolver por fact, y a partir de ahí se deshacen las llamadas recursivas y se libera la memoria, con lo que en el momento de mayor ocupación hemos necesitado 20 posiciones. Si en vez de con $n = 10$ lo hacemos con un n general necesitamos $2n$ posiciones, por lo que la ocupación de memoria es $m(n) = 2n$, donde podemos prescindir del 2 pues lo que nos interesa es la forma en que crece la función, ya que este valor lo hemos obtenido haciendo unas suposiciones un tanto artificiales.

2.1.3 Otros factores

Casi siempre lo que se estudia es el tiempo de ejecución, pues suelen aparecer antes los problemas con tiempos de ejecución que con ocupación de memoria.

Además de determinar la "bondad" de un algoritmo cuantitativamente estudiando el tiempo de ejecución y la ocupación de memoria, hay que tener en cuenta que en lo bueno que sea un programa entran otros factores como pueden ser el que sea fácil de mantener y de entender, fácil de programar, y que haga un uso eficiente de los recursos de que disponemos.

2.1.4 Conteo de instrucciones

La forma más básica de estudio del tiempo de ejecución consiste en contar las instrucciones que se ejecutan a lo largo de la ejecución de un programa.

Ejemplo 2.2 Estudiaremos el algoritmo de búsqueda secuencial con centinela:

  $i = 0$
  $a[n+1] = x$
  repetir
    $i = i + 1$
  hasta $a[i] = x$

Las dos primeras asignaciones se ejecutan siempre, y también se ejecuta siempre una vez el cuerpo del bucle y la comprobación final, por lo que en el caso mejor (cuando $a[1] = x$) se ejecutan 4 instrucciones que consideraremos elementales.

En el peor caso (cuando x no está en el array) tanto el cuerpo como la comprobación del bucle se ejecutan $n+1$ veces, por lo que se ejecutan $2n+4$ instrucciones.

En el caso promedio, si consideramos que x está en el array con probabilidad p (por lo tanto no está con probabilidad $1-p$) y que supuesto que está tiene la misma probabilidad de estar en cualquiera de los n lugares del array, tendremos una probabilidad $\frac{p}{n}$ de que esté en el lugar i con i entre 1 y n. Como estando en el lugar i el número de operaciones es $2i+2$, tendremos que el número medio de operaciones es:

$$\sum_{i=1}^{n}\left(\frac{p}{n}(2+2i)\right)+(4+2n)(1-p)=(3+n)\,p+(4+2n)(1-p)= \\ 2n+4-(n+1)\,p \qquad (2.2)$$

2.1.5 Influencia de la estructura de los datos

Tanto la complejidad temporal como espacial dependen no sólo del algoritmo sino también de la representación de los datos.

Ejemplo 2.3 Consideramos un grafo no dirigido con n nodos.

El grafo se puede representar por medio de una matriz de adyacencia que tendrá n filas y columnas, o por listas de adyacencia habiendo una lista para cada nodo y estando en la lista asociada a un nodo n_1 todos los nodos n_2 del grafo para los que hay una arista (n_1, n_2). En la figura 2.1 se muestra un ejemplo de un grafo con cuatro nodos y su representación como matriz de adyacencia y con listas de adyacencia.

La complejidad espacial de la representación con matriz de adyacencia es de n^2, y de las listas de adyacencia es $n+a$, donde a representa el número de aristas del grafo.

Un algoritmo de inicialización del grafo tendrá un coste n^2 si trabajamos con matrices de adyacencia, ya que hay que poner a 0 ó 1 todas las entradas de la matriz. Si se trabaja con listas de adyacencia hay que crear los n nodos cabecera de las listas, y por cada arista crear un nodo en una de las listas, de modo que el coste es $n+a$.

Para calcular el grado de salida de un nodo (el número de nodos a los que se puede acceder desde él directamente), con matrices de adyacencia el coste será n ya que hay que recorrer toda la fila de la matriz asociada al nodo e ir contando el número de unos. Con listas de adyacencia hay que recorrer la lista asociada al nodo, y esta lista tiene un nodo por cada nodo del grafo al que se puede acceder directamente desde el nodo del que queremos calcular el grado de salida, con lo que el coste será exactamente el grado de salida del nodo.

Para calcular el grado de entrada de un nodo (número de nodos desde los que se puede acceder directamente a ese nodo), si trabajamos con matrices de adyacencia hay que recorrer la columna asociada a ese nodo, con lo que el coste es n. Si trabajamos

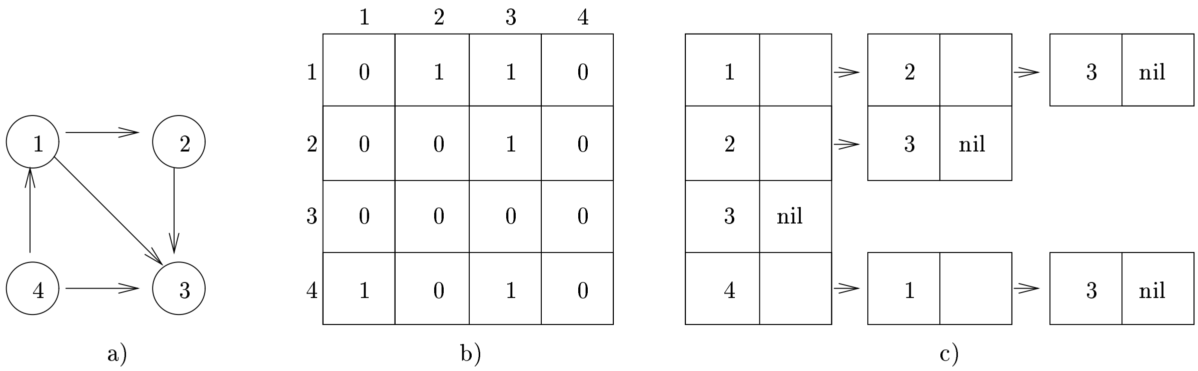


Figura 2.1: a) Grafo. b) Matriz de adyacencia. c) Listas de adyacencia

con listas de adyacencia, hay que recorrer las n listas de adyacencia visitando los a nodos de las listas y comprobando para cada uno de ellos si corresponde al nodo del que queremos calcular el grado de entrada, con lo que el coste será $n + a$.

No podemos por tanto deducir que una representación es mejor que la otra. Para el cálculo del grado de salida es mejor utilizar listas de adyacencia, pero para el grado de entrada es mejor utilizar la matriz de adyacencia.

Un factor a tener en cuenta también es que estos costes irán afectados de unas constantes que serán mayores en el caso de las listas de adyacencia.

2.1.6 Asignación de tiempos

Para el estudio de los programas hay que tener en cuenta que un criterio razonable de asignación de tiempos es:

- las operaciones de entrada/salida se hacen en una unidad de tiempo, en lo que se llama un paso de programa (salvo si son operaciones de entrada/salida de una hilera),
- en los IF y CASE tomaremos el tiempo de ejecución más largo de las distintas partes para obtener una cota superior, y el más corto para obtener una cota inferior,
- en los FOR es un $\sum$ con índices los del FOR,
- cuando hay procedimientos hay que identificar los que no llaman a ninguno, e ir averiguando los tiempos de los procedimientos en función de los procedimientos a que llaman,

- en los WHILE y REPEAT habrá que determinar, si se puede, una cota superior del número de veces que se ejecutan, u obtener los índices de un sumatorio que correspondan a las veces que se pasa por el bucle (tal como con los FOR).

2.2 Notaciones asintóticas

2.2.1 Definiciones

Definición 2.1 *Dada una función $f : N \longrightarrow R^+$ (donde R^+ es el conjunto de los números reales positivos), llamamos* **orden de** *f al conjunto de todas las funciones de N en R^+ acotadas superiormente por un múltiplo real positivo de f para valores de n suficientemente grandes. Se denota $O(f)$, y será:*

$$O(f) = \{t : N \longrightarrow R^+ / \exists c \in R^+, \exists n_o \in N, \forall n \geq n_0 : \ t(n) \leq cf(n)\}$$

Se pueden considerar funciones t que sean indefinidas o que tomen valores negativos para un número finito de valores de $n \in N$, pues sólo nos interesa lo que pase para valores de n suficientemente grandes. De este modo podremos decir que $n^2 - 8 \in O(n^2)$ o que $\log(n-10) \in O(n)$, aunque la función $f(n) = n^2 - 8$ no es una función de $N \longrightarrow R^+$ pues toma valores negativos, y la función $f(n) = \log(n-10)$ tampoco es de $N \longrightarrow R^+$ pues no está definida para valores ≤ 10.

Cuando se estudia un algoritmo se trata de encontrar la función f más simple posible de manera que $t \in O(f)$, donde t representa el tiempo de ejecución del algoritmo (también puede ser una medida de la cantidad de memoria necesaria) en función de la dimensión de la entrada. La función t es normalmente muy difícil de calcular, por lo que se trata de buscar la f que más se le aproxime asintóticamente (cuando n tiende a ∞).

Tenemos establecida una relación de orden en el conjunto de los órdenes de funciones del siguiente modo:

$$O(f) \leq O(g) \ si \ \forall t \in O(f), t \in O(g)$$

con lo que intentamos encontrar el menor $O(f)/t \in O(f)$.

Escribiremos $\leq$ o $\subseteq$ indistintamente pues la ordenación entre órdenes no es sino una inclusión de conjuntos.

Si tenemos que $t(n) = n^2 + 2n - 5$, $t(n) \in O(n^2)$, $t(n) \in O(n^3)$, etc..., pero la que más se acerca es n^2, por lo que decimos que t es del orden de $O(n^2)$. Además, se puede comprobar que $O(n^2 + 2n - 5) = O(n^2)$, y en general se cumple:

- $O(c) = O(d)$ con c y d constantes
- $O(c) \subset O(n)$

- $O(cn+b) = O(dn+e)$, si c y d son constantes positivas
- $O(p) = O(q)$ con p y q polinomios de igual grado
- $O(p) \subset O(q)$ si p es un polinomio de grado menor que q

donde consideramos que las constantes y los coeficientes principales de los polinomios tienen valores positivos.

Definición 2.2 *Dada una función $f : N \longrightarrow R^+$, llamamos* **omega de** f *al conjunto de todas las funciones de N en R^+ acotadas inferiormente por un múltiplo real positivo de f para valores de n suficientemente grandes. Se denota $\Omega(f)$, y será:*

$$\Omega(f) = \{t : N \longrightarrow R^+ / \exists c \in R^+, \exists n_o \in N, \forall n \geq n_0 : \; t(n) \geq cf(n)\}$$

Para la notación omega sirven las mismas relaciones que para el orden, pero cambiando el sentido de las desigualdades.

Definición 2.3 *Dada una función $f : N \longrightarrow R^+$, llamamos* **orden exacto de** f *al conjunto de todas las funciones de N en R^+ que crecen (salvo constantes y asintóticamente) de la misma forma que f. Se denota $\theta(f)$, y será:*

$$\theta(f) = O(f) \bigcap \Omega(f)$$

Se puede comprobar fácilmente que la definición de θ es equivalente a otra forma de definición similar a las dadas para O y Ω:

Propiedad 2.1 *Dadas f y g de N en R^+, $f \in \theta(g) \Leftrightarrow \exists c, d \in R^+, \exists n_0 \in N / \forall n \geq n_0 :$ $cg(n) \leq f(n) \leq dg(n)$.*

Cuando se estudia un algoritmo y se obtiene que $t(n) = n^2 + 2n - 5$ se sabe que su orden exacto es $\theta(n^2)$, pero en otros casos no se puede calcular el orden exacto sino el orden y el omega, con lo que tendremos en un cierto sentido una cota superior e inferior de la forma en que crece el tiempo de ejecución del algoritmo.

Algunas veces no interesa perder la información de los coeficientes del término de mayor orden. En este caso se utiliza la notación $o(f)$.

Definición 2.4 *Dada una función $f : N \longrightarrow R^+$, llamamos* **o pequeña de** f *al conjunto:*

$$o(f) = \left\{ t : N \longrightarrow R^+ / \lim_{n \to \infty} \frac{t(n)}{f(n)} = 1 \right\}$$

En el caso en que t sea un polinomio de grado m, $t \in O(n^m)$ y $t \in o(a_m n^m)$.

2.2.2 Propiedades

Algunas de las propiedades de las notaciones asintóticas vistas hasta ahora son:

Propiedad 2.2 *Si $f \in O(g)$ y $g \in O(h)$ entonces $f \in O(h)$. (Si $f \in \Omega(g)$ y $g \in \Omega(h)$ entonces $f \in \Omega(h)$)*

demostración:

Porque $f \in O(g) : \exists c$ y n_0 $/ \forall n > n_0,\ f(n) \leq cg(n)$

Porque $g \in O(h) : \exists d$ y n_1 $/ \forall n > n_1,\ g(n) \leq dh(n)$

Siendo $e = cd$ y $n_2 = max\,(n_0, n_1)$ se cumple que $\forall n > n_2$ es $f(n) \leq cg(n) \leq eh(n)$, por lo que $f \in O(h)$

Propiedad 2.3 *Si $f \in O(g)$ entonces $O(f) \subseteq O(g)$. (Si $f \in \Omega(g)$ entonces $\Omega(f) \subseteq \Omega(g)$)*

Propiedad 2.4 *Dadas f y g de N en R^+ se cumple:*

i) $O(f) = O(g) \Leftrightarrow f \in O(g)$ y $g \in O(f)$. ($\Omega(f) = \Omega(g) \Leftrightarrow f \in \Omega(g)$ y $g \in \Omega(f)$)

ii) $O(f) \subseteq O(g) \Leftrightarrow f \in O(g)$. ($\Omega(f) \subseteq \Omega(g) \Leftrightarrow f \in \Omega(g)$)

demostración:

i)

$\Rightarrow$:

$f \in O(f) = O(g)$ y $g \in O(g) = O(f)$,

$\Leftarrow$:

$f \in O(g) \Rightarrow$ (por la propiedad 2.3) $O(f) \subseteq O(g)$, y la otra inclusión igual.

ii)

$\Rightarrow$:

$f \in O(f) \subseteq O(g)$,

$\Leftarrow$:

$f \in O(g) \Rightarrow$ (por la propiedad 2.3) $O(f) \subseteq O(g)$.

Hay funciones que no cumplen $f \in O(g)$ ni $g \in O(f)$. Por ejemplo:

$$f(n) = \begin{cases} n^2 & si\ n\ es\ par \\ n^3 & si\ n\ es\ impar \end{cases}$$

y

$$g(n) = \begin{cases} n^3 & si\ n\ es\ par \\ n^2 & si\ n\ es\ impar \end{cases}$$

Propiedad 2.5 *Dadas f y g de N en R^+, $O(f+g) = O(max(f,g))$. ($\Omega(f+g) = \Omega(max(f,g))$)*

demostración:

$f(n) + g(n) \leq 2max(f(n), g(n)) \Rightarrow f + g \in O(max(f,g))$

y $max(f(n), g(n)) \leq f(n) + g(n) \Rightarrow max(f,g) \in O(f+g)$,

por lo que, por la propiedad 2.4 i), se deduce que $O(f+g) = O(max(f,g))$.

De la propiedad 2.5 se deduce que para obtener O de un algoritmo basta obtenerlo de su parte más larga. Pero hay que tener cuidado con los bucles, pues en un buble por el que se pasa un número de veces dependiente del tamaño de la entrada es erróneo deducir que el O del algoritmo es el de su parte más larga.

Propiedad 2.6 *Dadas f y g de N en R^+, $O(f) = O(g) \Leftrightarrow \theta(f) = \theta(g) \Leftrightarrow f \in \theta(g) \Leftrightarrow \Omega(f) = \Omega(g)$.*

demostración:

Si $O(f) = O(g)$, por la propiedad 2.4.i), $\exists b, c \in R^+, \exists n_0 \in N / \forall n \geq n_0 : \ f(n) \leq cg(n)$ y $g(n) \leq bf(n)$. Si $t \in \theta(f)$, por la propiedad 2.1, $\exists d, e \in R^+, \exists n_0 \in N / \forall n \geq n_0 : \ df(n) \leq t(n) \leq ef(n)$, por lo que a partir de un cierto n_0 se cumple que $\frac{d}{b} g(n) \leq t(n) \leq ecg(n)$, y por la propiedad 2.1, $t \in \theta(g)$, luego $\theta(f) \subseteq \theta(g)$. Del mismo modo se demuestra que $\theta(g) \subseteq \theta(f)$.

El resto de las equivalencias se demuestran de manera similar.

Propiedad 2.7 *Dadas f y g de N en R^+, se cumple:*

i) $\lim_{n\to\infty} \frac{f(n)}{g(n)} \in R^+ \Rightarrow O(f) = O(g)$, $\Omega(f) = \Omega(g)$ y $\theta(f) = \theta(g)$.

ii) $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow O(f) \subseteq O(g)$ y $\Omega(g) \subseteq \Omega(f)$, pero f no tiene por qué pertenecer a $\theta(g)$.

iii) $\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty \Rightarrow \Omega(f) \subseteq \Omega(g)$ y $O(g) \subseteq O(f)$, pero f no tiene por qué pertenecer a $\theta(g)$.

demostración:

i) Todas las igualdades son equivalentes por la propiedad 2.1, por lo que basta con demostrar una de ellas.

Si el límite es l, a partir de un cierto n_0 tenemos que $(l - \delta) < \frac{f(n)}{g(n)} < (l + \delta)$ para un $\delta \in R^+$ con $l - \delta > 0$, por lo que $(l - \delta)\, g(n) < f(n) < (l + \delta)\, g(n)$, y, por la propiedad 2.4 se demuestra.

ii) $\exists \delta / f < \delta g$, a partir de un cierto n_0, y esto implica que $f \in O(g)$. De la propiedad 2.4 se deduce.

Como contraejemplo podemos tomar $f(n) = n, \ g(n) = n^2$.

iii) $\forall M, f > Mg$, a partir de un cierto n_0, y esto implica que $f \in \Omega(g)$. De la propiedad 2.4 se deduce.

Como contraejemplo podemos tomar $g(n) = n, \ f(n) = n^2$.

2.2.3 Cotas de complejidad más corrientes

El orden de un polinomio $a_n x^n + \ldots + a_0$, con a_n positivo es $O(x^n)$.

El orden de un sumatorio $\sum_{i=1}^{m} i^n$ es $O(m^{n+1})$.

La medida logarímica aparece cuando se hace una operación para $\frac{n}{2}$, otra para $\frac{n}{4}$, otra para $\frac{n}{8}$, y así sucesivamente. En este caso la complejidad es $O(\log_2 n)$.

Las medidas más frecuentes que aparecen en el estudio de los algoritmos son las polinómicas y logarítmicas y producto de éstas.

Algunas relaciones entre órdenes de complejidad son: $O(1) \subset O(\log n) \subset O(\sqrt{n}) \subset O(n) \subset O(n \log n) \subset O(n \log n \log n) \subset O(n^2) \subset O(n^3) \subset \ldots \subset O(2^n) \subset O(n!) \subset O(n^n)$. En general, con p y q enteros positivos se cumple $O\left((\log n)^p\right) \subset O\left(n^{\frac{1}{q}}\right)$.

Con la notación Ω se invierten las inclusiones.

Todas las inclusiones se pueden demostrar utilizando la propiedad 2.7.

Por ejemplo, podemos demostrar que $O(\log n) \subset O(\sqrt{n})$ calculando el límite:

$$\lim_{n\to\infty} \frac{ln\ n}{\sqrt{n}}$$

ya que $O(ln\ n) = O(\log n)$ al diferenciarse en una constante. Aplicando la regla de l'Hôpital tenemos:

$$\lim_{n\to\infty} \frac{ln\ n}{\sqrt{n}} = \lim_{n\to\infty} \frac{\frac{1}{n}}{\frac{1}{2\sqrt{n}}} = \lim_{n\to\infty} \frac{2}{\sqrt{n}} = 0$$

con lo que queda demostrado lo que queríamos. Del mismo modo se puede demostrar que $O\left((\log n)^2\right) \subset O\left(n^{\frac{1}{2}}\right)$:

$$\lim_{n\to\infty} \frac{ln^2\ n}{\sqrt{n}} = \lim_{n\to\infty} \frac{2ln\ \frac{1}{n}}{\frac{1}{2\sqrt{n}}} = \lim_{n\to\infty} \frac{4ln\ n}{\sqrt{n}} = 0$$

Para demostrar que $O\left(2^n\right) \subset O(n!)$ basta con observar que $\frac{2^n}{n!} \leq 2\frac{2}{n}$ por lo que su límite es cero. En el caso de $O(n!) \subset O\left(n^n\right)$ basta con observar que $\frac{n!}{n^n} \leq \frac{1}{n}$, cuyo límite es también cero.

2.3 Ecuaciones de recurrencia

En el análisis de algoritmos se suele llegar en el cálculo del tiempo de ejecución a ecuaciones de recurrencia que se presentan en general como:

$$t(n) = \begin{cases} b & si\ n \leq n_0 \\ g(t(n), t(n-1), \ldots, t(n-k), n) & si\ n > n_0 \end{cases} \tag{2.3}$$

Ejemplo 2.4 En el algoritmo de las Torres de Hanoi tenemos:

Hanoi(n, i, j, k) /*Paso de n aros de i a j teniendo a k de pivote*/
  si $n = 1$
    mover(i, j)
  en otro caso
    Hanoi($n-1, i, k, j$)
    mover(i, j)
    Hanoi($n-1, k, j, i$)
  finsi

fin

Y el número de movimientos se puede obtener a partir de la fórmula:

$$t(n) = \begin{cases} 1 & si\ n = 1 \\ 2t(n-1)+1 & si\ n > 1 \end{cases}$$

lo que da lugar a una ecuación de recurrencia.

2.3.1 Ecuaciones lineales homogéneas

El caso más sencillo será el de ecuaciones lineales homogéneas de coeficientes constantes:

$$a_0t(n) + a_1t(n-1) + \ldots + a_kt(n-k) = 0 \tag{2.4}$$

donde los coeficientes a_i son constantes. Se llama lineal porque la ecuación es lineal y homogénea porque no tiene término independiente de t.

Buscamos una solución de la forma $t(n) = x^n$ con x constante. De este modo:

$$a_0x^n + a_1x^{n-1} + \ldots + a_kx^{n-k} = 0$$

con lo que habrá que resolver la **ecuación característica**:

$$a_0x^k + a_1x^{k-1} + \ldots + a_k = 0 \tag{2.5}$$

Puede ocurrir que las **soluciones** de la ecuación característica sean todas **distintas** $(s_1, s_2, \ldots, s_k)$, en cuyo caso toda combinación lineal $t(n) = \sum_{i=1}^{k}(c_is_i^n)$ es una solución de la recurrencia, y para determinar los valores de las constantes habrá que resolver el sistema formado imponiendo las condiciones iniciales.

Ejemplo 2.5 Como ejemplo veamos cuál es la solución de la recurrencia $t(n)-3t(n-1)-4t(n-2) = 0$ cuando $n > 2$, con condiciones iniciales $t(0) = 0$ y $t(1) = 1$.

La ecuación característica es $x^2 - 3x - 4 = 0$, que tiene soluciones -1 y 4, por lo que $t(n) = c_1(-1)^n + c_24^n$, e imponiendo las condiciones iniciales tenemos:

$$\begin{array}{l} c_1 + c_2 = 0 \\ -c_1 + 4c_2 = 1 \end{array}$$

de donde $c_1 = -\frac{1}{5}$ y $c_2 = \frac{1}{5}$, con lo que la solución es $t(n) = \frac{1}{5}(4^n - (-1)^n)$.

Puede ocurrir que las **soluciones** de la ecuación característica **no sean simples**.

Si s es una raíz de multiplicidad m, el polinomio característico se descompone como $p(x) = (x-s)^mq(x)$, con lo que todas las derivadas del polinomio $p(x)$ hasta la de grado $m-1$ se anulan para $x = s$.

De este modo, sea $x^{n-k}p(x) = 0$ la ecuación polinómica que representa la ecuación de recurrencia original, su derivada es $(n-k)x^{n-k-1}p(x) + x^{n-k}p'(x)$, que se anula para $x = s$.

Por otro lado, la derivada es $a_0nx^{n-1} + a_1(n-1)x^{n-2} + \ldots + a_k(n-k)x^{n-k-1}$, y con $x = s$ queda $a_0ns^{n-1} + a_1(n-1)s^{n-2} + \ldots + a_k(n-k)s^{n-k-1}$, y multiplicando por s tendremos $a_0ns^n + a_1(n-1)s^{n-1} + \ldots + a_k(n-k)s^{n-k}$, con lo que se ve que $t(n) = ns^n$ es solución de la ecuación de recurrencia.

Del mismo modo se puede comprobar que siendo s una solución de multiplicidad m, $s^n, ns^n, n^2s^n, \ldots, n^{m-1}s^n$ son soluciones de la ecuación de recurrencia.

Ejemplo 2.6 Como ejemplo veamos cuál es la solución de la recurrencia $t(n) = 5t(n-1) - 8t(n-2) + 4t(n-3)$ cuando $n > 3$, con condiciones iniciales $t(0) = 0$, $t(1) = 1$ y $t(2) = 2$.

La ecuación característica es $x^3 - 5x^2 + 8x - 4 = 0$, que se descompone como $(x-1)(x-2)^2 = 0$, por lo que la solución general de la recurrencia es $t(n) = c_11^n + c_22^n + c_3n2^n$, e imponiendo las condiciones iniciales tenemos el sistema de ecuaciones:

$$\begin{aligned} c_1 + c_2 &= 0 \\ c_1 + 2c_2 + 2c_3 &= 1 \\ c_1 + 4c_2 + 8c_3 &= 2 \end{aligned}$$

y resolviendo el sistema tenemos $t(n) = -2 + 2^{n+1} - n2^{n-1}$.

Este ejemplo de resolución de ecuación de recurrencia no puede corresponder a un caso real de cálculo de tiempo de ejecución pues el término de mayor orden en $t(n)$ aparece con coeficiente negativo, lo que corresponde a un tiempo de ejecución negativo cuando el tamaño de la entrada aumenta.

2.3.2 Caso general

Generalizando al caso:

$$a_0t(n) + a_1t(n-1) + \ldots + a_kt(n-k) = b^np(n) \tag{2.6}$$

donde b es una constante y $p(n)$ un polinomio de grado d, la ecuación característica es:

$$\left(a_0x^k + a_1x^{k-1} + \ldots + a_k\right)(x-b)^{d+1} = 0 \tag{2.7}$$

Ejemplo 2.7 Este tipo de ecuaciones también se pueden resolver reduciendo el orden del polinomio $p(n)$.

Si tenemos

$$t(n) - 2t(n-1) = 3^n(n+1) \tag{2.8}$$

multiplicando por 3 tendremos:

$$3t(n) - 6t(n-1) = 3^{n+1}(n+1) \tag{2.9}$$

y sustituyendo en la ecuación 2.8 n por $n+1$ tenemos:

$$t(n+1) - 2t(n) = 3^{n+1}(n+2) \tag{2.10}$$

y restando a 2.10 la ecuación 2.9:

$$t(n+1) - 5t(n) + 6t(n-1) = 3^{n+1} \tag{2.11}$$

con lo que se ha disminuido en uno el grado de $p(n)$. Haciendo lo mismo repetidamente se llega, en un número finito de pasos, a una ecuación homogénea que se resuelve como se vio en la subsección anterior.

Para ecuaciones de la forma:

$$a_0 t(n) + a_1 t(n-1) + \ldots + a_k t(n-k) = b_1^n p_1(n) + b_2^n p_2(n) + \ldots \tag{2.12}$$

se tiene como ecuación característica:

$$\left(a_0 x^k + a_1 x^{k-1} + \ldots + a_k\right)(x-b_1)^{d_1+1}(x-b_2)^{d_2+1}\ldots = 0 \tag{2.13}$$

2.3.3 Cambio de variable

Algunas veces, para resolver ecuaciones de recurrencia, se puede hacer un cambio de variable para transformar la ecuación a uno de los tipos vistos en las subsecciones anteriores.

Ejemplo 2.8 Si

$$t(n) = \begin{cases} a & si\ n = 1 \\ 2t\left(\frac{n}{2}\right) + bn & con\ b\ positivo,\ si\ n > 1,\ y\ n = 2^k \end{cases}$$

haciendo el cambio $n = 2^k$ obtenemos:

$$t\left(2^k\right) = \begin{cases} a & si\ k = 0 \\ 2t\left(2^{k-1}\right) + b2^k & si\ k > 0 \end{cases}$$

La ecuación característica es $(x-2)^2$, por lo que la solución general es de la forma $t_k = c_1 2^k + c_2 k 2^k$ y, deshaciendo el cambio, es $t(n) = c_1 n + c_2 n \log n \in \theta(n \log n)$. Faltaría calcular las constantes o al menos asegurarnos de que c_2 es positiva, pero calculándolas queda $c_1 = a$ y $c_2 = b$.

Como hemos supuesto que n es potencia de dos habría que aplicar el teorema 2.1 para quitar esa restricción.

2.3.4 Transformación de la imagen

Se utiliza en algunos casos para resolver ecuaciones recurrentes no lineales.

Ejemplo 2.9 Si

$$t(n) = \begin{cases} 6 & si\ n = 1 \\ nt^2\left(\frac{n}{2}\right) & si\ n > 1 \end{cases}$$

haciendo el cambio $n = 2^k$ obtenemos:

$$t\left(2^k\right) = \begin{cases} 6 & si\ k = 0 \\ 2^k t^2\left(2^{k-1}\right) & si\ k > 0 \end{cases}$$

y tomando logaritmos:

$$\log t_k = \begin{cases} \log 6 & si\ k = 0 \\ k + 2\log t_{k-1} & si\ k > 0 \end{cases}$$

Se hace una transformación de la imagen $v_k = \log t_k$, con lo que queda:

$$v_k = \begin{cases} \log 6 & si\ k = 0 \\ k + 2v_{k-1} & si\ k > 0 \end{cases}$$

La ecuación característica es $(x-2)(x-1)^2 = 0$, por lo que las soluciones posibles son de la forma $v_k = c_1 2^k + c_2 + c_3 k$. Se calculan c_1, c_2 y c_3 planteando las ecuaciones correspondientes a v_0, v_1 y v_2, con lo que queda $v_k = (3+\log 3)2^k - k - 2 = \log t_k$, y tomando exponentes y simplificando queda que $t(n) = \frac{2^{3n-2}3^n}{n}$.

Como hemos supuesto que n es potencia de dos habría que aplicar el teorema 2.1 para quitar esa restricción.

2.3.5 Técnica de la inducción constructiva

Para resolver algunas ecuaciones de recurrencia se puede utilizar la técnica de inducción, de modo que si suponemos que $t \in \theta(f)$ ($O(f)$ ó $\Omega(f)$) lo podemos demostrar por inducción.

Ejemplo 2.10 Como ejemplo vemos la ecuación de recurrencia:

$$t(n) = \begin{cases} a & si\ n = 1 \\ bn^2 + nt(n-1) & si\ n > 1 \end{cases}$$

Podemos suponer (pues es razonable hacerlo así) que $t(n) \in \theta(n!)$.

Para demostrar esto demostraremos por inducción que $t(n) \in O(n!)$ y que $t(n) \in \Omega(n!)$.

Empezamos demostrando que $t(n) \in \Omega(n!)$:

Está claro que $t(1) \geq v1!$ para un cierto valor constante de v que puede ser a.

Suponemos que $t(n-1) \geq v(n-1)!$. Como $t(n) = bn^2 + nt(n-1) \geq bn^2 + nv(n-1)! \geq vn!$, luego $t(n) \in \Omega(n!)$.

Demostrando ahora que $t(n) \in O(n!)$:

Está claro que $t(1) \leq u1!$ para ciertos valores constantes de u.

Suponemos que $t(n-1) \leq u(n-1)!$. Tenemos $t(n) = bn^2 + nt(n-1) \leq bn^2 + nu(n-1)!$, de donde no podemos deducir que $t(n) \in O(n!)$.

Suponemos que existen enteros positivos u y w tal que $t(n) \leq un! - wn$. Esto se cumplirá para $n = 1$ dependiendo de los valores de u y w.

Suponemos que se cumple para $t(n-1)$, por lo que $t(n) \leq bn^2 + n\,(u(n-1)! - w(n-1)) = un! + ((b-w)n + w)\,n$, y para que esta expresión sea menor o igual que $un! - wn$ tiene que ser $(b-w)n + w \leq -w$, de donde tiene que ser $w \geq b\frac{n}{n-2}$. Tomando $w = 2b$ se satisface la desigualdad y se cumple la recursión, y tomando $a = u - 2b$ se cumplirá también el caso base.

Lo difícil con este método puede ser acertar con la función.

2.4 Generalización de las notaciones

2.4.1 Notaciones condicionales

Algunos algoritmos son más fáciles de estudiar cuando se consideran tamaños en la entrada dentro de un cierto conjunto como puede ser el de las potencias de 2. Para estos casos se pueden utilizar notaciones asintóticas condicionales. La ventaja de utilizar estas notaciones es que generalmente se puede eliminar la condición una vez estudiado el algoritmo.

Definición 2.5 *Dada una función $f : N \longrightarrow R^+$ y $P : N \longrightarrow \{true, false\}$ definimos el* **orden de f según P** *como:*

$$O(f|P) = \{t : N \longrightarrow R^+ / \exists c \in R^+, \exists n_0 \in N, \forall n \geq n_0 :\ P(n) \Rightarrow t(n) \leq cf(n)\}$$

Se pueden definir de manera similar $\Omega(f|P)$ y $\theta(f|P)$.

En algunos casos puede interesar restringirnos a valores de n que cumplan una cierta condición y posteriormente quitar la restricción utilizando el siguiente teorema:

Teorema 2.1 *Si $b \geq 2$ es un entero y $f : N \longrightarrow R^+$ una función que es no decreciente a partir de un valor n_0 (***eventualmente no decreciente***) y $f(bn) \in O(f(n))$ (f es* **b-armónica***), y $t : N \longrightarrow R^+$ es eventualmente no decreciente tal que $t(n) \in \theta(f|n$ potencia de $b)$, entonces $t(n) \in \theta(f)$.*

Ejemplo 2.11 Si

$$t(n) = \begin{cases} a & si\ n = 1 \\ t\left(\lfloor \frac{n}{2} \rfloor\right) + t\left(\lceil \frac{n}{2} \rceil\right) + bn & con\ b\ positivo,\ si\ n > 1 \end{cases}$$

donde $\lfloor\ \rfloor$ representa al entero más próximo menor y $\lceil\ \rceil$ al entero más próximo mayor.

Será más fácil resolverlo suponiendo que n es una potencia de 2, pues en este caso tenemos:

$$t(n) = \begin{cases} a & si\ n = 1 \\ 2t\left(\frac{n}{2}\right) + bn & con\ b\ positivo,\ si\ n > 1,\ y\ n = 2^k \end{cases}$$

Podemos calcular su orden del siguiente modo:

si $n = 2^k$ tenemos $t\left(2^k\right) = 2t\left(2^{k-1}\right) + b2^k = 2\left(2t\left(2^{k-2}\right) + b2^{k-1}\right) + b2^k = 2^2t\left(2^{k-2}\right) + 2b2^k = \ldots = an + bn\log n$ que es de orden $\theta(n \log n | n\ es\ potencia\ de\ 2)$.

A partir de esto, y utilizando el teorema 2.1, se puede calcular el orden exacto del tiempo de ejecución en el caso general: sólo hay que demostrar que t y $f(n) = n \log n$ son no decrecientes y que f es 2-armónica.

f es 2-armónica pues $f(2n) = 2n \log 2n = 2n \log n + 2n \in \theta(n \log n)$. Y además es eventualmente no decreciente pues $n \log n$ es creciente en todo su dominio.

Para demostrar que t es eventualmente no decreciente tenemos el problema de que sólo conocemos t por su ecuación de recurrencia. Se puede demostrar por inducción:

$t(2) = 2t(1) + b > t(1)$,

y suponiendo que es eventualmente no decreciente para valores menores o iguales a n demostraremos que $t(n+1) \geq t(n)$:

si n es par tenemos $\lfloor \frac{n}{2} \rfloor = \lfloor \frac{n+1}{2} \rfloor$ y $\lceil \frac{n}{2} \rceil + 1 = \lceil \frac{n+1}{2} \rceil$, por lo que $t(n+1) \geq t(n)$. Y similarmente se demuestra la desigualdad para n impar.

2.4.2 Notaciones con varios parámetros

Puede ocurrir que tengamos un algoritmo en el que el tiempo de ejecución dependa de más de un parámetro. En este caso se necesitan notaciones asintóticas con varios parámetros, por ejemplo:

Definición 2.6 *Dada una función $f : N^m \longrightarrow R^+$, llamamos* **orden de** *f al conjunto de todas las funciones de N^m en R^+ acotadas superiormente por un múltiplo real positivo de f para valores de $(n_1, n_2, \ldots, n_m)$ suficientemente grandes. Se denota $O(f)$, y será:*

$$O(f) = \{t : N^m \longrightarrow R^+ / \exists c \in R^+, \exists n_1, n_2, \ldots, n_m \in N,$$

$$\forall k_1 \geq n_1, k_2 \geq n_2, \ldots, k_m \geq n_m : t(k_1, k_2, \ldots, k_m) \leq cf(k_1, k_2, \ldots, k_m)\}$$

2.5 Problemas

Problema 2.1 Obtener O y Ω para el algoritmo de multiplicación de matrices:

```
FOR i = 1 TO n
    FOR j = 1 TO n
        suma = 0
        FOR k = 1 TO n
            suma = suma + a[i,k]b[k,j]
        ENDFOR
        c[i,j] = suma
    ENDFOR
ENDFOR
```

Solución:

En este caso siempre se ejecutan las mismas instrucciones independientemente de la entrada, por lo que $O(f) = \Omega(f) = \theta(f)$.

Tenemos:

$$
\begin{aligned}
c_1 + \sum_{i=1}^{n}\left(c_2 + \sum_{j=1}^{n}\left(c_3 + \sum_{k=1}^{n} c_4\right)\right) &= \\
c_1 + \sum_{i=1}^{n}\left(c_2 + \sum_{j=1}^{n}(c_3 + c_4 n)\right) &= \\
c_1 + \sum_{i=1}^{n}\left(c_2 + c_3 n + c_4 n^2\right) &= \\
c_1 + c_2 n + c_3 n^2 + c_4 n^3
\end{aligned}
$$

por lo que $t \in \theta(n^3)$.

Problema 2.2 Obtener O y Ω, y θ del tiempo promedio para el algoritmo de búsqueda binaria:

```
i = 1
j = n
REPEAT
    m = (i + j) DIV 2
    IF a[m] > x
        j = m - 1
    ELSE
        i = m + 1
    ENDIF
UNTIL i > j OR a[m] = x
```

Solución:

El caso más favorable será cuando en la primera comparación de $a[m]$ con x sean iguales. Se habrán ejecutado 6 instrucciones, por lo que $t \in \Omega(1)$.

El caso más desfavorable será cuando x no esté en el array (estamos suponiendo el array ordenado). Si n es una potencia de 2 tenemos que $t(n) = 2 + 4 + t_r\left(\frac{n}{2}\right) = 2 + 4 + 4 + t_r\left(\frac{n}{4}\right) = \ldots = 2 + 4(\log n + 1)$, con lo que $t \in O(\log n | n \textit{ potencia de } 2)$.

Para quitar la restricción de que n sea potencia de 2 se aplicará el teorema 2.1. $\log n$ es creciente y 2-armónica, pues $\log 2n = 1 + \log n \in O(\log n)$. Además, $t(n) = 6 + t_r\left(\lfloor\frac{n-1}{2}\rfloor\right)$ ó $t(n) = 6 + t_r\left(\lfloor\frac{n}{2}\rfloor\right)$, lo que da una función eventualmente no decreciente pues si es creciente para valores menores que n también será $t(n+1) \geq t(n)$ como se ve fácilmente sustituyendo n por $n+1$ en la fórmula de t.

Para calcular el tiempo promedio supondremos que el elemento está en el array con probabilidad p (por tanto, no está en el array con probabilidad $1-p$). Las dos primeras instrucciones se ejecutan siempre, después se ejecutan 4 instrucciones por cada paso por el bucle. El número de pasos por el bucle es lo que varía según la posición donde esté el elemento: puede encontrarse a la primera con probabilidad $\frac{p}{n}$ pues esto ocurre cuando es igual al primer elemento con que se compara; se puede encontrar a la segunda si es el elemento en la posición $\frac{n}{4}$ o $\frac{3n}{4}$, lo que ocurre con probabilidad $\frac{2p}{n}$; en general, se encuentra en el paso i con probabilidad $\frac{2^{i-1}p}{n}$. Por tanto el tiempo promedio será:

$$2 + 4\left(\frac{p}{n} + 2\frac{2p}{n} + 3\frac{4p}{n} + \ldots + \log n \frac{2^{\log n - 1}p}{n}\right) = 2 + 4\frac{p}{n}\sum_{i=1}^{\log n} i2^{i-1} \simeq 2 + 2\frac{p}{n}\int_0^{\log n} x2^x \, dx$$

y resolviendo la integral por partes tendremos una $o\left(\frac{2}{ln\ 2} p \log n\right)$.

Problema 2.3 Demostrar que siendo $a, b, b' \in R^+$ y $c \in N$ con $c > 0$ y

$$f(n) = \begin{cases} b' & si\ n \leq n_0 \\ af(n-c) + b & si\ n > n_0 \end{cases}$$

$a < 1 \Rightarrow f \in O(1)$,
$a = 1 \Rightarrow f \in O(n)$,
$a > 1 \Rightarrow f \in O\left(a^{\frac{n}{c}}\right)$.

Solución:

$f(n) = b + af(n-c) = b + ab + a^2 f(n-2c) = \ldots = b\left(1 + a + a^2 + \ldots + a^{k-1}\right) + a^k f(n-kc)$, con k el menor entero tal que $n - kc \leq n_0$.

Si $a < 1$:
$f(n) = \frac{b\left(1-a^k\right)}{1-a} + a^k b' < \frac{b}{1-a} + b'$, con lo que $f \in O(1)$.

Si $a = 1$:
$f(n) = bk + b'$, y al ser $n - kc \geq 0$ es $k \leq \frac{n}{c}$, y $f \in O(n)$.

Si $a > 1$:
al ser k el menor entero tal que $n - kc \leq n_0$, tendremos $n - (k-1)c > n_0 \Rightarrow n - n_0 + c > kc \Rightarrow \frac{n-n_0}{c} + 1 > k \Rightarrow f(n) = b\frac{\left(a^k-1\right)}{a-1} + a^k b' \leq b\frac{\left(a^{\frac{n-n_0}{c}+1}-1\right)}{a-1} + a^{\frac{n-n_0}{c}+1}b'$, y $f \in O\left(a^{\frac{n}{c}}\right)$.

Problema 2.4 Demostrar que siendo $a, b, b' \in R^+$, $c \in N$ con $c > 0$ y $d \in R$ y:

$$f(n) = \begin{cases} b' & si\ 0 \leq n \leq n_0 \\ af(n-c) + bn + d & si\ n > n_0 \end{cases}$$

$a < 1 \Rightarrow f \in O(n)$,
$a = 1 \Rightarrow f \in O(n^2)$,
$a > 1 \Rightarrow f \in O\left(a^{\frac{n}{c}}\right)$.

Solución:

$$\begin{gathered} f(n) = d + bn + af(n-c) = d + bn + ad + ab(n-c) + a^2 f(n-2c) = \\ d\left(1 + a + a^2\right) + b\left(n + a(n-c) + a^2(n-2c)\right) + a^3 f(n-3c) = \ldots = \\ d\left(1 + a + \ldots + a^{k-1}\right) + b\left(n + a(n-c) + \ldots + a^{k-1}(n-(k-1)c)\right) + a^k b' \end{gathered}$$

El término que multiplica a b se puede poner como

$$n\left(1 + a + \ldots + a^{k-1}\right) - ca\left(1 + 2a + \ldots + (k-1)a^{k-2}\right)$$

y el término que multiplica a ca es la derivada de $a + a^2 + \ldots + a^{k-1}$, cuya suma es $\frac{a^k-a}{a-1}$, nos queda que:

$$f(n) = d\frac{1-a^k}{1-a} + bn\frac{1-a^k}{1-a} - bca\frac{1 + ka^{k-1}(a-1) - a^k}{(a-1)^2} + a^k b'$$

Y sustituyendo para los distintos valores de $a > 1$ y $a < 1$ obtenemos el resultado, tal como en el problema 2.3.

Con $a = 1$ no podemos utilizar la fórmula pues el denominador es 0. Pero en este caso la fórmula es $f(n) = dk + bnk - bc\frac{k+1}{2}k + b' \leq d\left(\frac{n-n_0}{c} + 1\right) + bn\left(\frac{n-n_0}{c} + 1\right) \in O\left(n^2\right)$.

Problema 2.5 Dados a, b, b' y c como en el problema 2.3, con $c > 1$, y:

$$f(n) = \begin{cases} b' & si\ 1 \leq n \leq n_0 \\ af\left(\frac{n}{c}\right) + b & si\ n > n_0 \end{cases}$$

demostrar que:
$a < 1 \Rightarrow f \in O(1)$,

$a = 1 \Rightarrow f \in O(\log_c n)$,
$a > 1 \Rightarrow f \in O\left(n^{\log_c a}\right)$.

Problema 2.6 Dados a, b, b' y c como en el problema 2.5, y $d \in R$, y:

$$f(n) = \begin{cases} b' & si\ 1 \leq n \leq n_0 \\ af\left(\frac{n}{c}\right) + bn + d & si\ n > n_0 \end{cases}$$

demostrar que:
$a < c \Rightarrow f \in O(n)$,
$a = c \Rightarrow f \in O(n \log_c n)$,
$a > c \Rightarrow f \in O\left(n^{\log_c a}\right)$.

Problema 2.7 Siendo b y k dos constantes positivas, $b, k \in N^+$, y siendo $a, c, p \in R^+$, de forma que $a \geq 1$, $b \geq 2$, y dada la siguiente ecuación:

$$t(n) = \begin{cases} at\left(\frac{n}{b}\right) + cn^k & si\ n > n_0 \\ p & si\ 0 \leq n \leq n_0 \end{cases}$$

obtener, desarrollando la fórmula mediente sucesivas sustituciones, el orden de $t(n)$ dependiendo de la relación entre a y b^k.

Solución:

$$t(n) = at\left(\frac{n}{b}\right) + cn^k = a\left(at\left(\frac{n}{b^2}\right) + c\left(\frac{n}{b}\right)^k\right) + cn^k = a^2t\left(\frac{n}{b^2}\right) + cn^k\left(1 + \frac{a}{b^k}\right) = \ldots$$
$$= a^m t\left(\frac{n}{b^m}\right) + cn^k\left(1 + \frac{a}{b^k} + \left(\frac{a}{b^k}\right)^2 + \ldots + \left(\frac{a}{b^k}\right)^{m-1}\right)$$

con m el menor tal que $\frac{n}{b^m} \leq n_0$, con lo que $\frac{n}{b^{m-1}} > n_0 \Rightarrow \log_b \frac{n}{n_0} > m - 1 \Rightarrow \log_b n - \log_b n_0 + 1 > m$.

Si $a = b^k$ queda:

$$a^m p + cn^k m \leq a^{\log_b n - \log_b n_0 + 1} p + cn^k \left(\log_b n - \log_b n_0\right)$$

y comparando los términos en n, tenemos $a^{\log_b n} = n^{\log_b a} = n^k$ y $n^k \log_b n$. Con lo que el orden es $O(n^k \log n)$.

Si $a < b^k$, la función se puede poner en la forma:

$$a^m p + cn^k \frac{b^{km} - a^m}{b^k - a} \frac{b^k}{b^{km}}$$

y teniendo en cuenta que $\log_b n - \log_b n_0 + 1 > m$, y comparando los términos en n, tendríamos los términos $n^{\log_b a}$ y n^k que, al ser $\log_b a < k$, da un orden $O\left(n^k\right)$.

Si $a > b^k$ se razona de manera similar, siendo en este caso los términos en n: $n^{\log_b a}$ y $n^k \left(\frac{a}{b^k}\right)^{\log_b n} = n^{\log_b a}$, y el orden $O\left(n^{\log_b a}\right)$.

Problema 2.8 Calcular el tiempo de ejecución y el orden exacto del siguiente algoritmo:

```
p = 0
for i = 1 to n
    p = p + i * i
    for j = 1 to p
        writeint(a[p, j])
    endfor
endfor
```

Solución:

Con $i = 1$, $p = 1$, y j varía entre 1 y 1,
con $i = 2$, $p = 1 + 2^2$, y j varía entre 1 y $1 + 2^2$,
con $i = 3$, j varía entre 1 y $1 + 2^2 + 3^2$,
y en general, para un cierto i, j varía entre 1 y $1 + 2^2 + 3^2 + ... + i^2$.

Supondremos que todas las operaciones elementales consumen un tiempo unidad. j toma el valor 1 n veces, los siguientes valores, del 2 al $1+2^2$, $n-1$ veces, los siguientes valores $n-2$ veces, ... De este modo, el número de operaciones elementales es

$$n + (n-1)2^2 + (n-2)3^2 + ... + (n-(n-1))n^2 = \sum_{i=0}^{n-1}(n-i)(i+1)^2 =$$
$$n\sum_{i=0}^{n-1}(i+1)^2 - \sum_{i=0}^{n-1} i(i+1)^2 \simeq n\int_0^n \left(i^2+2i+1\right)\, di \, - \int_0^n \left(i^3+2i^2+i\right)\, di =$$
$$\frac{n^4}{12} + \frac{n^3}{3} + \frac{n^2}{2}$$

y el orden exacto es $\theta\left(n^4\right)$.

Problema 2.9 Resolver la ecuación recurrente $t_n - 2t_{n-1} - 1 = 0$, con $n \geq 1$, y $t_0 = 0$.

Solución:

La ecuación característica es $(x-1)(x-2) = 0$, con lo que la solución general será de la forma $t(n) = c_1 2^n + c_2$.

Obtenemos c_1 y c_2 con:

$$t(0) = 0 = c_1 + c_2$$
$$t(1) = 2t(0) + 1 = 1 = 2c_1 + c_2$$

de donde $c_1 = 1$ y $c_2 = -1$.

Problema 2.10 Resolver la ecuación recurrente $t_n - 2t_{n-1} = n + 2^n$, con $n \geq 1$, y $t_0 = 0$.

Solución:

La ecuación característica es $(x-2)(x-1)^2(x-2) = 0$, con lo que la solución general será de la forma $t(n) = c_1 + c_2 n + c_3 2^n + c_4 n 2^n$.

Obtenemos c_1, c_2, c_3 y c_4 con:

$$\begin{aligned} t(0) = 0 &= c_1 + c_3 \\ t(1) = 2t(0) + 3 = 3 &= c_1 + c_2 + 2c_3 + 2c_4 \\ t(2) = 2t(1) + 6 = 12 &= c_1 + 2c_2 + 4c_3 + 8c_4 \\ t(3) = 2t(2) + 11 = 35 &= c_1 + 3c_2 + 8c_3 + 24c_4 \end{aligned}$$

de donde $c_1 = -2$, $c_2 = -1$, $c_3 = 2$ y $c_4 = 1$.

Problema 2.11 Resolver la ecuación recurrente $t(n) = at\left(\frac{n}{b}\right) + cn^k$, si $n > n_0$, siendo $b \geq 2$, $n_0 \geq 1$, $k \geq 0$, a y c reales positivos y n una potencia de b.

Solución:

Hacemos el cambio $m = \log_b n$, con lo que queda $t_m - at_{m-1} = cb^{km}$.

La ecuación característica es $(x-a)(x-b^k) = 0$, con lo que la solución general será de la forma $t(n) = c_1 a^m + c_2 b^{km}$.

Deshaciendo el cambio queda:

$$t(n) = c_1 a^{\log_b n} + c_2 b^{k \log_b n} = c_1 n^{\log_b a} + c_2 n^k$$

Faltaría calcular c_1 y c_2, pero para esto necesitaríamos algún caso base.

Problema 2.12 Calcular el tiempo de ejecución de la siguiente función. Escribir también el valor final de la función.

PROCEDURE recursiva(n:INTEGER):INTEGER
  IF $n = 1$
    $recursiva = 1$
  ELSE
    $recursiva =$ 2recursiva$(n-1)$
  ENDIF
ENDrecursiva

Solución:

Sea a el tiempo constante debido a la comprobación de la condición $n = 1$, b el tiempo constante de la asignación $recursiva = 1$, c el tiempo constante de la llamada a recursiva con $n-1$ (asignación y liberación de memoria), la multiplicación del resultado por 2, y la asignación a $recursiva$.

De este modo:

$$t(n) = \begin{cases} a+b & si\ n=1 \\ a+c+t(n-1) & si\ n>1 \end{cases}$$

Si $n \leq 0$ no tiene sentido la llamada a la función pues no acabaría sino al producirse un error.

Desarrollando:

$$t(n) = a+c+a+c+t(n-2) = \ldots = (a+c)(n-1)+t(1) = (a+c)(n-1)+a+b$$

por lo que $t \in \theta(n)$.

El valor final de recursiva(n) es 2^{n-1}, lo que se demuestra por inducción:
recursiva(1)=1,
y si recursiva(n)= 2^{n-1}, recursiva($n+1$)= $2\ 2^{n-1} = 2^n$.

Problema 2.13 Calcular el tiempo de ejecución, en función del valor de n, del siguiente algoritmo.

```
PROCEDURE algoritmo(x:ARRAY[1..MAX,1..MAX] OF tipo;n:INTEGER)
VAR
   i, j:INTEGER
   a, b, c:ARRAY[1..MAX,1..MAX] OF tipo
BEGIN
   IF n > 1
      FOR i = 1 TO n/2
         FOR j = 1 TO n/2
            a[i, j] = x[i, j]
            b[i, j] = x[i, j + n/2]
            c[i, j] = x[i + n/2, j]
         ENDFOR
      ENDFOR
      algoritmo(a, n/2)
      algoritmo(b, n/2)
      algoritmo(c, n/2)
   ENDIF
ENDalgoritmo
```

(la división es división entera).

Solución:

Sea a el tiempo de comprobación de la condición $n > 1$, b el tiempo de una asignación en un FOR, c el tiempo de comprobación de si el índice del FOR sobrepasa el límite superior, d el tiempo de acceso a $x[i, j]$ y asignación (suponemos que las tres asignaciones que aparecen de elementos del array tardan el mismo tiempo d), e el tiempo en la llamada y regreso del procedimiento algoritmo.

De este modo:

$$t(n) = \begin{cases} a & si\ n=1 \\ a+\sum_{i=1}^{\lfloor\frac{n}{2}\rfloor}\left(b+c+\sum_{j=1}^{\lfloor\frac{n}{2}\rfloor}(3d+b+c)\right)+3e+3t\left(\lfloor\frac{n}{2}\rfloor\right) & si\ n>1 \end{cases}$$

Si $\frac{n}{2}$ >MAX suponemos que se produce error y no consideramos este caso. Supondremos para calcular el tiempo que n es potencia de 2 (posteriormente se puede quitar esta restricción).

Consideraremos $t(n)=k_1+k_2n+k_3n^2+3t\left(\frac{n}{2}\right)$, siendo $k_1=a+3e$, $k_2=\frac{b+c}{2}$ y $k_3=\frac{3d+b+c}{4}$.

Desarrollando se llega a:

$$t(n)=k_1\frac{n^{\log 3}-1}{2}+2k_2\left(n^{\log 3}-n\right)-4k_3\left(n^{\log 3}-n^2\right)+n^{\log 3}a\in\theta(n^2)$$

Problema 2.14 Calcular el tiempo de ejecución de:

```
PROCEDURE uno(n:INTEGER)
VAR
   i, j, x, y:INTEGER
BEGIN
   x = 0
   y = 0
   FOR i = 1 TO n
      IF par(i)
         FOR j = i TO n
            x = x + 1
         ENDFOR
      ELSE
         FOR j = 1 TO i − 1
            y = y + 1
         ENDFOR
      ENDIF
   ENDFOR
ENDuno
```

Solución:

Las asignaciones $x=0$ y $y=0$ consumen un tiempo constante independientemente del valor de n. Lo llamamos a.

El FOR consume un tiempo $\sum_{i=1}^{n}(b+t(i))$, donde $t(i)$ es el tiempo consumido en el cuerpo del FOR para un valor determinado de i, y b es el tiempo correspondiente a la actualización de i y a la comprobación de si se ha llegado al final del FOR.

En la instrucción IF se produce siempre la comprobación par(i), luego nos queda $\sum_{i=1}^{n}(b+c+t_{if}(i))$, donde t_{if} depende de que i sea par o impar.

Si i es par $t_{if}(i)=\sum_{j=i}^{n} d$, y si es impar $t_{if}(i)=\sum_{j=1}^{i-1} d$.

El tiempo de ejecución del procedimiento es:

si n es par:

$$t(n) = a + \sum_{i=1}^{n}(b+c+t_{if}(i)) =$$
$$a+(b+c)n+\sum_{i=1}^{\frac{n}{2}}(t_{if}(2i-1)+t_{if}(2i)) =$$
$$a+(b+c)n+\sum_{i=1}^{\frac{n}{2}}\left(\sum_{j=1}^{2i-1} d + \sum_{j=2i}^{n} d\right) =$$
$$a+(b+c)n+d\frac{n^2}{2}$$

y en el caso impar se procede de modo análogo quedando

$$t(n) = a+(b+c)n+d\frac{n^2-1}{2}$$

con lo que $t \in O\left(n^2\right)$.

Problema 2.15 Calcular una buena cota superior de la complejidad del algoritmo:

```
PROCEDURE dos(a:ARRAY[1..n] OF INTEGER;x, y:INTEGER)
   IF x ≠ y
      IF 2a[x] < a[y]
         dos(a, x, y DIV 2)
      ELSE
         dos(a, 2x, y)
      ENDIF
   ENDIF
ENDdos
```

suponiendo que los datos en el array están ordenados de manera creciente.

Solución:

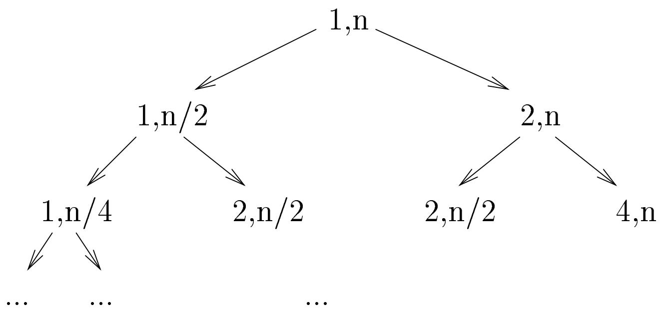
El tiempo de ejecución depende en este caso del valor de n, de los elementos del array y de los valores iniciales de x e y.

Si $x=y$ habrá un tiempo constante.

Si $x>y$, $2a[x]>a[y]$ pues los elementos están en el array de manera creciente, luego se ejecuta dos$(a,2x,y)$ y no para la ejecución.

Consideraremos el caso $x<y$ y que $x=1$, $y=n$, que es cuando mayor tiempo de ejecución tendremos. Además, si n no es potencia de dos dará error debido a que

$x > y$ en algún momento de la ejecución. Suponemos por tanto que $n = 2^k$, con lo que tendremos el siguiente árbol de posibles llamadas:



y a nivel k tendremos $1, \frac{n}{2^k}$; $2, \frac{n}{2^{k-1}}$; ...; $2^k, n$; siendo en cualquiera de ellos $x = y$, con lo que el tiempo es $t \in \theta(\log n)$.

Problema 2.16 Hacer un algoritmo recursivo para el cálculo del n-simo número de Fibonacci y calcular su tiempo de ejecución y la memoria ocupada.

Solución:

Los números de Fibonacci se generan de la forma $F(n) = F(n-1) + F(n-2)$ teniendo como casos base $F(1) = F(2) = 1$.

Un algoritmo recursivo evidente es:

```
PROCEDURE Fibonacci(n:INTEGER):INTEGER
   IF n = 1 OR n = 2
      RETURN 1
   ELSE
      RETURN(Fibonacci(n − 1)+Fibonacci(n − 2))
   END
ENDFibonacci
```

donde

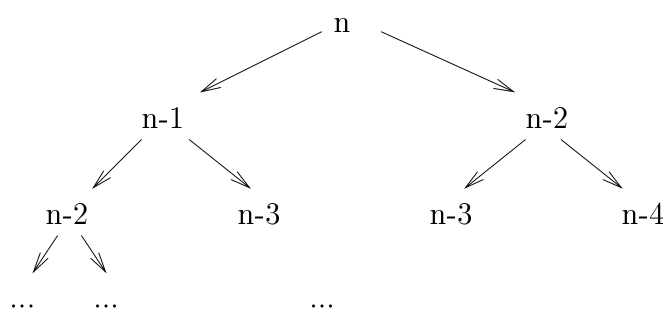
$$t(n) = \begin{cases} a & si\ n = 1,\ n = 2 \\ b + t(n-1) + t(n-2) & si\ n > 2 \end{cases}$$

Resolviendo la ecuación de recurrencia tenemos $(x^2 - x - 1)(x - 1) = 0$, y la solución general tiene la forma:

$$t(n) = c_1 + c_2 \left(\frac{1+\sqrt{5}}{2}\right)^n + c_3 \left(\frac{1-\sqrt{5}}{2}\right)^n$$

Habría que calcular los valores de c_1, c_2 y c_3 en función de a y b imponiendo las condiciones iniciales $t(2)$, $t(3)$ y $t(4)$. El orden exacto será $\theta\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$.

Para obtener la complejidad de la memoria ocupada, consideramos el árbol de llamadas recursivas:



obteniéndose la máxima profundidad para la sucesión de llamadas con tamas $n, n-1, \ldots, 2$, por lo que en el momento de máxima ocupación de memoria tenemos $n-1$ bloques de memoria ocupados y la complejidad es $O(n)$.

Problema 2.17 Dado el programa:

```
i = 1
while i ≤ n
    if a[i] ≥ a[n]
        a[n] = a[i]
    endif
    i = i * 2
endwhile
```

Calcular:

a) Su orden exacto.

b) El número promedio de asignaciones del array.

Solución:

a)

Para calcular el orden exacto calcularemos el orden y el omega.

El caso más desfavorable es cuando hay que hacer siempre la asignación de elementos del array, y el más favorable cuando esta asignación no se hace nunca.

El tiempo de ejecución será de la forma: $t(n) = 1 + t_w(1, n)$, siendo $t_w(i, j)$ el tiempo del paso por el while con índices i y j (en nuestro caso j será siempre n). Y $t_w(1, n) = 4 + t_w(2, n)$ en el caso más desfavorable, y en el caso más favorable será $t_w(1, n) = 3 + t_w(2, n)$, por lo que la única diferencia entre los dos casos será debida a la constante 4 ó 3 que aparece. Desarrollamos el caso más desfavorable y el más favorable se hará de forma similar:

$$t_w(1, n) = 4 + t_w(2, n) = 4 * 2 + t_w\left(2^2, n\right) = \ldots = 4 \log n + 1$$

con lo que $t(n) \in O(\log n | n \textit{ es potencia de } 2)$. Como del mismo modo se obtendría $t(n) \in \Omega(\log n | n \textit{ es potencia de } 2)$, tenemos que $t(n) \in \theta(\log n | n \textit{ es potencia de } 2)$.

Para quitar la restricción aplicamos el teorema 2.1:

$f(n) = \log n$ es creciente en su dominio y es 2-armónica pues $f(2n) = \log 2n = 1 + \log n \in \theta(\log n)$.

Para comprobar que $t(n)$ es eventualmente no decreciente basta comprobar que $t(n) \leq t(n+1)$. Si $2^k \leq n < n+1 < 2^{k+1}$, $t(n) = t(n+1)$ pues en los dos casos se da el mismo número de pasos. Y si $2^k \leq n < n+1 = 2^{k+1}$, será $t(n+1) > t(n)$ pues se da un paso más.

b)

Hay que calcular la probabilidad con que se ejecuta cada una de las asignaciones $a[n] = a[i]$. Con $i = 1$, y si suponemos inicialmente una distribución uniforme de los datos, tendremos que $a[i] \geq a[n]$ con probabilidad $\frac{1}{2}$, y se ejecutará $a[n] = a[1]$ con la misma probabilidad. Tras el paso con $i = 1$ tenemos en $a[n]$ el mayor de los valores iniciales de $a[1]$ y $a[n]$. Con $i = 2$, al comparar $a[2]$ y $a[n]$ estamos comprobando si $a[2]$ es el mayor de los valores iniciales de $a[1]$, $a[2]$ y $a[n]$, con lo que la probabilidad es de $\frac{1}{3}$. En general, para un cierto $i = 2^k$, la asignación se ejecutará con probabilidad $\frac{1}{k+2}$. Por tanto, el tiempo promedio será:

$$t(n) = \sum_{i=2}^{logn+1} \frac{1}{i} \simeq \int_1^{\log n+1} \frac{1}{i} di = ln\ i]_1^{\log n+1} = ln\ (\log n + 1) \in \theta(\log \log n)$$

Hemos considerado que n es potencia de dos, y la restricción se quitaría como en el apartado a).

Problema 2.18 Dado el siguiente procedimiento:

contar(izq,der:indices):integer
  if $izq \geq der$
    return 0
  elsif $a[izq] < a[der]$
    return(contar($izq * 2$,der)+1)
  else
    return(contar($izq * 2$,der))
  endif

donde a es un array global de valores enteros. Calcular, para valores iniciales de $izq = 1$ y $der = n$, el orden exacto del tiempo de ejecución en el caso más favorable, más desfavorable y promedio.

Solución:

Tenemos como caso base que $izq \geq der$, por lo que, con valores iniciales de $izq = 1$ y $der = n$ llegaremos al caso base cuando $izq \geq n$, ya que, cuando no estamos en el caso base se hacen llamadas recursivas a contar($izq * 2$,der), con lo que aumenta el índice izquierdo pero el derecho no varía. El tiempo de este caso base es $t(izq, der) = a$, cuando $izq \geq der$, con a constante correspondiente a la primera comparación y el primer return.

El caso más favorable se dará cuando todas las llamadas recursivas se ejecuten sin sumarle después 1, y el más desfavorable cuando después de la llamada se sume 1 siempre. Por tanto, el caso más favorable será que $a[2^i] \geq a[n]$, $\forall i$, y el más desfavorable que $a[2^i] < a[n]$, $\forall i$. De este modo, el tiempo en el caso más favorable y más desfavorable es $t(izq, der) = t(izq * 2, der) + c$, siendo c distinto para los dos casos, ya que en el caso más desfavorable, además de hacer la llamada recursiva, hay que sumar uno al resultado. Expandiendo la recurrencia, con los valores iniciales de $izq = 1$ y $der = n$, y suponiendo que $n = 2^k$, tenemos:

$$t(1, n) = t(2, n) + c = t(4, n) + 2c = ... = t(n, n) + kc = a + c \log n \in \theta(\log n | n = 2^k)$$

Falta quitar la restricción de que n sea potencia de 2:

$\log n$ es 2-armónica y creciente, y hay que demostrar que la función tiempo de ejecución que estamos calculando es eventualmente no decreciente, y para esto consideramos $n < m$ y demostramos que $t(n) \leq t(m)$:

si $2^k < n < m \leq 2^{k+1}$ en los dos casos se harán $k + 1$ llamadas, con lo que $t(n) = t(m)$,

y si $n \leq 2^k < m < 2^{k+1}$, con tamaño n se hará un máximo de k llamadas, y con tamaño m se harán $k + 1$ llamadas, con lo que $t(n) < t(m)$.

Hemos demostrado que se cumplen las condiciones del teorema por el que se puede quitar la restricción de que n sea potencia de 2, por lo que $t(n) \in \theta(\log n)$ en el caso más favorable y en el más desfavorable.

Lo anterior asegura además que el tiempo de ejecución crece en la forma $\log n$ en todos los casos, ya que el tiempo más desfavorable nos da una cota superior y el más favorable una inferior, y estas dos cotas son iguales. Por tanto, el tiempo promedio también será del orden exacto de $\log n$. Esto se puede también demostrar considerando que $a[izq] < a[der]$ con probabilidad $\frac{1}{2}$, con lo que para el tiempo promedio tenemos la recurrencia:

$$t_p(1, n) = \frac{1}{2} (t_p(2, n) + c_1) + \frac{1}{2} (t_p(2, n) + c_2) = t_p(2, n) + \frac{c_1 + c_2}{2} = t_p(2, n) + c$$

donde los valores c_1 y c_2 corresponden a las constantes en el caso más favorable y más desfavorable, y c es otra constante, por lo que la expansión de la recurrencia nos llevará al resultado final $a + c \log n$, como en los casos antes estudiados.

Problema 2.19 Tenemos claves formadas por una serie no limitada de campos ($campo_1$, $campo_2$, ...), y sabemos que la probabilidad de que el contenido del campo i-ésimo sea igual en dos claves es $\frac{1}{i+1}$. Calcular el orden exacto del tiempo promedio de la comparación de dos claves.

Solución:

Dos claves se compararán comparando el $campo_1$, si son iguales comparando el $campo_2$, ... y por tanto la comparación acabará cuando un campo sea distinto en las dos claves. El número máximo de comparaciones será igual al número de campos de la clave con menos campos, y llamaremos a este número n. Como queremos calcular el orden exacto del tiempo promedio de comparación de dos claves tenemos que contar el número promedio de comparaciones, independientemente del coste de éstas, y para calcular el tiempo promedio habría que multiplicar el número promedio de comparaciones por el coste de una comparación.

Se hará una única comparación si el primer campo es distinto, con lo que tendremos un número promedio de comparaciones $1\frac{1}{2}$,

se harán dos comparaciones si el primer campo es igual y el segundo distinto, y el número promedio de comparaciones es $2\frac{1}{2}\frac{2}{3}$,

se harán 3 comparaciones si los dos primeros campos son iguales y el tercero distinto: $3\frac{1}{2}\frac{1}{3}\frac{3}{4}$,

y se harán i comparaciones si los $i-1$ primeros son iguales y el i-ésimo distinto, siendo el número promedio de comparaciones $i\frac{1}{2}\frac{1}{3}\dots\frac{1}{i}\frac{i}{i+1}$.

De esta forma, el número total de comparaciones en promedio será:

$$\sum_{i=1}^{n} i\frac{1}{2}\frac{1}{3}\dots\frac{1}{i}\frac{i}{i+1} = \sum_{i=1}^{n} \frac{i^2}{(i+1)!} \leq \sum_{i=1}^{n} \frac{1}{(i-1)!}$$

que está acotado inferiormente por uno (al menos se hace una comparación), y queremos ver que está acotado superiormente por otra constante, con lo que tendremos que las cotas inferior y superior son constantes y el orden exacto es constante.

Para encontrar una cota superior constante consideramos que añadimos términos al sumatorio anterior y que $k! \geq 2^k$ (lo que se puede demostrar fácilmente por inducción que ocurre a partir de $k = 4$), por lo que:

$$\sum_{i=1}^{n} \frac{1}{(i-1)!} \leq \sum_{i=1}^{\infty} \frac{1}{(i-1)!} \leq \sum_{i=1}^{\infty} \frac{1}{2^{i-1}} = 2$$

(ya que el último sumatorio es la suma de una progresión geométrica infinita con razón $\frac{1}{2}$).

Problema 2.20 a) Encontrar O y Ω del algoritmo:

```
max = 0
for i = 1 to n
    cont = 1
    j = i + 1
    while a[i] ≤ a[j]
        j = j + 1
        cont = cont + 1
    endwhile
```

```
    if cont > max
        max = cont
    endif
endfor
```

b) Hacer un programa que haga lo mismo y que tenga $\theta(n)$.

Solución:

a) Para el cálculo de O y Ω podemos suponer que todas las instrucciones tienen el mismo coste. Numeramos las instrucciones para referirnos a ellas:

```
1   max = 0
2   for i = 1 to n
3       cont = 1
4       j = i + 1
5       while a[i] ≤ a[j]
6           j = j + 1
7           cont = cont + 1
8       endwhile
9       if cont > max
10          max = cont
11      endif
12  endfor
```

El número de veces que se ejecuta cada instrucción es:

instrucción 1, 1 vez

2, n veces

3, n veces

4, n veces

5: para cada valor de i entre 1 y n un mínimo de 1 vez y un máximo de $n-i+1$, suponiendo que si es siempre $a[i] \leq a[j]$ para cuando $j = n+1$. Por tanto, se ejecuta un mínimo de n veces y un máximo de $\sum_{i=1}^{n} \sum_{j=i+1}^{n} 1 = \sum_{i=1}^{n}(n-i) = \frac{n^2-n}{2}$

6: igual que la 5, pero un mínimo de 0 veces y un máximo de $\frac{n^2-n}{2}$,

7: un mínimo de 0 y un máximo de $\frac{n^2-n}{2}$,

9, n veces,

10: un mínimo de 1 y un máximo de n.

Por tanto, la cota inferior es $2 + 5n \Rightarrow t(n) \in \Omega(n)$. Y la cota superior es $1 + 3n + \frac{n^2+n}{2} + n^2 - n + 2n = \frac{3n^2+9n+2}{2} \Rightarrow t(n) \in O\left(n^2\right)$

b) El programa calcula, almacenándolo en la variable max, la máxima longitud de una secuencia de datos $s_i, s_{i+1}, ..., s_{i+k}$ dentro del array en la que $s_i \leq s_{i+j}$ con $j = 1, 2, ...k$. Una de estas secuencias acaba cuando encontramos $s_{i+k+1} < s_i$, y por tanto las secuencias empezando en s_{i+j} con $j = 1, 2, ...k$ acabarán en s_{i+k} al ser $s_{i+k+1} < s_i \leq s_{i+j}$, por lo que no habría que probar con secuencias que empiecen en s_{i+j}, sino con las que empiezan en s_{i+k+1}. De este modo, un programa equivalente es:

```
max = 0
```

$cont = 1$
$j = 1$
$k = 2$
for $i = 1$ to n
    if $a[j] \leq a[k]$
        $k = k + 1$
        $cont = cont + 1$
    else
        $j = k$
        $k = j + 1$
        if $cont > max$
            $max = cont$
        endif
    endif
endfor
if $cont > max$
    $max = cont$
    $cont = 1$
endif

Donde $t(n) \in \theta(n)$ pues se ejecuta n veces el cuerpo del for estando acotado inferiormente el número de instrucciones por 4 y superiormente por 6.

Problema 2.21 Calcular o (o-pequeña) del número promedio de asignaciones del algoritmo:

$max = a[1]$
for $i = 2$ to n
    if $a[i] > max$
        $max = a[i]$
    endif
endfor

Solución:

La primera asignación se ejecuta siempre, y la segunda (la interior al if) se ejecutará o no dependiendo de si $a[i] > max$, por lo que el número de asignaciones (sin tener en cuenta las que modifican i que son n) será $A(n) = 1 + \sum_{i=2}^{n} p(a[i] > max)$, donde $p(a[i] > max) = \frac{1}{i}$, pues la probabilidad de que de i datos el mayor sea uno de ellos (en este caso el i-ésimo) es $\frac{1}{i}$ si suponemos que todas las posibles entradas tienen la misma probabilidad. Por tanto $A(n) = 1 + \sum_{i=2}^{n} \frac{1}{i}$.

Para calcular el sumatorio podemos acotarlo de la forma:

$$\int_{2}^{n+1} \frac{1}{x} dx \leq \sum_{i=2}^{n} \frac{1}{i} \leq \int_{1}^{n} \frac{1}{x} dx$$

con lo que $1 + ln\ (n+1) - ln\ 2 \leq A(n) \leq 1 + ln\ n$ y como $\lim_{n\to\infty} \frac{1+ln\ n}{1+ln\ (n+1)-ln\ 2} = \lim_{n\to\infty} \frac{\frac{1}{n}}{\frac{1}{n+1}} = 1$ las dos cotas tienen el mismo o y $A(n) \in o(ln\ n)$.

Problema 2.22 Dado el programa:

$i = 1$
$j = 1$
while $i \leq n$ and $j \leq n$
    if $a[i, j+1] < a[i+1, j]$
        $j = j + 1$
    else
        $i = i + 1$
    endif
endwhile

Calcular su Ω, O y θ.

Solución:

Cada vez que se pasa por el bucle while se realiza la misma cantidad de operaciones: la comprobación del if, y la actualización de una variable (i o j); por lo tanto el tiempo de ejecución vendrá determinado por el número de veces que se entre en el bucle.

Cuando la condición del if es siempre verdad (o siempre falsa) se incrementa siempre la misma variable, con lo que se saldrá del bucle cuando dicha variable tome el valor $n+1$. En este caso se entrará n veces en el bucle while y el número de instrucciones ejecutadas será 2 de las asignaciones iniciales, $n+1$ comprobaciones del bucle, $2n$ instrucciones de los n pasos por el interior del bucle; el número total de instrucciones ejecutadas es $3n+3$ y el algoritmo tiene $\Omega(n)$.

Para obtener el orden, como el número máximo de pasos por el bucle se da cuando se llega a $i = n$ y $j = n$, no entrándose en el bucle en el siguiente paso, se tendrá un número de instrucciones ejecutadas: 2 de las asignaciones iniciales, $2n+1$ de las comprobaciones del bucle, $4n$ de los $2n$ pasos por el cuerpo del bucle; el número total de instrucciones es $6n+3$ y el algoritmo tiene $O(n)$.

Como para este algoritmo es $\Omega = O$, tendremos que el orden exacto es $\theta(n)$.

Problema 2.23 Dado el programa:

$max = -\infty$
for $i = 1$ to n
    for $j = 1$ to m
        if $a[i, j] > max$
            $max = a[i, j]$
        endif
    endfor
endfor

Calcular el o (o pequeña) del número promedio de instrucciones que se ejecutan.

Solución:

Para contar el número de instrucciones numeraremos las líneas del programa:

```
1   max = -∞
2   for i = 1 to n
3       for j = 1 to m
4           if a[i,j] > max
5               max = a[i,j]
```

La instrucción uno se ejecuta una vez, la dos n veces, la tres nm veces, la cuatro nm veces, y sólo queda por determinar el número promedio de veces que se ejecuta la instrucción cinco.

El número promedio de instrucciones queda:
$1+\sum_{i=1}^{n}\left(1+\sum_{j=1}^{m}(2+prob(a[i,j]>max))\right)=1+n+2nm+\sum_{i=1}^{n}\sum_{j=1}^{m}prob(a[i,j]>max)$
y sólo queda por obtener el valor del sumatorio.

Con $i=1$ y $j=1$, $a[1,1]>max$, y $prob(a[1,1]>max)=1$.

Con $i=1$ y $j=2$, $a[1,2]>max$ si $a[1,2]>a[1,1]$, y $prob(a[1,2]>a[1,1])=\frac{1}{2}$ (considerando una distribución uniforme de los datos).

En general, $a[i,j]>max$ si $a[i,j]$ es el máximo de los valores $a[1,1],\ldots,a[1,m],a[2,1],\ldots,a[2,m],\ldots,a[i,1],\ldots,a[i,j]$, y la probabilidad es $\frac{1}{(i-1)m+j}$.

El sumatorio queda:
$\sum_{i=1}^{n}\sum_{j=1}^{m}\frac{1}{(i-1)m+j}=\sum_{k=1}^{nm}\frac{1}{k}\simeq\int_{1}^{nm+1}\frac{1}{x}\,dx=ln\ (nm+1)$,
y el número de operaciones será aproximadamente:
$1+n+2nm+ln\ (nm+1)\in o(2nm)$.

Problema 2.24 a) Dado el programa:

```
i = 1
j = n
while i < j
    if a[i] < a[j]
        i = i * 2
    else
        j = j/2
    endif
endwhile
```

Calcular su orden exacto. Habrá que hacerlo considerando el tamaño potencia de dos y quitando posteriormente esta restricción.

b) Calcular el orden exacto en el caso en que se sustituyan las sentencias $i=i*2$ y $j=j/2$ por $i=i+1$ y $j=j-1$, respectivamente.

Solución:

a) Supondremos que $n=2^k$. Si calculamos el orden contando el número de instrucciones, el tiempo de ejecución vendrá dado por $t(n)=2+t_w(i,j)$, siendo $t_w(i,j)$ el tiempo de ejecución del while con índices i y j, e inicialmente $i=1$ y $j=n$. Si

$a[i] < a[j]$ quedará $t_w(i,j) = 3+t_w(2i,j)$, y si $a[i] \geq a[j]$ quedará $t_w(i,j) = 3+t_w\left(i, \frac{j}{2}\right)$. En cualquier caso se ejecutan tres instrucciones por cada pasada por el while, con lo que para obtener el orden habrá que contar el número de veces que se pasa por el bucle. Como inicialmente los índices son 1 y n, después de la primera pasada serán 2 y n o 1 y $\frac{n}{2}$, después de la segunda serán 4 y n, o 2 y $\frac{n}{2}$, o 1 y $\frac{n}{4}$. En general, después de la pasada p-ésima serán 2^q y $\frac{n}{2^{p-q}}$, siendo q un valor entre 0 y p. Como no se entra en el bucle cuando $i = j$, esto ocurrirá en la pasada p-ésima en la que $2^q = \frac{n}{2^{p-q}}$, o lo que es lo mismo cuando $n = 2^p$. Por lo tanto, se realizan k pasadas por el bucle while sean cuales sean los datos en el array, por lo que $\Omega(t) = O(t) = \theta(t)$, y $t(n) \in \theta\left(\log n | n = 2^k\right)$.

Para quitar la restricción de que n sea potencia de dos hay que comprobar que $f(n) = \log n$ es eventualmente no decreciente y 2-armónica, y que $t(n)$ es eventualmente no decreciente.

$\log n$ es creciente, y $\log 2n = 1 + \log n \in \theta(\log n)$, por lo que es 2-armónica.

Para demostrar que $t(n)$ es creciente compararemos los tiempos para n y $n+1$. Si $2^k < n < n+1 \leq 2^{k+1}$, $t(n) = t(n+1)$ pues en los dos casos se entra $k+1$ veces en el bucle while; y si $n = 2^k < n+1$, $t(n) < t(n+1)$ pues para n se entra k veces en el bucle y para $n+1$ se entra $k+1$ veces.

b) Se razona igual que en el caso anterior y se obtiene $t(n) = 2 + t_w(i,j)$, con $t_w(i,j) = 3 + t_w(i+1,j)$ o $t_w(i,j) = 3 + t_w(i,j-1)$. Los índices tras la pasada p-ésima son en este caso q y $n-(p-q)$, con q entre 0 y p, y no se entra en el while cuando $q = n-(p-q)$, o lo que es lo mismo, cuando $n = p$, por lo que se pasa $n-1$ veces por el bucle while independientemente de la distribución de los datos en el array, por lo que $\Omega(t) = O(t) = \theta(t) = \theta(n)$.

En este caso no hemos impuesto la condición de que n sea potencia de dos.

Problema 2.25 Dado el programa:

```
encontrado=false
for i = 1 to n − 1
    if x[i] = y[1]
        if x[i + 1] = y[2]
            encontrado=true
            lugar = i
        endif
    endif
    i = i + 1
endfor
```

Calcular su tiempo promedio de ejecución suponiendo una probabilidad p de que dos elementos cualesquiera (de x o de y) sean iguales.

Solución:

La instrucción $i = i+1$ no tiene sentido en este algoritmo. Si consideramos que no está tendremos $n-1$ pasadas por el cuerpo del for, y si consideramos que está y que los valores de i con los que se entra en el bucle son 1,3,5,...,$n-1$ o 1,3,5,...,$n-2$, no habrá variación en el tiempo de ejecución salvo en factores constantes. Por tanto, supondremos que dicha instrucción no aparece en el programa, con lo que tendremos $t(n) = a + \sum_{i=1}^{n-1}(b + t_{if}(i))$, donde b corresponde al tiempo de comprobación y actualización del índice en el for, y $t_{if}(i)$ es el tiempo de paso por el if externo. Como la instrucción if $x[i] = y[1]$ se ejecuta siempre y es verdad en probabilidad p, la if $x[i+1] = y[2]$ se ejecutará con probabilidad p y será verdad con probabilidad p, y las dos asignaciones del if más interno se ejecutarán con probabilidad p^2. Por tanto, tendremos

$$t(n) = a + \sum_{i=1}^{n-1}\left(b + c + dp + ep^2\right) = n\left(b + c + dp + ep^2\right) + a - b - c - dp - ep^2 \in \theta(n)$$

independientemente del valor de p, que al ser una probabilidad está entre 0 y 1.

Problema 2.26 Dado el programa:

```
cont = 0
for i = 1 to n
    j = 1
    while j < i
        if a[i] < a[j]
            cont = cont + 1
            j = j * 2
        else
            j = j + 1
        endif
    endwhile
endfor
```

Calcular Ω y O de su tiempo de ejecución (tienen que ser unos buenos valores, no unos cualesquiera).

Solución:

Supondremos n potencia de dos ($n = 2^k$), y después quitaremos esta restricción utilizando el teorema visto en clase.

Tenemos $t(n) = a + \sum_{i=1}^{n}(b + t_w)$, donde a es la constante correspondiente a ejecutar la instrucción $cont = 0$, b representa el tiempo empleado en asignar un valor al índice i, asignar a j el valor 1, en la comprobación de la condición del for y la primera comprobación del while; y t_w es el tiempo empleado en el while.

El caso más favorable será cuando $a[i] < a[j]$ cada vez que se pase por la condición del while. En este caso j varía multiplicándose por dos en cada paso, con lo que $t(n) \geq a + \sum_{i=1}^{n}(b + c\log i)$, siendo c la constante correspondiente a la comprobación del while y las asignaciones $cont = cont + 1$ y $j = j * 2$. Tenemos $t(n) \geq a + bn + c\int_1^{n+1}\log i \; di = a + bn + c\int_1^{n+1}\log e \;\; \ln i \; di$. Con lo que, para obtener Ω basta con calcular la integral $\int_1^{n+1} \ln i \; di$ que, aplicando partes con $u = \ln i$ y $dv = di$, queda $(n+1)\ln(n+1) - n$, y $t(n) \in \Omega\left(n \ln n | n = 2^k\right)$.

Para quitar la restricción basta con probar que $f(n) = n \ln n$ es eventualmente no decreciente (en realidad es creciente para valores positivos), y es 2-armónica pues $f(2n) = 2n \ln 2n = 2n \ln n + 2n \ln 2 \in O(n \ln n)$. Además, la función tiempo de ejecución es creciente trivialmente, pues con tamaño $n+1$ se ejecutan las mismas pasadas por el bucle for que con tamaño n, mas la última pasada correspondiente al valor $i = n+1$.

El caso más desfavorable será cuando cada vez que se pase por la condición del while sea $a[i] \geq a[j]$, con lo que el cuerpo del while se ejecuta $i-1$ veces. En este caso $t(n) = a + \sum_{i=1}^{n}(b + c(i-1)) = a + bn + c\frac{n^2-n}{2}$, y $t(n) \in O\left(n^2\right)$. En este caso no se ha impuesto que n sea potencia de dos.

Problema 2.27 Dado el programa:

$cont = 0$
for $i = 1$ to n
    for $j = 1$ to $i-1$
        if $a[i] < a[j]$
            $cont = cont + 1$
        endif
    endfor
endfor

Calcular el orden exacto del número promedio de veces que se ejecuta la instrucción $cont = cont + 1$.

Solución:

Como las valores del array a no se modifican a lo largo de la ejecución del programa, podemos considerar que estos valores están uniformemente distribuidos y que en cada comparación la probabilidad de que sea verdad es $\frac{1}{2}$.

Tendremos $t(n) = \sum_{i=1}^{n} \sum_{j=1}^{i-1} \frac{1}{2} = \sum_{i=2}^{n} \frac{i-1}{2} = \frac{n^2-n}{4} \in \theta\left(n^2\right)$.

Problema 2.28 Dado el siguiente algoritmo para obtener el segundo mayor elemento en un array:

$max1 = -\infty$
$max2 = -\infty$
for $i = 1$ to n
    if $a[i] > max1$
        $max2 = max1$
        $max1 = a[i]$
    else if $a[i] > max2$
        $max2 = a[i]$
    endif

endfor

a) Obtener θ del algoritmo obteniendo O y Ω.

b) Calcular o (o minúscula) del número promedio de instrucciones que se ejecutan, suponiendo que cada línea en el programa es una instrucción.

c) Dar un programa distinto para resolver el mismo problema, y que tenga una o (o minúscula) del número promedio de instrucciones que se ejecutan menor que la del programa anterior.

Solución:

a) Las inicializaciones de $max1$ y $max2$ se ejecutan una vez, y después se ejecuta el bucle for con valores de $i = 1$ a n, por lo que $t(n) = a + \sum_{i=1}^{n} t_{for}(i)$, donde a es una constante y dentro del for se ejecutan (para cualquier valor de i) un mínimo de 2 instrucciones (las dos comparaciones) y un máximo de 3 (una comparación y dos asignaciones o dos comparaciones y una asignación). Tendremos que $t_{for}(i)$ está acotado inferior y superiormente por constantes k_1 y k_2, y $a + nk_1 \leq t(n) \leq a + nk_2$, con lo que $t(n) \in \Omega(n)$, y $t(n) \in O(n)$, lo que implica que $t(n) \in \theta(n)$.

b) Numeramos las líneas del programa:

```
1   max1 = -∞
2   max2 = -∞
3   for i = 1 to n
4       if a[i] > max1
5           max2 = max1
6           max1 = a[i]
7       else if a[i] > max2
8           max2 = a[i]
9       endif
10  endfor
```

y no consideraremos las líneas 9 y 10.

Las líneas 1 y 2 se ejecutan cada una una vez, la 4 se ejecuta n veces, y la 3 consideraremos que se ejecuta $n+1$ veces: las n veces que se entra dentro del for y la última para comprobar que no hay que entrar. El número de veces que se ejecuta cada una de las demás líneas depende de la distribución de los datos en el array, por lo que, para obtener el número promedio de instrucciones ejecutadas, supondremos que los datos están distribuidos inicialmente de manera uniforme, con lo que un dato tiene una probabilidad $\frac{1}{i}$ de ser el mayor de entre i datos, y la misma probabilidad de ser el segundo mayor. Con $i = 1$ las instrucciones 5 y 6 se ejecutan seguro, y con $i > 1$ se ejecutarán con probabilidad $\frac{1}{i}$, con lo que el número promedio de veces que se ejecutan estas instrucciones es $2\left(1 + \sum_{i=2}^{n} \frac{1}{i}\right)$. La instrucción 7 se ejecutará cada vez con probabilidad $1 - \frac{1}{i}$, y tendremos que el número promedio de veces que se ejecuta es $\sum_{i=2}^{n} \left(1 - \frac{1}{i}\right)$. La instrucción 8 no se ejecuta con $i = 1$, y en los demás casos se ejecuta con probabilidad $\frac{1}{i}$ (la de que $a[i]$ sea el segundo mayor elemento de los i primeros), con lo que el número promedio de veces que se ejecuta es $\sum_{i=2}^{n} \frac{1}{i}$. De este modo, el número promedio de veces que se ejecutan todas las instrucciones es:

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

Y como:

$$\int_{2}^{n+1} \frac{1}{x}dx \leq \sum_{i=2}^{n} \frac{1}{i} \leq \int_{1}^{n} \frac{1}{x}dx \Rightarrow ln\ (n+1) - ln\ 2 \leq \sum_{i=2}^{n} \frac{1}{i} \leq ln\ n$$

tenemos que:

$$4 + 3n + 2ln\ (n+1) + 2ln\ n \leq t(n) \leq 4 + 3n + 2ln\ n$$

Como en las dos cotas es término de mayor orden es el término en n, tendremos que $t(n) \in o(3n)$.

c) Si hicieramos la comprobación de si $a[i] > max2$ antes de comprobar si $a[i] > max1$ nos podríamos ahorrar algo de trabajo ya que al aumantar i es cada vez más improbable que $a[i] > max2$.

El programa podría ser:

```
max1 = -∞
max2 = -∞
for i = 1 to n
    if a[i] > max2
        if a[i] > max1
            max2 = max1
            max1 = a[i]
        else
            max2 = a[i]
        endif
    endif
endfor
```

Y se puede ver (como en el apartado b) que en este caso $t(n) \in o(2n)$.

Problema 2.29 Un número capicúa está formado por una cadena de n dígitos, c:array$[1..n]$ of 0..9, donde cada $c[i] = c[n-i+1]$. Si consideramos que los números de n dígitos pueden tener los 10 dígitos de 0 a 9 con la misma probabilidad en cada una de sus posiciones (incluimos números que contengan el 0 al principio), calcular el θ del tiempo promedio de ejecución de un programa que determine si un número de n dígitos es capicúa.

Solución:

Un programa para resolver este problema puede ser:

$i = 1$
mientras $c[i] = c[n-i+1]$ y $i \leq \lfloor \frac{n}{2} \rfloor$
    $i = i + 1$
finmientras
return $\left(i > \lfloor \frac{n}{2} \rfloor\right)$

La probabilidad de que se cumpla $c[i] = c[n-i+1]$ es $\frac{1}{10}$ cada vez pues dado un dígito entre 0 y 9 hay una posibilidad entre diez de que otro dígito entre 0 y 9 tenga el mismo valor que el primero.

Para obtener θ del tiempo promedio sólo hay que contar el número promedio de veces que se pasa por el interior del bucle, pues las instrucciones fuera del bucle y la última comprobación gastan un tiempo constante.

La primera pasada por el bucle (hacer $i = 2$) se hace con probabilidad $\frac{1}{10}$, que es $p(c[1] = c[n])$. La segunda pasada ($i = 3$) se hace con probabilidad $p(c[1] = c[n] \text{ } y \text{ } c[2] = c[n-1]) = \frac{1}{10^2}$. En general, la pasada k-ésima se hace con probabilidad $\frac{1}{10^k}$; con lo que el número promedio de pasadas por el bucle es $\sum_{k=1}^{\lfloor \frac{n}{2} \rfloor} \frac{1}{10^k}$, lo que es una progresión geométrica cuya suma es $\frac{10}{9}\left(\frac{1}{10} - \frac{1}{10^{k+1}}\right) \leq \frac{1}{9}$.

Al estar el número promedio de pasadas acotado superiormente por una constante y tener las operaciones externas un tiempo constante, el orden es $\theta(1)$.

Problema 2.30 Calcular el orden exacto del tiempo de ejecución, y la o minúscula del número de operaciones en coma flotante, del cálculo de la potencia n-ésima, siendo $n = 2^k$, de una matriz cuadrada $n \times n$ cuando se utiliza la multiplicación directa de matrices y la multiplicación de Strassen. Se debe indicar cómo se calcula la potencia y el orden de ejecución debe ser menor de n^4. Analizar también la ocupación de memoria en los dos casos.

Solución:

Para calcular A^{2^k} se puede hacer $A^2 = AA$, después $A^4 = A^2A^2$, después $A^8 = A^4A^4$, y así sucesivamente hasta $A^{2^k} = A^{2^{k-1}}A^{2^{k-1}}$, con lo que se hacen k multiplicaciones matriciales donde las dos matrices a multiplicar son la misma. Si las multiplicaciones matriciales se hacen con la multiplicación directa el coste será $\theta\left(kn^3\right) = \theta\left(n^3 \log_2 n\right)$, y si es con el método de Strassen $\theta\left(n^{\log_2 7} \log_2 n\right)$.

En cuanto a la o pequeña del número de operaciones en coma flotante, en el caso directo es $o\left(2n^3 \log_n\right)$. Haciendo la multiplicación por el método de Strassen tendríamos la ecuación de recurrencia:

$$t(n) = 7t\left(\frac{n}{2}\right) + \frac{b}{4}n^2$$

donde b representa el número de operaciones de orden n^2 (sumas y restas de matrices de tamaño $\frac{n}{2}$) que se realizan en cada llamada recursiva, y que para multiplicar dos matrices distintas es 18. Resolviendo la recursión tenemos $t(n) = c_1 7^{\log_2 n} + c_2 n^2$, e imponiendo los casos base 1 y 2:

$$t(1) = 1 = c_1 + c_2$$
$$t(2) = 7 + b = c_1 7 + c_2 4$$

con lo que $c_1 = 1 + \frac{b}{3}$, que en el caso normal da valor 7, pero en nuestro caso podemos ver si se puede obtener un valor menor. Si ponemos las fórmulas de la multiplicación de Strassen teniendo en cuenta que las dos matrices a multiplicar son la misma tenemos:

$$\begin{aligned}
P &= (A_{11} + A_{22})^2 \\
Q &= (A_{21} + A_{22})A_{11} \\
R &= A_{11}(A_{12} - A_{22}) \\
S &= A_{22}(A_{21} - A_{11}) \\
T &= (A_{11} + A_{12})A_{22} \\
U &= (A_{21} - A_{11})(A_{11} + A_{12}) \\
V &= (A_{12} - A_{22})(A_{21} + A_{22}) \\
C_{11} &= P + S - T + V \\
C_{12} &= R + T \\
C_{21} &= Q + S \\
C_{22} &= P + R - Q + U
\end{aligned}$$

donde vemos que hay matrices repetidas: $A_{21} + A_{22}$ en el cálculo de Q y V, $A_{12} - A_{22}$ en el de R y V, $A_{21} - A_{11}$ en el de S y U, y $A_{11} + A_{12}$ en el de T y U. Por lo tanto, el número de operaciones de suma y resta de matrices se reduce a 13, y $c_1 = \frac{16}{3}$, con lo que el orden es $o\left(\frac{16}{3} n^{\log_2 7} \log_2 n\right)$.

Para la ocupación de memoria, en el método directo tenemos en cada multiplicación que multiplicar una matriz por sí misma, con lo que se necesita espacio para dos matrices: $2n^2$. Al multiplicar A^2 por sí misma el resultado se puede dejar en la zona de memoria de A si no nos importa perder esta matriz. De este modo, la zona de memoria de la matriz origen en un paso pasa a ser la de la matriz destino en el siguiente paso, y la de destino pasa a ser la de origen, con lo que no son necesarias más de $2n^2$ posiciones de memoria.

En la multiplicación de Strassen se necesitan $2n^2$ posiciones para las matrices origen y destino como en el caso anterior, y además los espacios de memoria que se generan en las llamadas recursivas, en las que cuando se llama con dimensión de las matrices $n \times n$ se necesitan tres matrices $\frac{n}{2} \times \frac{n}{2}$, con lo que tenemos la recurrencia $m(n) = m\left(\frac{n}{2}\right) + 3\left(\frac{n}{2}\right)^2$, que expandiéndola se llega a $m(n) = m(1) + 3n^2\left(\frac{1}{4} + \frac{1}{4^2} + \ldots + \frac{1}{4^k}\right) = m(1) + n^2\left(1 - \frac{1}{4^{\log_2 n}}\right)$, y el orden de la ocupación de memoria para las matrices auxiliares es $o\left(n^2\right)$, y para el total del programa $o\left(3n^2\right)$.

Problema 2.31 Dado el siguiente esquema:

$der = 0$

```
izq = 0
for i = 1 to n
    if par(a[i,i])
        for j = i+1 to n
            if a[i,j] > 0
                der++
            endif
        endfor
    else
        for j = 1 to i-1
            if a[i,j] > 0
                izq++
            endif
        endfor
    endif
endfor
```

calcular:
a) θ del tiempo de ejecución.
b) o (o pequeña) del número promedio de instrucciones que se ejecutan.
Solución:
a) El tiempo de ejecución se puede poner en la forma $t(n) = a + \sum_{i=1}^{n} t_{for}(i)$, donde $t_{for}(i)$ representa el tiempo de ejecución en el paso por el for con índice i. Dentro del for se ejecuta un segundo for con índice j, que puede variar de $i+1$ a n o de 1 a $i-1$.

Lo más favorable es ejecutar el for de 1 a $i-1$ con i entre 1 y $\frac{n}{2}$, y de $i+1$ a n con i entre $\frac{n}{2}+1$ y n. De este modo el caso más favorable será cuando con i entre 1 y $\frac{n}{2}$ es $a[i,i]$ impar, y entre $\frac{n}{2}+1$ y n es par, y además los $a[i,j]$ son negativos o cero (sólo es necesario que lo sean los que se analizan: la izquierda de la diagonal en la primera mitad del array y la parte de la derecha de la diagonal en la segunda mitad). Se tiene $t_m(n) = a + \sum_{i=1}^{\frac{n}{2}} \left(b + \sum_{j=1}^{i-1} c\right) + \sum_{i=\frac{n}{2}+1}^{n} \left(b + \sum_{j=i+1}^{n} c\right) = a + bn + \sum_{i=1}^{\frac{n}{2}} c(i-1) + \sum_{i=\frac{n}{2}+1}^{n} c(n-i) = a + bn + c\frac{n^2}{4} - c\frac{n}{2} \in \theta\left(n^2\right)$, con lo que $t(n) \in \Omega\left(n^2\right)$.

El caso más desfavorable ocurre cuando con i entre 1 y $\frac{n}{2}$ es $a[i,i]$ par, y entre $\frac{n}{2}+1$ y n es impar, y además los $a[i,j]$ son positivos (sólo es necesario que lo sean los que se analizan: la derecha de la diagonal en la primera mitad del array y la parte de la izquierda de la diagonal en la segunda mitad). Se tiene $t_M(n) = a + \sum_{i=1}^{\frac{n}{2}} \left(b + \sum_{j=i+1}^{n} c\right) + \sum_{i=\frac{n}{2}+1}^{n} \left(b + \sum_{j=1}^{i-1} c\right) = a + bn + c\frac{3n^2}{4} - c\frac{n}{2} \in \theta\left(n^2\right)$, con lo que $t(n) \in O\left(n^2\right)$. El valor de c en este caso será mayor que en el caso anterior pues incluye el incremento de una variable.

Al coincidir Ω y O tenemos que $t(n) \in \theta\left(n^2\right)$.
b) Para calcular la o del tiempo promedio vamos a contar el número de veces que se ejecuta cada instrucción afectada por la probabilidad de que se ejecute. Para esto

supondremos que la probabilidad de que un número sea par o impar es la misma, y la de que sea positivo o no sea positivo también es la misma. Tendremos las 2 instrucciones externas al bucle en i y los $n+1$ pasos por ese bucle, siendo el último para comprobar que no se entra al interior del bucle. Cada vez que se entra al bucle se ejecuta la comprobación de si el elemento de la diagonal es par y con probabilidad de $\frac{1}{2}$ se ejecuta el primero de los bucles en j y con la misma probabilidad el segundo; en cada uno de estos bucles se ejecuta el paso por el bucle y una comprobación de si el elemento correspondiente es positivo, y con probabilidad $\frac{1}{2}$ se actualiza una variable.

El número de instrucciones será $2+n+1+\sum_{i=1}^{n}\left(3+\frac{1}{2}\sum_{j=i+1}^{n}\left(2+\frac{1}{2}\right)+\frac{1}{2}\sum_{j=1}^{i-1}\left(2+\frac{1}{2}\right)\right)=\frac{5}{4}n^2+\frac{11}{4}n+3\in o\left(\frac{5}{4}n^2\right)$.

Problema 2.32 Dado el esquema:

$m=a[1]$
$i=2$
while $i\leq n$ and $a[i]>m$
  $m=\frac{m*(i-1)+a[i]}{i}$
  $i++$
endwhile

Calcular:
a) Ω y O del tiempo de ejecución.
b) θ y o del número promedio de instrucciones que se ejecutan.

Solución:

a) El caso más favorable lo tendremos cuando se entre una única vez en el bucle while, lo que ocurre cuando $a[2]\leq a[1]$. En este caso se ejecutan las dos asignaciones iniciales, una comparación del bucle en la que se entre, las dos actualizaciones del while, y la última comparación del while para no entrar. El número de instrucciones es 6, $t_m(n)\in\theta(1)$, y $t(n)\in\Omega(1)$.

El caso más desfavorable se da cuando $a[i]>m$ cada vez que se comprueba la condición del while, por lo que se sale de este al ser $i>n$. Por lo tanto, se entra en el while con $i=2$ hasta $i=n$, $t_M(n)\in\theta(n)$, y $t(n)\in O(n)$.

b) El número promedio de instrucciones que se ejecuta viene dado por la fórmula $t_p(n)=3+3\sum_{i=2}^{n}p(a[i]>m)$; donde el primer 3 corresponde a las dos asignaciones iniciales y a la última comprobación del bucle while (para comprobar que no se entra); el segundo 3 corresponde a las dos actualizaciones internas al while y a la comprobación de éste; y por $p(a[i]>m)$ representamos la probabilidad de que el elemento i-ésimo sea mayor que la media de los elementos anteriores, pero habiendo llegado a esa comparación, que se llega cuando $a[j]$ es mayor que la media de los elementos anteriores, con $j=2,3,\ldots,i$. Si suponemos una distribución aleatoria de los datos, la probabilidad de que un elemento sea mayor que la media de otros elementos podemos considerarla $\frac{1}{2}$, y $p(a[i]>m)=\left(\frac{1}{2}\right)^{i-1}$. Por tanto la fórmula será:

$$t_p = 3 + 3\sum_{i=2}^{n}\left(\frac{1}{2}\right)^{i-1} = 3 + 3\frac{\frac{1}{2}-\left(\frac{1}{2}\right)^n}{1-\frac{1}{2}} = 6 - \frac{3}{2^{n-1}};\text{ y } t_p(n) \in \theta(1),\text{ y } t_p(n) \in o(6)$$

Problema 2.33 a) Calcular el orden exacto y la función o (o pequeña) de la función:

```
funcion1(n:integer):integer
if n = 1
    return 1
else
    return(2*funcion1(n − 1)+1)
endif
```

b) ¿Cuáles serían el orden exacto y la función o de funcion1 si sustituimos en el caso base $n = 1$ por $n = k$ con k constante?

Solución:

a) En el caso base ($n = 1$) se ejecuta la comprobación del if y el return, por lo que el tiempo es constante, que llamaremos a. Cuando no estamos en el caso base, se hace una llamada con $n - 1$ y el resto de operaciones llevan un tiempo constante que llamaremos b. La ecuación de recurrencia queda:

$$t(n) = \begin{cases} a & si\ n = 1 \\ b + t(n-1) & si\ n > 1 \end{cases}$$

Expandiendo la recurrencia tenemos:

$$t(n) = b + t(n-1) = 2b + t(n-2) = \ldots = ib + t(n-i)$$

En el caso en que el caso base sea uno, la expansión de la recurrencia acaba cuando $n - i = 1$ y por tanto $i = n - 1$, y queda $t(n) = (n-1)b + a = bn + a - b$, con lo que $t(n) \in \theta(n)$, y $t(n) \in o(bn)$.

b) Si se cambia el caso base a $n = k$, la expansión de la recurrencia acaba cuando $n - i = k$, con lo que $i = n - k$, y $t(n) = (n-k)b + a = bn + a - bk$, como k es una constante el término de mayor orden es el término en n, y volvemos a tener $t(n) \in \theta(n)$, y $t(n) \in o(bn)$.

Problema 2.34 Calcular el orden exacto del tiempo promedio de ejecución de la función:

```
funcion2(a: array[1,...,n] of integer;i,d:integer):integer
if i ≥ d
    return
else
    if par(a[i]) and par(a[d])
        i = i + (d−i)/2
        return(1+funcion2(a,i,d))
    else
```

$d = d - \frac{d-i}{2}$
return(funcion2(a,i,d))
endif
endif

cuando se llama con funcion2(a,1,n). Hacerlo suponiendo que n es potencia de dos y después quitar la restricción.

Solución:

La ejecución del caso base lleva un tiempo constante que llamamos a. Cuando no estamos en el caso base, tanto si se entra por la primera rama del if o por la segunda, se ejecutan las mismas operaciones de comprobación del if, actualización de un índice, y el return, todo esto llevará un tiempo constante (b), y la única diferencia es que si la condición del if es verdadera se suma uno al resultado de la siguiente llamada a funcion2, y esta suma tiene un tiempo constante que llamamos c. Para plantear la ecuación de recurrencia en el caso del tiempo promedio hay que ver con qué probabilidad se entra en cada una de las ramas del if. Se entra en la primera si $a[i]$ y $a[j]$ son pares, y si suponemos una distribución aleatoria de números en el array, esto ocurrirá con probabilidad $\frac{1}{4}$, y lo contrario con probabilidad $\frac{3}{4}$. Como el contenido de los diferentes elementos del array no cambia a lo largo de la ejecución, las probabilidades son siempre esas, y la ecuación de recurrencia (suponiendo n potencia de dos) queda:

$$t(n) = \begin{cases} a & si\ n = 1 \\ \frac{1}{4}\left(b + c + t\left(\frac{n}{2}\right)\right) + \frac{3}{4}\left(b + t\left(\frac{n}{2}\right)\right) = b + t\left(\frac{n}{2}\right) + \frac{1}{4}c & si\ n > 1 \end{cases}$$

Para obtener el orden exacto podemos considerar que $b + \frac{1}{4}c$ es una constante que llamamos k, y expandiendo la recurrencia tenemos:

$$t(n) = k + t\left(\frac{n}{2}\right) = \ldots = ki + t\left(\frac{n}{2^i}\right)$$

Como el caso base es cuando queda un único elemento, el valor de i será $i = \log n$, y tendremos $t(n) = k \log n + a$, con lo que $t(n) \in (\log n | n\ potencia\ de\ 2)$.

Para quitar la restricción de que n sea potencia de dos, hay que demostrar que $\log n$ es eventualmente no decreciente y 2-armónica, y que $t(n)$ es eventualmente no decreciente. $\log n$ es creciente en su campo de existencia, y $\log 2n = 1 + \log n \in \theta(\log n)$, con lo que es 2-armónica.

Para demostrar que $t(n)$ es eventualmente no decreciente demostraremos que es creciente, y lo haremos por inducción. En el caso base de la inducción $t(1) = a$, y $t(2) = k + t(1) = k + a$, por lo que $t(2) > t(1)$.

Si suponemos que $t(i) \geq t(i-1)$ con $i = 2, 3, \ldots, n$, será $t(n+1) \geq t(n)$, ya que si $2^r - 1 \leq n < n+1 < 2^{r+1} - 1$ es $t(n+1) = t(n) = t\left(\frac{n+1}{2}\right) + k$, y si $n < 2^r - 1 = n+1$ es $t(n+1) = t\left(\frac{n+1}{2}\right) + k > t\left(\frac{n+1}{2} - 1\right) + k = t(n)$.

Problema 2.35 Dado el esquema:

for $j = 1$ to n
  for $i = 1$ to j
    if $a[i, j] < a[1, j]$
      $a[1, j] = a[i, j]$
    endif
  endfor
endfor
for $i = 1$ to n
  if $a[1, i] < a[1, 1]$
    $a[1, 1] = a[1, i]$
  endif
endfor

Estudiar su O y Ω, cotas para el o, y o del número promedio de asignaciones de elementos del array.

Solución:

El tiempo de ejecución tiene la fórmula:

$$a + \sum_{j=1}^{n} \left(b + \sum_{i=1}^{j} c \right) + \sum_{i=1}^{n} d$$

donde a es una costante que corresponde al último paso por cada uno de los `for`, b una constante que corresponde al último paso por el `for` más interno de los dos primeros, c es un valor que corresponde a la ejecución del interior de los dos primeros `for` y al paso por el interior de ellos, y d es un valor que corresponde al paso por el último `for`.

Una cota superior, y por lo tanto O, se tendría con c obtenida ejecutándose siempre las asignaciones; y una cota inferior, y por lo tanto Ω, no ejecutándose ninguna asignación. Desarrollando la fórmula tenemos:

$$a + bn + \sum_{j=1}^{n} cj + dn = a + bn + \frac{n+1}{2} n + dn = c\frac{n^2}{2} + \left(b + d + \frac{c}{2} \right) n + a$$

Dado que el valor de las constantes no influye en los órdenes, tenemos que $t(n) \in O(n^2)$, y $t(n) \in \Omega(n^2)$.

Para obtener o hay que determinar el valor de la constante que afecta a n^2. Si contamos el número de instrucciones, el valor de c sería 2 en el caso más favorable y 3 en el más desfavorable, pues siempre se ejecuta el paso por el `for` interno y la comprobación interna a ese `for`, y la asignación puede ejecutarse o no. De este modo, aunque no sabemos el valor de o, si sabemos unas cotas: $n^2 \leq o(t) \leq \frac{3}{2}n^2$.

Para calcular el número medio de asignaciones que se ejecutan consideraremos que la distribución inicial de los datos es aleatoria, pudiendo un dato ser menor que otro con probabilidad $\frac{1}{2}$.

En los primeros dos bucles se recorren las columnas de un array $n \times n$ variando j, y para cada columna se van comparando los elementos de esa columna hasta llegar al elemento diagonal quedando en la primera fila el menor de los elementos.

Dado un j fijo:

la primera asignación no se ejecutará nunca pues se compara un elemento consigo mismo;

la segunda asignación se ejecutará con probabilidad $\frac{1}{2}$;

la tercera asignación se ejecutará si $a[3,j] < a[1,j]$, pero el valor actual de $a[1,j] = min\{a[1,j], a[2,j]\}$ (mínimo de los valores iniciales), con lo que la asignación se ejecuta si $a[3,j]$ es el menor de los tres primeros elementos de la columna, lo que ocurre con probabilidad $\frac{1}{3}$;

en general se ejecuta la asignación k-ésima si $a[k,j] < min\{a[1,j], \ldots, a[k-1,j]\}$, lo que ocurre si $a[k,j]$ es el menor de los k primeros elementos de la columna, lo que ocurre con probabilidad $\frac{1}{k}$.

El número medio de veces que se ejecuta la primera asignación es:

$$\sum_{j=2}^{n}\sum_{i=2}^{j}\frac{1}{i} \simeq \sum_{j=2}^{n}\int_{1}^{j}\frac{1}{x}dx = \sum_{j=2}^{n}\ln j \simeq \int_{1}^{n}\ln x \; dx = n\ln n - n + 1$$

El segundo bucle tiene un orden lineal, por lo que el número medio de asignaciones en el segundo bucle no puede superar el término $n\ln n$ que nos aparece en el primer bucle, por lo que el o que estamos buscando es $o(n\ln n)$.

Problema 2.36 Dado el programa:

for $i = 1$ to n
    $j = i + 1$
    while $a[j] < a([i]$ y $j \leq n$
        $a[i] = a[j]$
        $j++$
    endwhile
endfor

a) Encontrar un error en el programa y corregirlo.
b) Obtener O del programa.
c) Obtener Ω del programa.
d) Obtener θ del número promedio de instrucciones ejecutadas.

Solución:

a) En el bucle while se evalúa la condición $j \leq n$ después de acceder al array a en la primera parte de la evaluación del while ($a[i] < a[j]$) por lo que puede que j tome el valor $n+1$ y pretendamos acceder a la posición $n+1$ del array, que puede no existir y por tanto dar error de ejecución. La solución más simple es cambiar el orden de las comprobaciones en el while, con lo que quedaría:

while $j \leq n$ y $a[i] < a[j]$

b) Como nos piden órdenes consideraremos todas las instrucciones de coste 1 salvo las dos que indican el fin del while y del for, que no las consideraremos.

El tiempo será $\sum_{i=1}^{n}(2+t_w(i))$, correspondiendo el sumatorio a los n pasos por el for, el valor 2 a las dos instrucciones que se ejecutan en cada paso (la actualización del índice y del valor de j), y $t_w(i)$ al tiempo del while en el paso i-ésimo por el for.

En el caso más desfavorable se realizan pasos por el while hasta que $j=n+1$, que no se entra en el cuerpo del while, con lo que $t_w(i)=\sum_{j=i+1}^{n}3=3(n-i)$, y el tiempo queda $\sum_{i=1}^{n}(2+3(n-i))=2n+3n^2-3\sum_{i=1}^{n}i=2n+3n^2-3\frac{1+n}{2}n=2n+3n^2-\frac{3}{2}n^2-\frac{3}{2}n=\frac{3}{2}n^2+\frac{n}{2}\in O(n^2)$.

c) En el caso más favorable no se entraría nunca en el cuerpo del while porque sea siempre $a[i+1]\geq a[i]$, con lo que el tiempo será $\sum_{i=1}^{n}3=3n\in\Omega(n)$.

d) Para estudiar el tiempo en el caso promedio tendremos que obtener el valor de $t_w(i)$ suponiendo una distribución uniforme de los datos.

Para $i=1$, cuando j vale 2, será $a[j]<a[i]$ con probabilidad $\frac{1}{2}$. j pasará a valer 3 con probabilidad $\frac{1}{2}$, y será $a[3]>a[1]$ con probabilidad $\frac{1}{2}\frac{1}{3}=\frac{1}{3!}$, que corresponde a ser $a[3]<a[1]$ pero habiendo sido antes $a[2]<a[1]$, en cuyo caso se ha copiado $a[2]$ en $a[1]$; por lo que estamos en el caso en que los tres primeros números estaban ordenados de la forma $a[1]>a[2]>a[3]$, lo que ocurre en una sola de las 3! posibles combinaciones. Por tanto, para $i=1$, para un valor j se llega a ese valor y se entra dentro del while si $a[1]>a[2]>\ldots>a[j]$, lo que ocurre con probabilidad $\frac{1}{j!}$. Tendremos $t_w(1)=1+\sum_{j=2}^{n}\frac{1}{j!}$.

Para un valor cualquiera de i, para $j=i+1$ se entrará en el cuerpo del while si $a[i]>a[i+1]$, lo que ocurre con probabilidad $\frac{1}{2}$, se pasará a $j=i+2$ con probabilidad $\frac{1}{2}$ y se entrará en el cuerpo del while con probabilidad $\frac{1}{3!}$, que corresponde a $a[i]>a[i+1]>a[i+2]$. Para un valor cualquiera de j se entrará con probabilidad $\frac{1}{(j-i+1)!}$. Tendremos $t_w(i)=1+3\sum_{k=2}^{n-i+1}\frac{1}{k!}$.

El tiempo promedio será $\sum_{i=1}^{n}\left(3+3\sum_{k=2}^{n-i+1}\frac{1}{k!}\right)$. Este sumatorio no podemos obtenerlo con la teoría estudiada, por lo que lo dejamos así.

Capítulo 3

DIVIDE Y VENCERÁS

Esta técnica (como cualquier otra de diseño de algoritmos) no se puede aplicar a todo tipo de problemas. En algunos casos se puede aplicar pero no da lugar a soluciones óptimas y en otros casos sí, no existiendo una clasificación de problemas en los que conviene aplicarla, por lo que no intentamos dar una clasificación de problemas que se resuelvan por una técnica u otra, sino unas técnicas de las más usadas, que hay que conocer para poder decidir ante un problema concreto si consideramos conveniente aplicar una determinada técnica, una mezcla de varias o ninguna de ellas.

3.1 Método general

La técnica **divide y vencerás** consiste en intentar resolver un problema dividiéndolo en k subproblemas que se resuelvan más fácilmente, resolverlos, y combinar los resultados de los subproblemas para obtener una solución del problema original.

3.1.1 Esquema general

Un esquema general de la aplicación de la técnica puede ser:

Algoritmo 3.1 *Esquema general de la técnica* **divide y vencerás***.*

```
divide_venceras(p:problema):
    dividir(p, p1, p2, ..., pk)  /*pi son subproblemas de p*/
    para i = 1, 2, ..., k
        si=resolver(pi)
    finpara
    return(combinar(s1, s2, ..., sk))
```

El ejemplo visto de las torres de Hanoi puede considerarse un método divide y vencerás pues la solución del problema de n aros se resuelve resolviendo dos de $n-1$ aros que a su vez se resuelven en función de otros menores. En este caso no hay combinación de los resultados sino que ésta se obtiene variando los palos origen, destino y pivote, y haciendo un movimiento de un aro.

3.1.2 Esquema recursivo

Un caso muy corriente es tener los datos de entrada en un array global con índices entre i_1 e i_n y que se resuelva el problema dividiéndolo de manera recursiva en dos subproblemas de tamaño la mitad hasta llegar a un caso base que se resuelve directamente. Un esquema en este caso será:

Algoritmo 3.2 *Esquema de la técnica* **divide y vencerás** *cuando se divide el problema recursivamente en dos subproblemas de igual tamaño hasta llegar al caso base.*

```
divide_venceras(p, q:indice):
var m:indice
   si pequeño(p, q)
      return(solucion(p, q))
   en otro caso
      m=dividir(p, q)
      return(combinar(divide_venceras(p, m), divide_venceras(m + 1, q)))
   finsi
```

donde:

- **pequeño** es una función que determina si el tamaño es suficientemente pequeño para resolver el problema directamente,
- **solución** genera una solución con unos índices dados cuando el problema es suficientemente pequeño,
- **dividir** obtiene el índice, entre dos dados, por el que dividir el problema,
- **combinar** combina dos resultados de subproblemas para obtener el resultado del problema.

Como se ve en este esquema, puede pasar que los subproblemas en que dividimos el problema inicial no sean suficientemente pequeños para convenir resolverlos directamente, por lo que el proceso de dividir en subproblemas puede realizarse repetidamente.

Estudio Analizamos el tiempo de ejecución de un algoritmo con este esquema suponiendo que n es potencia de dos y que un problema se divide en dos subproblemas de igual dimensión. Se obtendrá la recurrencia:

$$t(n) = \begin{cases} g(n) & \text{si n es suficientemente pequeño} \\ 2t\left(\frac{n}{2}\right) + f(n) & \text{en otro caso} \end{cases} \tag{3.1}$$

donde $g(n)$ es el tiempo de generar la solución y $f(n)$ el de combinar los resultados (consideramos despreciables los tiempos de los procedimientos pequeño y dividir).

Desarrollando obtenemos:

$$\begin{aligned} t\left(2^k\right) = 2t\left(2^{k-1}\right) + f\left(2^k\right) = \\ 2\left(2t\left(2^{k-2}\right) + f\left(2^{k-1}\right)\right) + f\left(2^k\right) = \ldots = \\ 2^m g\left(2^{k-m}\right) + \sum_{i=0}^{m-1}\left(2^i f\left(2^{k-i}\right)\right) \end{aligned} \tag{3.2}$$

Utilizando esta fórmula podemos ver que en algunos casos el utilizar esta técnica en vez de resolver el problema directamente mejora el tiempo de ejecución, y en otros casos no lo mejora.

Ejemplo 3.1 Si $g(n) = c$ y $f(n) = d$ con c y d constantes positivas, si dividimos el problema hasta problemas de tamaño 1 tendremos (sustituyendo en la ecuación 3.2) el tiempo:

$$t(n) = nc + \sum_{i=0}^{k-1}\left(2^i d\right) = nc + d(n-1) \tag{3.3}$$

con lo que $t \in \theta(n)$.

Pero si resolvemos el problema directamente el tiempo es $g(n) = c \in \theta(1)$, con lo que es conveniente resolver el problema directamente.

Ejemplo 3.2 Si $g(n) = cn^2$ y $f(n) = dn$ con c y d constantes positivas, si dividimos el problema hasta problemas de tamaño 1 tendremos (sustituyendo en la ecuación 3.2) el tiempo:

$$\begin{aligned} t(n) = ng(1) + \sum_{i=0}^{k-1}\left(2^i f\left(2^{k-i}\right)\right) = nc + \sum_{i=0}^{k-1}\left(2^i d 2^{k-i}\right) = \\ nc + d2^k k = nc + dn\log n \end{aligned} \tag{3.4}$$

con lo que $t \in \theta(n \log n)$.

Pero si resolvemos el problema directamente el tiempo es $g(n) = cn^2 \in \theta\left(n^2\right)$, con lo que es conveniente resolver el problema por divide y vencerás.

Este es el caso, por ejemplo, de la ordenación por mezcla.

Vemos que si $g(n) = cn^3$ (o cualquier otra función de mayor orden) el tiempo de resolver el problema por divide y vencerás sigue siendo $\theta(n \log n)$ ya que, independientemente del valor de la función g, el caso base tiene un tiempo constante.

3.2 Búsqueda del máximo y mínimo

Estudiaremos el ejemplo del cálculo del máximo y mínimo de los elementos almacenados en un array. Este ejemplo es muy sencillo y sólo tiene interés para hacer un estudio detallado de los tiempos de ejecución.

3.2.1 Método directo

Un algoritmo para resolver el problema sin aplicar la técnica divide y vencerás puede ser:

Algoritmo 3.3 *Cálculo del máximo y mínimo, método directo.*

PROCEDURE MaxMin(a:ARRAY[1..n] OF tipo;VAR max, min:tipo)
VAR
  i:INTEGER
BEGIN
  $max = a[1]$
  $min = a[1]$
  FOR $i = 2$ TO n
    IF $a[i] > max$
      $max = a[i]$
    ELSIF $a[i] < min$
      $min = a[i]$
    ENDIF
  ENDFOR
ENDMaxMin

Para estudiar este algoritmo lo vamos a hacer contando el número de asignaciones y de comparaciones de elementos del tipo "tipo" (pues puede ocurrir que para elementos de este tipo tarde mucho más una asignación que una comparación o al revés) en el caso más favorable, más desfavorable o en promedio.

Comparaciones El número de comparaciones es $n-1$ cuando $a[i] > max$, $\forall i$ entre 2 y n, por lo que el caso más favorable para el número de comparaciones es que el array esté ordenado de menor a mayor.

El caso en que $a[i] < max$, $\forall i$ entre 2 y n será el más desfavorable, y se hacen $2(n-1)$ comparaciones, y corresponde a que el máximo sea $a[1]$.

Para estudiar el número de comparaciones en promedio hay que tener en cuenta que los n números pueden estar ordenados de $n!$ maneras y que suponemos que todas las ordenaciones tienen la misma probabilidad de aparecer. Con $i = 2$, $probabilidad(a[i] > max) = \frac{1}{2}$ y $probabilidad(a[i] < max) = \frac{1}{2}$; con $i = 3$, $probabilidad(a[i] > max) = \frac{1}{3}$ y $probabilidad(a[i] < max) = \frac{2}{3}$; y en general, $probabilidad(a[i] > max) = \frac{1}{i}$ y

$probabilidad(a[i] < max) = \frac{i-1}{i}$. Como cuando $a[i] > max$ se hace una comparación y cuando $a[i] < max$ se hacen dos, el número medio de comparaciones es

$$\sum_{i=2}^{n} \left(\frac{1}{i} + 2\frac{i-1}{i} \right) = \sum_{i=2}^{n} \left(2 - \frac{1}{i} \right) =$$
$$2(n-1) - \sum_{i=2}^{n} \frac{1}{i} \simeq 2(n-1) - \int_{1}^{n} \frac{1}{x} dx = 2(n-1) - \ln n \qquad (3.5)$$

Hemos obtenido el número de comparaciones en el caso más favorable, más desfavorable y promedio y en los tres son del orden de n, con lo que el número de comparaciones será $\theta(n)$.

Asignaciones Para las asignaciones el caso más favorable es que no sea nunca $a[i] > max$ ni $a[i] < min$, con lo que el máximo y el mínimo debe ser $a[1]$. En este caso se hacen sólo las dos asignaciones del principio.

El caso más desfavorable será que se haga siempre una asignación del interior del FOR (no puede ocurrir que se hagan las dos), con lo que tendremos $n+1$ asignaciones.

En el caso promedio, como $a[i] > max$ con probabilidad $\frac{1}{i}$, se ejecuta $max = a[i]$ un número medio de veces dado por $\sum_{i=2}^{n} \frac{1}{i}$, y como $a[i] < min$ también con probabilidad $\frac{1}{i}$, la misma expresión nos sirve para el número medio de veces que se ejecuta $min = a[i]$, con lo que el número medio de asignaciones es $2 + 2 \ln n$.

En el caso de las asignaciones tenemos que están entre $\Omega(1)$ y $O(n)$, y que el promedio pertenece a $\theta(\log n)$.

La complejidad en promedio del algoritmo será $\theta(n)$, pues éste domina sobre $\theta(\log n)$.

3.2.2 Con Divide y vencerás

Aplicando la técnica divide y vencerás se divide el problema por la mitad hasta que tengamos uno o dos elementos:

Algoritmo 3.4 *Cálculo del máximo y mínimo, con divide y vencerás.*

```
PROCEDURE MaxMinDV(a:ARRAY[1..n] OF tipo;i, j:INTEGER;
   VAR max, min:tipo)
BEGIN
   IF i < j − 1
      mit = (i + j) DIV 2
      MaxMinDV(a, i, mit, max1, min1)
      MaxMinDV(a, mit + 1, j, max2, min2)
      IF max1 > max2
         max = max1
```

```
        ELSE
            max = max2
        ENDIF
        IF min1 < min2
            min = min1
        ELSE
            min = min2
        ENDIF
    ELSIF i = j - 1
        IF a[i] < a[j]
            max = a[j]
            min = a[i]
        ELSE
            max = a[i]
            min = a[j]
        ENDIF
    ELSE
        max = a[i]
        min = max
    ENDIF
ENDMaxMinDV
```

En este caso, para calcular el tiempo de ejecución, podemos suponer que n es una potencia de dos y aplicar después el teorema 2.1 para quitar esta restricción.

Comparaciones Para calcular el número de comparaciones tenemos la ecuación de recurrencia $t(n) = 2t\left(\frac{n}{2}\right) + 2$, y haciendo el cambio $k = \log n$ tenemos $t_k = 2t_{k-1} + 2$, cuya ecuación característica es $(x-2)(x-1) = 0$, y la solución general $t(n) = c_1 n + c_2$, y como $t(2) = 1$ y $t(4) = 4$ tendremos:

$$
\begin{aligned}
1 &= 2c_1 + c_2 \\
4 &= 4c_1 + c_2
\end{aligned}
$$

y resolviendo el sistema $t(n) = \frac{3}{2}n - 2$.

Si para calcular los parámetros consideramos que podemos llegar al caso base 1 ó 2 tendremos:

$$
\begin{aligned}
0 &= c_1 + c_2 \\
2 &= 2c_1 + c_2
\end{aligned}
$$

y $c_1 = 2$, $c_2 = -2$, por lo que $t(n) = 2n - 2$.

Suponiendo distintos casos base obtenemos tiempos de ejecución distintos aunque el mismo orden. ¿Qué tiempo es el correcto? Si suponemos que n es potencia de dos el correcto será el primero que hemos calculado ya que siempre llegaremos al caso base de tamaño 2. Este análisis nos permite comprobar que si llegamos al caso base 1 tendremos mayor tiempo de ejecución, con lo que si en el programa quitamos el caso base de tamaño 2 llegaremos siempre al de tamaño 1 y el programa será un 33% más lento. De este modo, en algunos casos el variar el tamaño del caso base nos permite una reducción en el tiempo de ejecución manteniendo el mismo orden.

Asignaciones Para calcular el número de asignaciones tendríamos la misma ecuación de recurrencia, pero $t(2) = 2$ y $t(4) = 6$, con lo que obtenemos $t(n) = 2n - 2$.

3.2.3 Comparación

Hay que hacer varias consideraciones:

- En el caso de MaxMinDv hemos calculado el número de comparaciones y asignaciones de elementos del tipo "tipo" sólo para el caso en que n sea potencia de dos. En otro caso podemos deducir (utilizando el teorema mencionado) que el tiempo es $\theta(n)$, pero no obtenemos la fórmula exacta.

- Aunque los dos métodos utilizados tengan el mismo θ, dependiendo del coste de las operaciones de comparación y asignación un método puede ser preferible a otro. Si consideramos que las asignaciones son las operaciones más costosas, el método directo puede ser preferible. Si las comparaciones son mucho más costosas que las asignaciones (podría pasar si los datos están en listas almacenados como variables dinámicas y la comparación se hace comparando elemento a elemento de la lista y la asignación simplemente asignando putneros) puede ser preferible el método divide y vnecerás, pues el coste de las comparaciones puede ser $o\left(\frac{3}{2}n\right)$ y en el método directo $o\left(2n\right)$

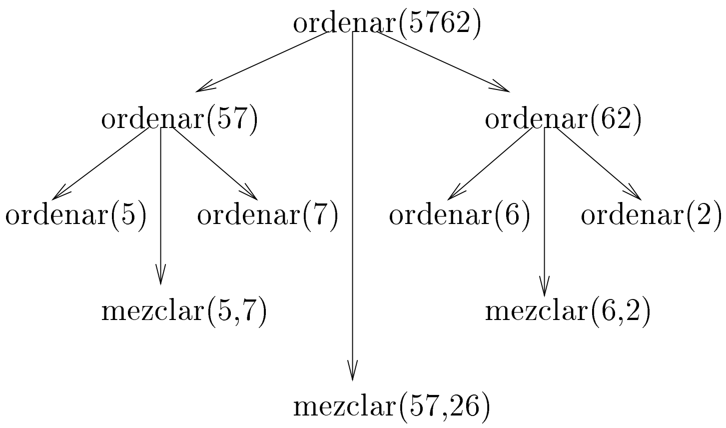
- Hay que tener en cuenta también que el método divide y vencerás es recursivo, lo que puede producir un mayor tiempo de ejecución de gestionar las llamadas recursivas, y que estas se gestionarán más o menos eficientemente dependiendo del compilador con que trabajemos.

3.3 Ordenación por mezcla

3.3.1 Descripción

Consideramos la ordenación de n datos almacenados en un array con índices entre 1 y n. Los datos se ordenan por un método divide y vencerás, dividiendo el problema en dos subproblemas de igual tamaño (si n es potencia de dos) hasta llegar al caso base que será tener un único elemento. En el caso base los datos (el dato) están ordenados, con lo que la solución del problema consiste en no hacer nada. Una vez que tenemos dos subarrays ordenados hay que mezclarlos para obtener ordenado el array formado por esos dos subarrays, con lo que el procedimiento de combinación consiste en una mezcla de dos arrays ordenados.

Ejemplo 3.3 Vemos el funcionamiento del método con un ejemplo que se muestra en la siguiente figura:



donde ordenamos los datos 5, 7, 6 y 2.

3.3.2 Algoritmo

Un algoritmo ajustándonos al esquema divide y vencerás puede ser:

Algoritmo 3.5 PROCEDURE Mergesort(p, q:INTEGER)
  VAR
    m:INTEGER
  BEGIN
    IF $p < q$
      $m = (p + q)\ DIV\ 2$
      Mergesort(p, m)

```
        Mergesort(m + 1, q)
        Merge(p, m, q)
    ENDIF
ENDMergesort
```

donde Merge es:

```
PROCEDURE Merge(p, m, q:INTEGER)
VAR
    b:ARRAY[1..n] OF tipo
    h, i, j, k:INTEGER
BEGIN
    h = p
    i = p
    j = m + 1
    WHILE h ≤ m AND j ≤ q
        IF a[h] ≤ a[j]
            b[i] = a[h]
            h = h + 1
        ELSE
            b[i] = a[j]
            j = j + 1
        ENDIF
        i = i + 1
    ENDWHILE
    IF h > m
        FOR k = j TO q
            b[i] = a[k]
            i = i + 1
        ENDFOR
    ELSE
        FOR k = h TO m
            b[i] = a[k]
            i = i + 1
        ENDFOR
    ENDIF
    FOR k = p TO q
        a[k] = b[k]
    ENDFOR
ENDMerge
```

En este algoritmo los procedimientos del esquema del método divide y vencerás son:

- **pequeño**. Es la comprobación de que $p \geq q$.
- **solucion**. Es no hacer nada ya que un único dato es una secuencia ordenada.
- **dividir**. Es $m = (p + q)\ DIV\ 2$.
- **combinar**. Es el procedimiento Merge.

El algoritmo no es muy bueno por al menos dos motivos:

- Después de la mezcla se hace una copia de datos de un array temporal b en el array a. Esto se puede evitar poniendo un parámetro booleano en el procedimiento Mergesort que indique si hay que mezclar de a en b o de b en a.
- El que el método sea recursivo nos puede producir un mayor tiempo de ejecución y una mayor ocupación de memoria que si no lo fuera.

3.3.3 Estudio

Para estudiar el tiempo de ejecución consideramos n potencia de dos y contaremos las instrucciones que se ejecutan.

Empezamos analizando Merge:

Se ejecutan las tres inicializaciones, y después se entra en el bucle WHILE ejecutándose su cuerpo un número mínimo de veces $\frac{n}{2}$ y un número máximo $n-1$. Si llamamos $\frac{n}{2}+l$ al número de veces que se pasa por el cuerpo de WHILE tendremos un coste $5\left(\frac{n}{2}+l\right)$ del WHILE, y un coste $3\left(\frac{n}{2}-l\right)$ de la copia de los elementos que quedan en uno de los dos subarrays, y un coste $2n$ de la copia de b en a. Por tanto tenemos que $t(n) = 3 + 6n + 2l$, y como l está entre 0 y $\frac{n}{2}-1$, $t(n) \in \theta(n)$.

Para calcular el coste del Mergesort tenemos la recurrencia:

$$t(n) = \begin{cases} a & si\ n = 1 \\ 2t\left(\frac{n}{2}\right) + bn + c & si\ n > 1 \end{cases}$$

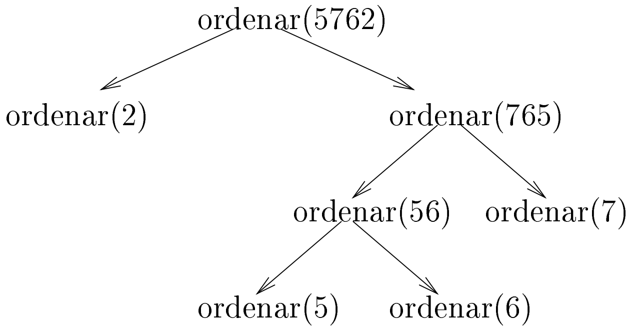
y resolviendo la ecuación se tiene $t(n) = bn \log n + (a+c)n - c$, y el orden será $\theta\left(n \log n\right)$.

3.4 Ordenación rápida

3.4.1 Descripción

En este caso un problema se divide en dos subproblemas pero estos no tienen por qué ser de igual tamaño. Dado el array a de índices p a q, se toma un elemento y se obtienen los menores que él almacenándolos en a en las posiciones de la p a la m, y los mayores o iguales almacenándolos en las posiciones de la $m+1$ a la q.

Ejemplo 3.4 Para ordenar los datos 5, 7, 6 y 2 podríamos tener un árbol de llamadas como el siguiente:



donde en cada paso se dividen los elementos en dos subconjuntos con todos los datos en el primer subconjunto menores o iguales que todos los del segundo subconjunto. Según la estrategia utilizada para elegir el pivote variará cómo se hace la división de un conjunto en dos subconjuntos.

3.4.2 Algoritmo

Un algoritmo puede ser:

Algoritmo 3.6 PROCEDURE Quicksort(p, q:INTEGER)

```
VAR
    j:INTEGER
BEGIN
    IF p < q
        j = q + 1
        particion(p, j)
        Quicksort(p, j - 1)
        Quicksort(j + 1, q)
    ENDIF
ENDQuicksort
```

donde el procedimiento partición es:

```
PROCEDURE particion(izq:INTEGER;VAR d:INTEGER)
VAR
    temp, v:tipo
    i:INTEGER
BEGIN
    v = a[izq]
    i = izq
```

```
    WHILE i < d
        REPEAT
            i = i + 1
        UNTIL a[i] ≥ v
        REPEAT
            d = d − 1
        UNTIL a[d] ≤ v
        IF i < d
            temp = a[i]
            a[i] = a[d]
            a[d] = temp
        ENDIF
    ENDWHILE
    a[izq] = a[d]
    a[d] = v
ENDparticion
```

En este algoritmo los procedimientos del esquema del método divide y vencerás son:

- **pequeño**. Es la comprobación de que $p \geq q$.
- **solucion**. Es no hacer nada ya que el caso base consta de un único elemento.
- **dividir**. Es particionar. En este caso en la división del problema, además de obtener el índice por el que se particiona se hace una ordenación relativa.
- **combinar**. No se hace combinación ya que la ordenación relativa que se hace en particionar nos asegura que el array quede ordenado al final.

También es este caso se podría evitar la recursión para hacerlo más eficiente.

3.4.3 Estudio

Con este algoritmo el caso más desfavorable se presenta cuando los datos están inversamente o directamente ordenados. En estos casos la recurrencia que se tiene es:

$$t(n) = \begin{cases} c & si\ n = 1 \\ t(n-1) + an + b & si\ n > 1 \end{cases}$$

y expandiendo la recurrencia se llega a $t(n) = c + a\frac{2+n}{2}(n-1) + b(n-1)$, y $t_M \in \theta\left(n^2\right)$ y $t \in O\left(n^2\right)$.

Algo más complicado es obtener el tiempo promedio, ya que en general la recurrencia es de la forma:

$$t(n) = \begin{cases} c & si\ n = 1 \\ t(n-i) + t(i-1) + an + b & si\ n > 1 \end{cases}$$

pudiendo tomar i un valor entre 1 y n correspondiente a las n posiciones en que puede quedar el elemento pivote después de la ejecución del procedimiento particion.

Para calcular el coste promedio consideraremos las comparaciones, y que el elemento pivote puede ir a parar con la misma probabilidad a cada una de las n posiciones del array. De esta manera tenemos la fórmula (cambiamos todas las constantes por unos para simplificar la expresión):

$$C(n) = n + 1 + \frac{1}{n}\sum_{i=1}^{n}\left(C(i-1) + C(n-i)\right) \tag{3.6}$$

y multiplicando por n:

$$nC(n) = n(n+1) + 2\left(C(0) + C(1) + \ldots + C(n-1)\right) \tag{3.7}$$

Sustituyendo en 3.7 n por $n-1$ queda:

$$(n-1)C(n-1) = (n-1)n + 2\left(C(0) + \ldots + C(n-2)\right) \tag{3.8}$$

Restando a 3.7 la ecuación 3.8, agrupando y dividiendo por $n(n+1)$ queda:

$$\frac{C(n)}{n+1} - \frac{C(n-1)}{n} = \frac{2}{n+1} \tag{3.9}$$

y desarrollando llegamos a:

$$\frac{C(n)}{n+1} = \frac{2}{n+1} + \frac{2}{n} + \ldots + \frac{1}{2} \simeq 2\int_{1}^{n+1}\frac{1}{x}dx = 2\ln(n+1) \tag{3.10}$$

con lo que $C(n) \in \theta(n \log n)$.

3.5 Multiplicación rápida de enteros largos

3.5.1 Método directo

Cuando se implementa el tipo **EnteroLargo** como:

```
TYPE EnteroLargo=POINTER TO nodo;
  nodo=RECORD
    item:INTEGER;
    sig:EnteroLargo;
  END;
```

donde cada EnteroLargo está compuesto por una serie de enteros de un tamaño determinado (vamos a suponer 1) almacenados en nodos distintos; la multiplicación de dos enteros largos se puede hacer multiplicando todos los nodos de un número por todos los de otro, lo que da una complejidad de nm con n y m las longitudes de los dos enteros largos, y a este tiempo hay que sumarle los tiempos de las sumas.

Para disminuir el tiempo de ejecución se puede utilizar un método divide y vencerás. Suponemos los enteros largos x e y de longitud n, y los descomponemos en dos partes iguales siendo:

$$\begin{aligned} x &= a\ 10^{\frac{n}{2}} + b \\ y &= c\ 10^{\frac{n}{2}} + d \end{aligned}$$

siendo:

$$xy = ac\ 10^n + (ad + cb)\ 10^{\frac{n}{2}} + bd \tag{3.11}$$

en cuyo caso tendremos que hacer 4 multiplicaciones de tamaño $\frac{n}{2}$ y las sumas, con lo que el tiempo de ejecución viene dado por la recurrencia:

$$t(n) = \begin{cases} c & si\ n = 1 \\ 4t\left(\frac{n}{2}\right) + bn & si\ n > 1 \end{cases} \tag{3.12}$$

de donde $t(n) = (c+b)n^2 - bn$, que da un tiempo de ejecución de orden $\theta\left(n^2\right)$, con lo que el método divide y vencerás por sí sólo no mejora el tiempo de ejecución de un método directo.

3.5.2 Multiplicación rápida

El método de Strassen consiste en modificar la fórmula de la multiplicación (ecuación 3.11) obteniendo la nueva fórmula:

$$xy = ac\ 10^n + ((a-b)(d-c) + ac + bd)\ 10^{\frac{n}{2}} + bd \tag{3.13}$$

con lo que la ecuación de recurrencia queda:

$$t(n) = \begin{cases} c & si\ n = 1 \\ 3t\left(\frac{n}{2}\right) + bn & si\ n > 1 \end{cases} \tag{3.14}$$

de donde desarrollando tenemos:

$$\begin{aligned} t(n) &= 3\left(3t\left(\frac{n}{2^2}\right) + b\frac{n}{2}\right) + bn = \\ \ldots &= 3^k t(1) + bn\frac{\left(\frac{3}{2}\right)^k - 1}{\frac{3}{2} - 1} = \end{aligned}$$

$$3^{\log n}c + 2bn\left(\frac{3^{\log n}}{2^{\log n}} - 1\right) =$$
$$n^{\log 3}c + 2b\left(n^{\log 3} - n\right)$$

con lo que el tiempo de ejecución es del orden $n^{\log 3} \simeq n^{1.59}$.

3.5.3 Implementación

La forma de implementar la multiplicación de enteros largos podría ser implementar el tipo en un módulo de manera que los procedimientos que aparecen a continuación deberían estar en el módulo para poder hacer la multiplicación:

Algoritmo 3.7 *Multiplicación de enteros largos por el método de Strassen.*

```
PROCEDURE Mult(x, y:EnteroLargo;n:INTEGER):EnteroLargo
  IF n ≤ b
    RETURN MultBasica(x, y)
  ELSE
    asignar(a,primeros(n/2, x))
    asignar(b,ultimos(n/2, x))
    asignar(c,primeros(n/2, y))
    asignar(d,ultimos(n/2, y))
    asignar(m1,Mult(a, c, n/2))
    asignar(m2,Mult(b, d, n/2))
    asignar(m3,Mult(restar(a, b), restar(d, c),n/2 ))
    RETURN sumar(sumar(Mult10(m1, n),
                       Mult10(sumar(sumar(m1, m2),m3),n/2)),m2)
  ENDIF
```

donde:

- *asignar*, hará la asignación entre dos enteros largos.
- *ultimos*, obtendrá los últimos dígitos de un entero largo.
- *primeros*, obtendrá los primeros dígitos de un entero largo.
- *MultBasica*, hará la multiplicación en el caso básico en que tengamos números de menos de b dígitos.
- *restar* y *sumar*, harán la resta y suma de enteros largos.
- *Mult10*, hará la multiplicación en el caso especial de añadir ceros al final del número.

En este esquema la función **pequeño** se corresponde con la comprobación $n \leq b$, la función **solución** es *MultBasica*. **dividir** es algo más complicado, pues se dividen los dos enteros largos en cuatro enteros (a, b, c y d) de longitud la mitad usando las funciones *primeros* y *ultimos*. **combinar** consiste en llevar a cabo las operaciones indicadas en la fórmula 3.13.

3.6 Multiplicación rápida de matrices

3.6.1 Método directo

Si tenemos dos matrices A y B de dimensión $n \times n$ y queremos obtener $C = AB$ habrá que calcular los n^2 elementos de C, necesitando para cada uno de ellos n multiplicaciones y $n-1$ sumas, por lo que la complejidad del algoritmo sería $n^2(2n-1)$, lo que supone un orden $\theta\left(n^3\right)$.

Si cada una de las matrices A y B se divide en cuatro submatrices de dimensión $\frac{n}{2} \times \frac{n}{2}$ se puede aplicar el esquema del divide y vencerás, teniendo 8 multiplicaicones de matrices y 4 sumas de matrices de dimensión $\frac{n}{2} \times \frac{n}{2}$. La división de las matrices se hace recursivamente hasta llegar a un determinado tamaño del caso base (b), con el que se utiliza el método directo. La ecuación de recurrencia queda:

$$t(n) = \begin{cases} 2n^3 & si\ n \leq b \\ 8t\left(\frac{n}{2}\right) + n^2 & si\ n > b \end{cases} \tag{3.15}$$

si contamos el número de operaciones de suma o resta y multiplicación. Resolviendo la ecuación de recurrencia tenemos $t(n) = c_1 n^3 + c_2 n^2$, y si obtenemos c_1 y c_2 imponiendo los casos base b y $2b$ queda $t(n) = \left(2 + \frac{1}{b}\right) n^3 - n^2$, con lo que el método divide y vencerás por sí sólo no mejora asintóticamente el tiempo de ejecución de un método directo.

3.6.2 Multiplicación rápida

El algoritmo de Strassen se basa también en la división de las matrices A y B en la forma:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \qquad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \tag{3.16}$$

pero reduciendo el número de multiplicaciones de ocho a siete y aumentando el de sumas de cuatro a dieciocho utilizando las fórmulas:

$$
\begin{array}{rcl}
P & = & (A_{11}+A_{22})\,(B_{11}+B_{22}) \\
Q & = & (A_{21}+A_{22})\,B_{11} \\
R & = & A_{11}\,(B_{12}-B_{22}) \\
S & = & A_{22}\,(B_{21}-B_{11}) \\
T & = & (A_{11}+A_{12})\,B_{22} \\
U & = & (A_{21}-A_{11})\,(B_{11}+B_{12}) \\
V & = & (A_{12}-A_{22})\,(B_{21}+B_{22}) \\
C_{11} & = & P+S-T+V \\
C_{12} & = & R+T \\
C_{21} & = & Q+S \\
C_{22} & = & P+R-Q+U
\end{array}
$$

con lo que la ecuación de recurrencia, si consideramos el tamaño del caso base b y contamos el número de operaciones aritméticas, queda:

$$
t(n) = \begin{cases} 2n^3 & si\ n \leq b \\ 7t\left(\frac{n}{2}\right) + \frac{9}{2}n^2 & si\ n > b \end{cases} \tag{3.17}
$$

de donde, resolviendo la ecuación de recurrencia e imponiendo los casos base b y $2b$, obtenemos $t(n) = \left(2b^{0.19} + \frac{6}{b^{0.81}}\right) n^{2.81} - 6n^2$.

El método de multiplicación rápida es más rápido que la multiplicación normal asintóticamente, pues su orden es $n^{2.81}$, pero el número de operaciones aritméticas varía al variar el tamaño del caso base. El tamaño óptimo del caso base se puede estimar teórica o experimentalmente, pero las constantes y los términos de menor orden tienen una gran importancia para tamaños pequeños, y el método divide y vencerás es más difícil de programar de manera eficiente. Todo esto puede producir que, en la práctica, el método directo sea más rápido que el divide y vencerás para tamaños de problema grandes, y el punto en que el método divide y vencerás pasa a ser mejor que el directo depende de factores como el lenguaje o compilador utilizado y la destreza del programador.

Se pueden conseguir mejoras dividiendo el tamaño de las matrices por valores superiores a dos, y hay métodos distintos, como por ejemplo uno debido a Schönage, de orden $n^{2.73}$, otro de Victor Pan, de orden $n^{2.61}$, e incluso de orden $n^{2.3755}$. A pesar de que estos métodos sean mejores asintóticamente nos encontramos otra vez con la direrencia en términos de menor orden y en las constantes, lo que hace que sean mejores para tamaños de matriz demasiado grandes como para considerar aplicarlos en la práctica.

3.7 Problemas

Problema 3.1 Suponiendo que tenemos un array de n enteros (siendo n una potencia de dos), hacer un procedimiento que calcule la media de los elementos del array según un esquema divide y vencerás. ¿Se puede generalizar al caso en que n no sea potencia de 2?

Problema 3.2 Realizar un algoritmo que utilizando la técnica de divide y vencerás encuentre el mayor y segundo mayor elemento de un array.

Solución:

Haremos un procedimiento al que mandaremos los índices del array sobre los que estamos trabajando, y dos valores M y m que contendrń el máximo y el segundo mayor. M y m serán variables pues hay que devolverlos al procedimiento que los llamó para combinarlos:

```
procedure maximos(p, q:indices;var M, m:datos)
var
    mit:indice
    M1, M2, m1, m2:datos
begin
    si p = q
        M = a[p]
        m = −∞
    en otro caso si q − p = 1
        si a[p] > a[q]
            M = a[p]
            m = a[q]
        en otro caso
            M = a[q]
            m = a[p]
        finsi
     en otro caso /*No es suficientemente pequeño*/
        mit = (p + q) div 2
        maximos(p, mit, M1, m1)
        maximos(mit + 1, q, M2, m2)
        /*Se combinan los resultados:*/
        si M1 > M2
            M = M1
            si M2 > m1
                m = M2
            en otro caso
                m = m1
            finsi
```

en otro caso
$M = M_2$
si $M_1 > m_2$
$m = M_1$
en otro caso
$m = m_2$
finsi
finsi
finsi
finmaximos

La primera llamada se haría con maximos$(1, n, M, m)$ estando los datos en un array de índices de 1 a n.

Problema 3.3 Tenemos datos de un cierto tipo almacenados en una lista de arrays (cada dato en un elemento de un array de un nodo de la lista) y se realiza la ordenación de los datos en dos fases: en la primera se ordenan todos los arrays por el método Quicksort, y en la segunda se ordenan los elementos mezclando primero el nodo uno y dos, después el resultado de mezclar los dos primeros nodos con el tercero, y así sucesivamente. Estudiar el tiempo de ejecución del algoritmo resultante en función del tamaño de los arrays y del número de datos a ordenar.

Solución:

Suponemos que tenemos n datos a ordenar y que t es el tamaño de los arrays. Tendremos $\frac{n}{t}$ nodos (tomando la parte entera superior, pero esto no nos influirá en el orden de ejecución). La ordenación de cada nodo por Quicksort lleva un tiempo del orden de $t \log t$ en promedio, y t^2 en el caso más desfavorable.

En la segunda fase se mezclan el nodo 1 y 2, lo que tiene un orden $2t$ en cualquier caso, después el resultados de mezclar el 1 y 2 se mezcla con el 3, lo que tiene un orden $3t$, y así sucesivamente hasta mezclar los nodos del primero al penúltimo con el último, lo que tendrá un coste $\frac{n}{t}t$, por lo que la segunda fase del algoritmo tiene un orden:

$$2t + 3t + \ldots + \frac{n}{t}t = t\left(2 + \frac{n}{t}\right)\frac{\frac{n}{t} - 1}{2} =$$
$$(2t + n)\frac{n - t}{2t}$$

y el orden total será en promedio:

$$n \log t + \frac{n^2 + nt - 2t^2}{2t}$$

y en el caso más desfavorable:

$$nt + \frac{n^2 + nt - 2t^2}{2t}$$

Problema 3.4 Dadas dos funciones f y g como en el método Divide y vencerás, ecuación 3.1. ¿Se cumple que si $O(f) < O(g)$ entonces es mejor aplicar Divide y vencerás que el método directo? Encontrar un contraejemplo.

Problema 3.5 Dado un array de N números hacer un programa que utilizando la técnica divide y vencerás encuentre una cadena de n números en el array con la que se obtenga una suma máxima. El algoritmo debe ser de coste $t(N, n) \in \theta(N)$.

Solución:

Para hacer un algoritmo sin la técnica divide y vencerás que resuelva el problema en un tiempo $t(N, n) \in \theta(N)$ hay que tener en cuenta que no se puede empezar en cada posición de la serie (de la posición 1 a la $N - n$) y sumar n números a partir de cada posición, pues el coste sería $n(N - n)$ y queremos que el coste no dependa de n. El problema con esta aproximación sería que hacemos trabajo de más pues si sumamos los números en las posiciones 1, 2, ..., n, y después los números en las 2, 3, ..., $n+1$, estamos volviendo a sumar los números en las posiciones 2, 3, ..., n (¡trabajo que ya habíamos hecho!). Para evitar este problema podemos resolverlo sumando los n primeros números (coste $n < N$) y a partir de ahí recorrer las posiciones de la $n+1$ a la N restando a la suma anterior el número $x[i-n]$ y sumando el $x[i]$. De este modo, un programa para resolver el problema con coste N sería:

```
s = 0  /*Suma actual*/
k = 1  /*Posición de inicio*/
para j = 1, 2, ..., n /*Suma de los n primeros datos del array x*/
    s = s + x[j]
finpara
so = s  /*Almacenamos la suma en una suma óptima actual*/
para j = n + 1, n + 2, ..., N /*Cálculo del resto de las sumas*/
    s = s - x[j - n] + x[j]
    si s > so
        so = s
        k = j - n + 1
    finsi
finpara
```

¿Esta manera de resolver el problema se puede considerar un divide y vencerás? Tenemos la resolución de un subproblema de tamaño n en el primer bucle, y la resolución de $N-n$ subproblemas de tamaño n (todos resueltos del mismo modo y utilizando la solución de los subproblemas anteriores) en el segundo bucle, y la combinación de los subproblemas se va haciendo conforme se va resolviendo cada subproblema. De este modo, podríamos considerar esta solución como un divide y vencerás muy alejado del esquema visto en clase.

Una solución más aproximada al esquema visto en clase podría ser dividir el problema en dos subproblemas de tamaño $\frac{N}{2}$ (o en un número constante c de subproblemas de tamaño $\frac{N}{c}$), resolver cada uno de los subproblemas con el método anterior, lo que nos daría un coste $\frac{N}{2}$ en cada subproblema y N en total, y hacer una combinación que no aumente el coste. De esta manera el esquema sería:

D_V:
    resolver con el método anterior con índices 1 a $\frac{N}{2}$ y obtener resultados en s_1 y k_1
    resolver con el método anterior con índices $\frac{N}{2}+1$ a N y obtener resultados en s_2 y k_2
    combinar$(x, N, n, s_1, k_1, s_2, k_2, s, k)$

Donde en combinar utilizamos como entradas los números, los valores de N y n (x, N, n) y las soluciones de los subproblemas resueltos (s_1, k_1, s_2, k_2), y se devuelve como resultado la solución global (s, k). El esquema de combinar sería similar al del algoritmo sin divide y vencerás, y se trata de hacer las sumas de las secuencias de n números de las que no se ha hecho la suma en ninguno de los subproblemas, y comparar los resultados que vamos obteniendo con los mejores de los obtenidos en los subproblemas:

combinar:
    si $s_1 > s_2$
        $s = s_1$
        $k = k_1$
    en otro caso
        $s = s_2$
        $k = k_2$
    finsi
    $s_1 = 0$
    para $j = \frac{N}{2} - n + 2, \frac{N}{2} - n + 3, \dots, \frac{N}{2} + 1$
        $s_1 = s_1 + x[j]$
    finpara
    si $s_1 > s$
        $s = s_1$
        $k = \frac{N}{2} - n + 2$
    finsi
    para $j = \frac{N}{2} - n + 3, \frac{N}{2} - n + 4, \dots, \frac{N}{2} - 1$
        $s_1 = s_1 - x[j] + x[j+n]$
        si $s_1 > s$
            $s = s_1$

$k = j$
finsi
finpara

Problema 3.6 Siguiendo el esquema divide y vencerás se programa un algoritmo de ordenación:

ordenar(i, d:indices):
si $d - i < 10$
ordenarbasico(i, d)
en otro caso
$m = (d + i)$ *div* 2
ordenar(i, m)
ordenar($m + 1, d$)
mezclar(i, m, d)
finsi

Obtener su orden exacto en el caso en que ordenarbasico sea un método de ordenación por la burbuja o un mergesort. ¿Qué pasaría con el orden exacto en los dos casos si la condición del si fuera $d - i < 100$?

Solución:

El método de la burbuja tiene un orden $\theta\left(n^2\right)$, y el mergesort un $\theta(n \log n)$, por tanto, tendremos que $t(n) = 2t\left(\frac{n}{2}\right) + bn + a$ si $n \geq 10$, y si $n < 10$ será $t(n) \in \theta\left(n^2\right)$ ó $t(n) \in \theta(n \log n)$ dependiendo de que ordenarbasico sea un método de la burbuja o un mergesort, respectivamente.

Expandiendo la recurrencia tendremos:

$$t(n) = 2t\left(\frac{n}{2}\right) + bn + a = 2\left(2t\left(\frac{n}{2^2}\right) + b\frac{n}{2} + a\right) + bn + a =$$
$$2^2 t\left(\frac{n}{2^2}\right) + b2n + a(1+2) = \ldots = 2^k t\left(\frac{n}{2^k}\right) + bkn + a\left(1 + 2 + \ldots + 2^{k-1}\right) =$$
$$2^k t\left(\frac{n}{2^k}\right) + bkn + a\left(2^k - 1\right)$$

cuando $\frac{n}{2^k} < 10$ es $\log \frac{n}{10} < k \Rightarrow \log n - \log 10 < k$, y tomando $\log n - \log 10 = k$ (lo que no va a influir en el orden) tenemos:

$$t(n) = 2^{\log n - \log 10} t(10) + b(\log n - \log 10)n + a\left(\frac{n}{10} - 1\right) =$$
$$\frac{n}{10} t(10) + bn \log n - bn \log 10 + a\left(\frac{n}{10} - 1\right) \in \theta(n \log n)$$

independientemente de que el caso base se resuelva por la burbuja o mergesort, ya que $t(10)$ será una costante en cualquier caso.

Si el caso base se considera de tamaño 100 el resultado sigue siendo el mismo sin más que sustituir en la fórmula anterior el valor 10 por 100. La única diferencia estará en las constantes que aparecen en $t(n)$, por lo que $t(n) \in \theta(n \log n)$.

El estudio lo hemos hecho suponiendo n potencia de dos. Para quitar esta restricción habría que utilizar el teorema 2.1.

Problema 3.7 Supuesto que tenemos un algoritmo que multiplica matrices triangulares ($a_{ij} = 0$ si $i > j$) cuadradas por matrices completas con un coste $\frac{3}{4}n^{2.81}$ (y lo mismo para la multiplicación de una matriz completa por otra triangular), resolver por divide y vencerás el problema de multiplicar matrices cuadradas triangulares utilizando los algoritmos de multiplicación por matrices completas. Calcular el tiempo de ejecución del algoritmo diseñado.

Solución:

Si consideramos las matrices triangulares A y B que queremos multiplicar para obtener la matriz C también triangular, el procedimiento podría ser multiplicar(A, B,d,C), siendo d la dimensión del problema que estamos resolviendo (el número de filas y columnas de las matrices).

Las matrices de dimensión $n \times n$ se pueden descomponer en submatrices de dimensión $\frac{n}{2} \times \frac{n}{2}$ teniendo:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ 0 & A_{22} \end{pmatrix} \quad , B = \begin{pmatrix} B_{11} & B_{12} \\ 0 & B_{22} \end{pmatrix} \quad , C = \begin{pmatrix} C_{11} & C_{12} \\ 0 & C_{22} \end{pmatrix}$$

siendo las matrices de subíndices 11 y 22 triangulares y las de subíndice 12 completas.

La matriz C se obtiene con la fórmula:

$$C = \begin{pmatrix} A_{11}B_{11} & A_{11}B_{12} + A_{12}B_{22} \\ 0 & A_{22}B_{22} \end{pmatrix}$$

con lo que la matriz C se obtiene haciendo dos multiplicaciones de matrices triangulares, y dos multiplicaciones de matrices triangulares por completas y una suma de matrices completas.

El esquema del algoritmo sería:

```
multiplicar(A, B,d,C):
   si pequeño(d)
      mult_completa(A, B,d,C)
   en otro caso
      obtener A11, A12, A22, B11, B12, B22
      multiplicar(A11, B11,d/2,C11)
      mult_triancomp(A11, B12,d/2,D)
      mult_comptrian(A12, B22,d/2,E)
      sumar(D, E,d/2,C12)
      multiplicar(A22, B22,d/2,C22)
   finsi
```

El tiempo de ejecución será:

$t(n) = 2t\left(\frac{n}{2}\right) + \frac{3}{2}n^{2.81} + \frac{1}{4}n^2$

pues se resuelven dos subproblemas del mismo tipo, se hacen dos multiplicaciones matriz triangular por completa y una suma de matrices que tiene coste $\frac{1}{4}n^2$.

Desarrollando tenemos:

$t(n) = 2\left(2t\left(\frac{n}{4}\right) + \frac{3}{2}\left(\frac{n}{2}\right)^{2.81} + \frac{1}{4}\left(\frac{n}{2}\right)^2\right) + \frac{3}{2}n^{2.81} + \frac{1}{4}n^2$

y agrupando:

$t(n) = 2^2t\left(\frac{n}{2^2}\right) + \frac{3}{2}n^{2.81}\left(1 + 2^{-1.81}\right) + \frac{1}{4}n^2\left(1 + 2^{-1}\right)$

A continuación

$t(n) = 2^3t\left(\frac{n}{2^3}\right) + \frac{3}{2}n^{2.81}\left(1 + 2^{-1.81} + \left(2^{-1.81}\right)^2\right) + \frac{1}{4}n^2\left(1 + 2^{-1} + \left(2^{-1}\right)^2\right)$

Y por simplificar consideraremos el tamaño del caso base uno y que n es potencia de dos. Teniendo en cuenta que los términos que afectan a $n^{2.81}$ y n^2 son progresiones geométricas de razón $2^{-1.81}$ y 2^{-1} respectivamente quedará:

$t(n) = n + \frac{3}{2}n^{2.81}\frac{\frac{n^{1.81}-1}{n^{1.81}}}{\frac{2^{1.81}-1}{2}} + \frac{1}{4}n^2\frac{\frac{n-1}{n}}{\frac{1}{2}} = n + \frac{3}{2^{1.81}-1}n^{2.81} - \frac{3}{2^{1.81}-1}n + \frac{1}{2}n^2 - \frac{1}{2}n \in \theta\left(n^{2.81}\right)$

Problema 3.8 Dado el esquema del algoritmo Quicksort:

quicksort(izq,der):
  if $izq < der$
    p=mediana(izq,der)
    particionar(p,izq,der,div)
    quicksort(izq,div)
    quicksort($div+1$,der)
  endif

Donde, con "mediana" se obtiene la mediana de los elementos del array a entre las posiciones izq y der (el elemento que ocuparía la posición central si estuvieran ordenados), y "particionar" es el procedimiento típico de particionar pero usando p (la mediana) como pivote, con lo que el problema se divide en dos subproblemas de igual tamaño. Si el tiempo de ejecución del procedimiento "mediana" es $t_{med}(n) = 20n$, y el de "particionar" es $t_{par}(n) = n$:

a) Calcular el tiempo de ejecución de esta versión del Quicksort.

b) Si el método de la burbuja tiene un tiempo de ejecución n^2, ¿para qué valores de la entrada es preferible esta versión del Quicksort al método de la burbuja?

Solución:

a) La ecuación de recurrencia siendo n potencia de dos sería:

$t(n) = \begin{cases} a & si\ n \leq 1 \\ 21n + b + 2t\left(\frac{n}{2}\right) & si\ n > 1 \end{cases}$

Expandiendo la recurrencia tenemos:

$$t(n) = 21n + b + 2t\left(\frac{n}{2}\right) = 21n + b + 2\left(21\frac{n}{2} + b + 2t\left(\frac{n}{2^2}\right)\right) =$$

$$21n2 + b(1+2) + 2^2t\left(\frac{n}{2^2}\right) = 21n2 + b(1+2) + 2^2\left(21\frac{n}{2^2} + b + 2t\left(\frac{n}{2^3}\right)\right) =$$

$$\ldots = 21nk + b\left(1 + 2 + \ldots + 2^{k-1}\right) + 2^k t\left(\frac{n}{2^k}\right) =$$

$$21nk + b\left(2^k - 1\right) + 2^k a = 21n\log n + b(n-1) + an \in \theta\left(n \log n | n = 2^k\right)$$

Para quitar la condición hay que aplicar el mismo teorema que en el problema anterior. En este caso $f(n) = n\log n$ es eventualmente no decreciente y 2-armónica, y faltaría demostrar que $t(n)$ es eventualmente no decreciente, lo que haremos por inducción:

$t(2) = 42 + b + 2t(1) > t(1)$, con lo que se cumple el caso base.

Si suponemos que es creciente hasta n, tendremos que demostrar que lo es hasta $n+1$ o, lo que es lo mismo, que $t(n) \leq t(n+1)$. Pero $t(n) = 21n+b+t\left(\lceil \frac{n}{2} \rceil\right)+t\left(\lfloor \frac{n}{2} \rfloor\right)$, y $t(n+1) = 21n + 21 + b + t\left(\lceil \frac{n+1}{2} \rceil\right) + t\left(\lfloor \frac{n+1}{2} \rfloor\right)$, y por la hipótesis de inducción es $t\left(\lceil \frac{n+1}{2} \rceil\right) \geq t\left(\lceil \frac{n}{2} \rceil\right)$ y $t\left(\lfloor \frac{n+1}{2} \rfloor\right) \geq t\left(\lfloor \frac{n}{2} \rfloor\right)$, y por tanto $t(n+1) > t(n)$.

b) Se trata de comparar el tiempo del algoritmo por quicksort con el de la burbuja. Será más rápido el quicksort cuando $21n\log n + b(n-1) + an < n^2$. Como no sabemos los valores de a y b supondremos que valen 1, con lo que hay que comparar $21n\log n + 2n - 1$ con n^2. La función $f(n) = n^2 - 21n\log n - 2n + 1$ es creciente para valores suficientemente grandes, y en $n = 128$ es negativa y en $n = 256$ es positiva, por lo tanto a partir de un cierto valor entre 128 y 256 esta versión del quicksort sería más rápida que el método de la burbuja (no calculamos el valor exacto de n a partir del cual es mejor pues para eso necesitaríamos una calculadora).

Problema 3.9 Suponemos el esquema del método divide y vencerás visto en clase:

resolverDV(p, i, d):
  if pequeño(p, i, d)
    resolverbasico$(p, i.d)$
  else
    $m = \frac{i+d}{2}$
    resolverDV$(p, i, m-1)$
    resolverDV(p, m, d)
    combinar(p, i, m, d)
  endif

Si f es la función que representa el coste de ejecución de obtener m, y de combinar los resultados de los subproblemas; y g es el coste de resolver el problema con el método básico; y consideramos el coste de la función pequeño despreciable.

a) ¿Es verdad que si $O(f) < O(g)$ es mejor, en términos de coste del algoritmo, resolver el problema por divide y vencerás que por el método directo (usando el procedimiento resolverbasico)?

b) Suponiendo que el problema es suficientemente pequeño cuando consta de un único elemento, y que los órdenes de f y g son polinómicos, contestar a la pregunta del apartado a).

En caso de que la respuesta sea negativa habrá que dar un contraejenplo, y en caso de que sea afirmativa habrá que demostrarlo.

Solución:

a) Sabemos que en este caso, si el caso base es de tamaño 1, se cumple que $t(n) = 2^k g(1) + \sum_{i=0}^{k-1} 2^i \left(\frac{n}{2^i}\right)$.

Si tomamos $g(n) = ln\ n$ y $f(n) = 1$ tendremos $t(n) \in O\left(2^k\right) = O(n)$, con lo que es peor resolverlo por divide y vencerás que con el método directo que tiene un orden $O(ln\ n)$.

b) Como ya hemos visto en el apartado a, si $O(f) = O(1)$ el tiempo del método divide y vencerás sería $O(t) = O(n)$, que es un orden menor o igual que el de cualquier polinomio de grado mayor o igual a uno.

Si $O(f) = O(n)$, será $t(n) = n + \sum_{i=0}^{k-1} 2^i \frac{n}{2^i} = n + n \log n \in O(n \log n)$, que está incluido en el orden de cualquier polinomio de grado mayor que uno.

Si $O(f) = O\left(n^a\right)$ con a mayor que uno, tendremos:

$$t(n) = n + n^a \sum_{i=0}^{k-1} \frac{1}{2^{i(a-1)}} = n + n^a \left(\frac{2}{n}\right)^{a-1} \frac{n^{a-1}-1}{2^{a-1}-1} \in O\left(n^a\right)$$

que es $< O\left(n^b\right)$, con $b > a$.

Problema 3.10 Se trata de resolver el problema de encontrar el cuadrado de unos más grande en una tabla cuadrada de bits (un array $n \times n$).

a) Programar un método directo para resolver el problema y dar una cota superior (no demasiado mala) de su tiempo de ejecución.

b) Programar un método para resolver el problema por divide y vencerás y dar una cota superior (no demasiado mala) de su tiempo de ejecución.

Solución:

a) Un método directo puede consistir en recorrer toda la tabla por filas y en cada posición donde encontremos un uno obtener el mayor cuadrado de unos con esa posición en la parte superior izquierda. Haremos un esquema del algoritmo donde se devuelve en f y c la fila y columna superior izquierda del cuadrado máximo, y en tma su tamaño.

```
metododirecto(t,n; var f,c,tma):
    tma = 0 /*Tamaño máximo actual*/
    fila = 1 /*Fila que se está evaluando*/
    while fila + tma ≤ n /*Si sobrepasa el tamaño de la tabla no podemos mejorar
la solución actual*/
        columna = 1
        while columna + tma ≤ n
            if t[fila, columna] = 1
                tam=cuadrado(fila,columna)
                if tam > tma
                    f = fila
```

```
                $c = columna$
                $tma = tam$
            endif
        endif
        $columna++$
    endwhile
    $fila++$
endwhile
```

La función cuadrado recorrerá las filas y columnas adyacentes al elemento que se ha encontrado igual a uno hasta encontrar un cero y sin salirse del cuadrado. No incluimos todos los parámetros por simplicidad.

```
cuadrado($fila$,$columna$):
    for $i = 0$ to min($n - fila$,$n - columna$)
        for $j = 0$ to $i$
            if $t[fila + i, columna + j] = 0$
                return $i$
            endif
        endfor
        for $j = 0$ to $i - 1$
            if $t[fila + j, columna + i] = 0$
                return $i$
            endif
        endfor
    endfor
    return $i$
```

Nos conformaremos con obtener una cota superior del tiempo de ejecución. Para esto acotaremos el número de comparaciones que se hacen de elementos de la tabla. El número máximo de comparaciones a hacer al evaluar la función cuadrado desde la posición (i, j) es $v(i, j) = (n - max(i, j) + 1)^2$ (para elementos de la última fila o columna es 1, para los de la penúltima fila o columna es 4, ...), y el número de elementos en la fila y columna i-ésima que no están en filas y columnas posteriores es $2n - 2i + 1$, con lo que el total de comparaciones será $\sum_{i=1}^{n} i^2(2n - 2i + 1) \in O(n^4)$.

b) Para resolver el problema lo descompondremos en cuatro cuadrados de dimensiones la mitad (supondremos que no hay problemas con los tamaños de los problemas que se van generando, lo que ocurre por ejemplo si n es potencia de dos). Las cuatro soluciones devueltas por las llamadas recursivas se comparan para quedarnos con el mayor de los cuatro cuadrados, pero como puede haber cuadrados mayores en las fronteras habrá que estudiar estas. Se estudiará la frontera vertical (columna $\frac{n}{2}$) buscando unos, y por cada uno en esta columna nos movemos a la izquierda hasta encontrar un cero o salirnos del cuadrado, y a partir de esta columna analizamos los cuadrados hasta la columna $\frac{n}{2}$. De manera similar habrá que trabajar con la frontera horizontal.

En el siguiente esquema se programa el método según las ideas anteriores, pero no se entra en detalles de implementación, donde habría que utilizar una variable leading dimension para el acceso a las filas y columnas, o habría que copiar los datos de las submatrices en otras posiciones para trabajar con ellos.

dívideyvenceras(t,n; var f,c,tma):
    if $n \leq base$
        metododirecto(t,n,f,c,tma)
    else
        divideyvenceras(&$t[1,1]$,$\frac{n}{2}$,f,c,tma)
        divideyvenceras(&$t[1,\frac{n}{2}+1]$,$\frac{n}{2}$,$f2$,$c2$,$tma2$)
        divideyvenceras(&$t[\frac{n}{2}+1,1]$,$\frac{n}{2}$,$f3$,$c3$,$tma3$)
        divideyvenceras(&$t[\frac{n}{2}+1,\frac{n}{2}+1]$,$\frac{n}{2}$,$f4$,$c4$,$tma4$)
        if $tma2 > tma$
            $tma = tma2$
            $f = f2$
            $c = c2$
        endif
        if $tma3 > tma$
            $tma = tma3$
            $f = f3$
            $c = c3$
        endif
        if $tma4 > tma$
            $tma = tma4$
            $f = f4$
            $c = c4$
        endif
        fronteras(t,n,f,c,tma)
    endif

y en el procedimiento fronteras se tratan estas tal como hemos indicado:

fronteras(t,n; var f,c,tma):
    $fila = 1$
    while $fila + tma \leq n$
        $columna = \frac{n}{2}$
        while $columna \geq 1$ and $t[fila, columna] = 1$
            $columna - -$
        endwhile
        $columna + +$
        while $columna + tma \leq n$ and $columna \leq \frac{n}{2}$
            $tam = cuadrado$(fila,columna)
            if $tam > tma$

```
                f = fila
                c = columna
                tma = tam
            endif
            columna++
        endwhile
        fila++
    endwhile
    columna = 1
    while columna + tma ≤ n
        fila = n/2
        while fila ≥ 1 and t[fila, columna] = 1
            fila--
        endwhile
        fila++
        while fila + tma ≤ n and fila ≤ n/2
            tam = cuadrado(fila,columna)
            if tam > tma
                f = fila
                c = columna
                tma = tam
            endif
            fila++
        endwhile
        columna++
    endwhile
```

Para obtener una cota del tiempo de ejecución tenemos la ecuación de recurrencia $t(n) = 4t\left(\frac{n}{2}\right) + t_c(n)$, donde t_c es el tiempo de combinación de los resultados (la función fronteras). Como en la combinación a lo sumo se evalúan cuadrados a partir de todas las casillas de la parte superior izquierda del array (la primera de las cuatro partes en que se ha dividido) una cota superior de ese tiempo será: $\sum_{i=\frac{n}{2}}^{n} i^2(2n-2i+1) \in O\left(n^4\right)$. Por lo tanto, la ecuación de recurrencia es de la forma $t(n) = 4t\left(\frac{n}{2}\right) + O\left(n^4\right)$, cuya solución nos da un $O\left(n^4\right)$. Como la cota superior coincide con la obtenida para el método directo, con el estudio que hemos hecho no podemos decidir qué método puede ser mejor en la práctica.

Problema 3.11 Dado el problema de obtener el tercer mayor elemento de un array con elementos de un cierto "tipo":
a) Hacer un programa para resolverlo directamente (sin divide y vencerás).
b) Calcular Ω y O del número de comparaciones de elementos del tipo "tipo" en el algoritmo del apartado a).

c) Calcular Ω y O del número de asignaciones de elementos del tipo "tipo" en el algoritmo del apartado a).
d) Calcular θ y o del número promedio de comparaciones de elementos del tipo "tipo" en el algoritmo del apartado a).
e) Calcular θ y o del número promedio de asignaciones de elementos del tipo "tipo" en el algoritmo del apartado a).
f) Hacer un programa para resolverlo por divide y vencerás.
g) Calcular θ y o del número promedio de comparaciones de elementos del tipo "tipo" en el algoritmo del apartado f).
h) Calcular θ y o del número promedio de asignaciones de elementos del tipo "tipo" en el algoritmo del apartado f).
Solución:
a) Usaremos tres variables *primero*, *segundo* y *tercero* para almacenar el primer, segundo y tercer dato, respectivamente. Inicialmente les asignaremos un valor especial menor que cualquier otro ($-\infty$). Consideraremos que los elementos del tipo "tipo" se comparan y se asignan usando los signos normales de comparación y asignación. El programa consistirá en recorrer los n datos del array determinando si son el primero, segundo o tercero de los que se llevan recorrido, y en este caso se mueven estos valores para poner el nuevo dato en su sitio. Dado que lo más probable será que un nuevo elemento no sea ninguno de los tres primeros, primero se comparará con *tercero*, y sólo en caso de ser mayor se obtiene la posición que ocupa.

```
primero = -∞
segundo = -∞
tercero = -∞
for i = 1 to n
   if a[i] > tercero
      if a[i] > primero
         tercero = segundo
         segundo = primero
         primero = a[i]
      elsif a[i] > segundo
         tercero = segundo
         segundo = a[i]
      else
         tercero = a[i]
      endif
   endif
endfor
```

b) Sólo se hacen comparaciones dentro del bucle, por lo que $c(n) = \sum_{i=1}^{n} c_i$, con c_i el número de comparaciones que se hacen dentro del for al pasar con el índice i. Como el número de comparaciones en cada paso está entre 1 ($a[i] \leq tercero$) y 3 ($a[i]$ segundo o

tercero), tendremos como cota inferior $\sum_{i=1}^{n} 1 = n$, y como cota superior $\sum_{i=1}^{n} 3 = 3n$, por lo que $c(n) \in \Omega(n)$ y $c(n) \in O(n)$.
c) Asignaciones se hacen tres fuera del bucle, y un número variable dentro, por lo que $a(n) = 3 + \sum_{i=1}^{n} a_i$, con a_i el número de asignaciones que se hacen dentro del for al pasar con el índice i. Como el número de asignaciones en cada paso está entre 0 ($a[i] \leq tercero$) y 3 ($a[i]$ primero), tendremos como cota inferior $3 + \sum_{i=1}^{n} 0 = 3$ (en realidad habrá más asignaciones pues en los tres primeros pasos siempre hay que actualizar alguna de las variables), y como cota superior $3 + \sum_{i=1}^{n} 3 = 3n$, por lo que $a(n) \in \Omega(1)$ y $a(n) \in O(n)$.
d) Dado que en este caso coinciden Ω y O, conocemos también el orden exacto que es $\theta(n)$.

Para calcular o hay que obtener cuál es la constante que multiplica a n en la fórmula. Cuando vamos por el valor i comprobamos si el nuevo elemento es el primero de los i primeros datos del array, si es el segundo mayor, el tercero mayor, o si es menor que el tercero mayor. El número de comparaciones si no es ninguno de los tres mayores es 1, si es el tercero o el segundo es 3, y si es el primero 2. La probabilidad de que sea el primero es $\frac{1}{i}$, de que sea el segundo $\frac{1}{i}$, de que sea el tercero $\frac{1}{i}$, y de que no sea ninguno de los tres es $1 - \frac{3}{i}$. El número promedio de comparaciones será $\sum_{i=1}^{n} \left(2\frac{1}{i} + 3\frac{1}{i} + 3\frac{1}{i} + 1\left(1 - \frac{3}{i}\right)\right) = n + 5\sum_{i=1}^{n} \frac{1}{i}$. Dado que el sumatorio se aproxima por un logaritmo neperiano el término de mayor orden es n, y va afectado por la constante uno, por lo que $c_p(n) \in o(n)$.
e) En este caso no coinciden Ω y O, por lo que no conocemos θ.

Se hacen las tres asignaciones externas y, en el bucle, cuando vamos por el valor i, si es el primero de los i primeros datos del array se hacen tres asignaciones, si es el segundo dos, si es el tercero una, y en otro caso ninguna. El número de asignaciones promedio de asignaciones será $3 + \sum_{i=1}^{n} \left(3\frac{1}{i} + 2\frac{1}{i} + 1\frac{1}{i} + 0\left(1 - \frac{3}{i}\right)\right) = 3 + 6\sum_{i=1}^{n} \frac{1}{i}$. Como ese sumatorio se aproxima por un logaritmo neperiano el término de mayor orden es $6ln\ n$, por lo que $a_p(n) \in \theta(ln\ n)$, y $a_p(n) \in o(6ln\ n)$.
f) La función recibirá el array donde están los datos, los índices izquierdo y derecho que indican dónde se trabaja en cada llamada, y devuelve como variables los tres valores mayores. El caso base será tener un único elemento. El problema se divide en dos subproblemas por la mitad. La combinación será obtener los tres primeros a partir de los tres primeros de las dos partes en que se ha dividido el problema. El esquema será:

DivVen(a:array of tipo;izq,der:índices;var pri,seg,ter:tipo):
if $izq = der$
  $pri = a[izq]$
  $seg = -\infty$
  $ter = -\infty$
else
  $m = \frac{izq+der}{2}$
  DivVen(a,izq,m,$p1$,$s1$,$t1$)

```
        DivVen(a,m + 1,der,p2,s2,t2)
        combinar(p1,s1,t1,p2,s2,t2,pri,seg,ter)
    endif
```

donde en combinar los seis primeros parámetros son de entrada y los tres últimos de salida:

```
    combinar(p1,s1,t1,p2,s2,t2,pri,seg,ter):
    if p1 > p2
        pri = p1
        if s1 > p2
            seg = s1
            if t1 > p2
                ter = t1
            else
                ter = p2
            endif
        else
            seg = p2
            if s1 > s2
                ter = s1
            else
                ter = s2
            endif
        endif
    else
        pri = p2
        if s2 > p1
            seg = s2
            if t2 > p1
                ter = t2
            else
                ter = p2
            endif
        else
            seg = p1
            if s1 > s2
                ter = s1
            else
                ter = s2
            endif
        endif
    endif
```

La combinación se podría hacer de otras maneras: con una mezcla de seis elementos hasta obtener los tres mayores, con el método directo del apartado a), ...
g) Para calcular el número de comparaciones hay que plantear una ecuación de recurrencia. El caso base es $n = 1$, y no se hacen comparaciones; y en el caso general $c(n) = 2c\left(\frac{n}{2}\right) + 3$, siendo 3 el número de comparaciones que se hacen al combinar. Haciendo el cambio $n = 2^k$ queda $c_k - 2c_{k-1} = 3$, con lo que la ecuación carcterística es $(x-2)(x-1) = 0$, y la fórmula general del número de comparaciones es $c_k = a_1 2^k + a_2 1^k$, con lo que $c(n) = a_1 n + a_2$. Para calcular las constantes a_1 y a_2 se plantea un sistema de dos ecuaciones con dos incógnitas imponiendo los casos base 1 y 2:

$$\begin{array}{rcccl} c(1) & = & 0 & = & a_1 + a_2 \\ c(2) & = & 3 & = & 2a_1 + a_2 \end{array}$$

Resolviéndolo se tiene $c(n) = 3n - 3$, por lo que $c(n) \in \theta(n)$ y $c(n) \in o(3n)$.
h) La ecuación de recurrencia es en este caso similar a la anterior. El caso base es $n = 1$, y se hacen 3 asignaciones; y en el caso general $a(n) = 2a\left(\frac{n}{2}\right) + 3$, siendo 3 el número de asignaciones que se hacen al combinar. La fórmula general del número de asignaciones es $a(n) = c_1 n + c_2$. Para calcular las constantes c_1 y c_2 se plantea un sistema de dos ecuaciones con dos incógnitas imponiendo los casos base 1 y 2:

$$\begin{array}{rcccl} a(1) & = & 3 & = & c_1 + c_2 \\ a(2) & = & 9 & = & 2c_1 + c_2 \end{array}$$

Resolviéndolo se tiene $a(n) = 6n - 3$, por lo que $a(n) \in \theta(n)$ y $a(n) \in o(6n)$.

Problema 3.12 Programar por Divide y Vencerás, con un esquema recursivo similar al visto en clase, la resolución del problema de encontrar la subsecuencia más larga de caracteres iguales en una secuencia de caracteres. Analizar el tiempo de ejecución. ¿Es conveniente resolver este problema por Divide y Vencerás o es preferible resolverlo directamente? (justificar la respuesta).

Solución:

El esquema recursivo visto en clase es:

```
divide_venceras(datos:array[1,...,n] of tipo;p,q:índices):
   m:índice
begin
   if pequeño(p,q)
      solucion(datos,p,q)
   else
      m=dividir(p,q)
      divide_venceras(datos,p,m)
      divide_venceras(datos,m+1,q)
      combinar(datos,p,m,q)
   endif
end
```

Una primera versión de algoritmo resolviendo nuestro problema según este esquema puede ser:

- El problema será suficientemente pequeño cuando sólo tengamos un elemento.

  pequeño(p,q):

  return $p = q$

- La cadena de elementos repetidos más larga en el caso de tener un elemento está formada por ese único elemento. Para devolver la solución del problema necesitamos devolver la posición donde empieza la cadena y el número de elementos que tiene, con lo que las funciones solucion, divide_venceras y combinar tendrán dos parámetros variables más: *inicio*:índice, y *longitud*:integer.

  solucion($datos$,p,q,$inicio$,$longitud$):

  $inicio = p$

  $longitud = 1$

- Cuando no estamos en el caso base el problema se dividirá por la mitad.

  dividir(p,q):

  return $\frac{p+q}{2}$

- Las llamadas a divide_venceras serán:

  divide_venceras($datos$,p,m,$i1$,$l1$)

  divide_venceras($datos$,$m + 1$,q,$i2$,$l2$)

  de manera que con $i1$, $l1$ y $i2$, $l2$, tengamos localizadas las subcadenas máximas entre las posiciones p y m, y $m + 1$ y q, respectivamente. Supondremos también que la primera llamada se ha hecho con:

  divide_venceras($datos$,1,n,i,l)

- La función más complicada es la de combinar, pues la subcadena máxima puede haber quedado cortada entre las dos subcadenas, y esto hay que comprobarlo. Además, debe recibir las soluciones parciales de las dos subcadenas ($i1$, $l1$, $i2$ y $l2$) y devolver la solución global (i, l), que será lo que devuelva divide_venceras.

  combinar($datos$,p,m,q,$i1$,$l1$,$i2$,$l2$,i,l):

  $l = 0$

  if $datos[m] = datos[m + 1]$

  /*obtener en i la posición de inicio de la cadena cortada, y en l la longitud de la cadena*/

  $l = 2$

  $i = m$

```
    while $i-1 \geq p$ and $datos[i] = datos[i-1]$
        $l = l+1$
        $i = i-1$
    endwhile
    $j = m+1$
    while $j+1 \leq q$ and $datos[j] = datos[j+1]$
        $l = l+1$
        $j = j+1$
    endwhile
endif
/*comparar las longitudes de las tres subcadenas máximas: la de las subcadenas
izquierda y derecha y la cadena cortada. Si no hay cadena cortada $l = 0$*/
if $l1 > l$
    $l = l1$
    $i = i1$
endif
if $l2 > l$
    $l = l2$
    $i = i2$
endif
```

En el caso más desfavorable la función combinar tendrá un tiempo de orden n pues tendría que recorrer todo el array para comprobar que la cadena que ha quedado cortada llega desde el principio hasta el final del subarray que está tratando (el caso más desfavorable será, por tanto, cuando todos los caracteres de la cadena total sean iguales). De este modo la ecuación de recurrencia es:

$$t(n) = \begin{cases} a & si\ n = 1 \\ 2t\left(\frac{n}{2}\right) + n & si\ n > 1 \end{cases}$$

Y expandiendo la recurrencia tenemos:

$$t(n) = n + 2t\left(\frac{n}{2}\right) = n + 2\left(\frac{n}{2} + 2t\left(\frac{n}{4}\right)\right) = \ldots = kn + 2^k t\left(\frac{n}{2^k}\right)$$

y en el caso base es $k = \log n$, y por tanto $t(n) = n \log n + an \in O(n \log n)$.

Un método directo resuelve el problema con un orden $\theta(n)$ pues basta con recorrer la cadena de principio a final llevando una solución óptima actual y actualizándola cada vez que se encuentre una subcadena más larga. Por tanto, sería preferible un método directo.

Pero no podemos concluir que un método directo sea mejor que un divide y vencerás para resolver este problema. El orden $n \log n$ en el divide y vencerás nos aparece porque hacemos trabajo repetido en la función de combinación, pero podemos evitar esta repetición de trabajos. Se pueden combinar las funciones pequeño, solucion y dividir en una sola. Se accederá al elemento central de la cadena y se obtendrá la subcadena más larga de caracteres iguales que contiene a ese elemento, y si su longitud coincide con la del subarray que estamos tratando el problema es suficientemente pequeño y la solución que hemos encontrado es la solución de la subcadena que estamos tratando. Si el problema no es suficientemente pequeño se devuelven los índices inicial y final de la subcadena máxima encontrada para dividir el problema usando esos índices y no volver a tratar los elementos de la subcadena máxima encontrada. En todo momento llevaremos una subcadena máxima actual indicada por unos valores globales ig y lg, de manera que la función combinar compare la longitud de las subcadenas máximas encontradas con la de la máxima global actual, y actualizará esta si es necesario. Las funciones pueden ser:

- pequeño es parte de la función combinar anterior:

  pequeño($datos$,p,q,i,l):

  /*i y l son variables donde se devuelve la cadena más larga*/

  $l = 0$

  $m = p + \frac{q-p}{2}$

  if $datos[m] = datos[m+1]$

      $l = 2$

      $i = m$

      while $i - 1 \geq p$ and $datos[i] = datos[i-1]$

          $l = l + 1$

          $i = i - 1$

      endwhile

      $j = m + 1$

      while $j + 1 \leq q$ and $datos[j] = datos[j+1]$

          $l = l + 1$

          $j = j + 1$

endwhile
endif
/*la subcadena obtenida se compara con la óptima actual*/
if $l > lg$
$lg = l$
$ig = i$
endif
/*si la subcadena es toda de caracteres idénticos estamos en el caso base*/
return $i = p$ and $i + l - 1 = q$

- No se necesita función solucion pues la solución en el caso base se ha obtenido en la función anterior.
- Con los valores devueltos por la función pequeño se divide el problema, por lo que no se necesita función dividir.
- Las llamadas a divide_venceras son:

$i1 = 0$
$l1 = 0$
if $i \geq p$
divide_venceras($datos$,p,i,$i1$,$l1$)
endif
$i2 = 0$
$l2 = 0$
if $i + l - 1 \leq q$
divide_venceras($datos$,$i + l - 1$,q,$i2$,$l2$)
endif

- La función combinar consiste en comparar las subcadenas máximas de los subproblemas izquierdo y derecho con la máxima global.

combinar($datos$,p,m,q,$i1$,$l1$,$i2$,$l2$):
if $l1 > lg$
$lg = l1$
$ig = i1$
endif

if $l2 > lg$
  $lg = l2$
  $ig = i2$
endif

Con esta implementación el orden del algoritmo por divide y vencerás será $\theta(n)$ pues sólo se trata una vez cada uno de los elementos de *datos* (lo que se hace en la función pequeño). Pero con el esquema de divide y vencerás se hacen llamadas recursivas y es necesario utilizar más variables auxiliares y más trabajo sobre ellas que en el método directo, por lo que sigue siendo preferible el método directo.

Pero esto tampoco significa que sea preferible un método directo a uno divide y vencerás. El método por divide y vencerás se puede mejorar haciendo unas pequeñas modificaciones. Las llamadas a divide_venceras sólo es necesario hacerlas cuando la longitud de las cadenas a tratar sea mayor que la de la solución óptima actual:

$i1 = 0$
$l1 = 0$
if $i \geq p$ and $i - p + 1 \geq lg$
  divide_venceras($datos$,p,i,$i1$,$l1$)
endif
$i2 = 0$
$l2 = 0$
if $i + l - 1 \leq q$ and $q - i - l + 2 \geq lg$
  divide_venceras($datos$,$i + l - 1$,q,$i2$,$l2$)
endif

De este modo, el orden en el caso más desfavorable sigue siendo $\theta(n)$, pero es posible que muchos elementos no tengamos que tratarlos si las subcadenas en que se encuentran cuando vamos a tratarlos son de longitud menor que la cadena de longitud máxima actual. Esto también se puede hacer en el método directo, pero en ese caso sólo evitamos tratar algunos elementos del final, mientras que con el divide y vencerás evitamos tratar elementos de toda la cadena. Qué método es preferible puede depender de los datos que tengamos, por lo que quizás hay que estudiarlo experimentalmente.

Problema 3.13 Se tiene un algoritmo para ordenar datos en un array a por un método divide y vencerás dividiendo cada problema en tres subproblemas, según un esquema del tipo:

D_V(a:array de datos;i,j:indices)
if $i < j$
  $m1 = (j - i)\ div\ 3 + i$
  $m2 = ((j - i)\ div\ 3) * 2 + i$
  D_V(a,i,$m1$)
  D_V(a,$m1 + 1$,$m2$)

D_V(a,$m2+1$,j)
combinar(a,i,$m1$,$m2$,j)
endif

donde los índices i y j indican la zona del array a donde se trabaja en cada llamada.

Programar la función "combinar", estudiar el tiempo de ejecución de este método de ordenación y compararlo con el de la ordenación por mezcla normal donde se divide cada problema en dos subproblemas.

Solución:

Como vamos a tener que comparar el tiempo de ejecución del algoritmo que implementemos con el de ordenación por mezcla normal en el que cada problema se divide en dos subproblemas, empezamos recordando el tiempo de ejecución de la ordenación por mezcla. La recurrencia es:

$$t(n) = \begin{cases} a & si \quad n = 1 \\ 2t\left(\frac{n}{2}\right) + f(n) & si \quad n > 1 \end{cases}$$

donde $f(n)$ representa el tiempo de dividir el problema en subproblemas y de combinar los resultados de los subproblemas. El coste de la mezcla sabemos que es lineal, por lo que $f(n) = bn+c$, tanto en el caso más favorable como en el más desfavorable. Distintas implementaciones dan lugar a distintos valores de b y c. Dado que el valor de c no influye en el orden del algoritmo consideraremos $f(n) = bn$. En este caso, la ecuación de recurrencia queda, suponiendo $n = 2^k$, como $t_k = 2t_{k-1} + b2^k$, y $t_k = c_1 2^k + c_2 k 2^k$, y $t(n) = c_1 n + c_2 n log_2\, n$. Planteando el sistema:

$$\begin{array}{ccccc} t(1) & = & a & = & c_1 \\ t(2) & = & 2a + 2b & = & 2c_1 + 2c_2 \end{array}$$

obtenemos $c_1 = a$ y $c_2 = b$, y $t(n) \in o\,(bnlog_2 n)$, condicionado a que n sea potencia de dos, pero sabemos que para la ordenación por mezcla esa restricción se puede quitar.

Del mismo modo estudiamos el método de ordenación que se nos propone dividiendo cada problema en tres subproblemas. La recurrencia es:

$$t(n) = \begin{cases} a & si \quad n = 1 \\ 3t\left(\frac{n}{3}\right) + f(n) & si \quad n > 1 \end{cases}$$

donde $f(n)$ representa el tiempo de dividir el problema en subproblemas y de combinar los resultados de los subproblemas. La combinación en este caso se puede hacer de distintas maneras, y en el problema se nos dice que programemos una de ellas. Se puede hacer una combinación de coste lineal ya que, por ejemplo, se podrían mezclar los dos primeros trozos, con un orden lineal en función del número de datos a mezclar, que es $\frac{2n}{3}$, y una segunda mezcla del primer trozo de $\frac{2n}{3}$ datos ordenados con los últimos $\frac{n}{3}$ datos también ordenados. El coste de la combinación será lineal, por lo que $f(n) = bn + c$, tanto en el caso más favorable como en el más desfavorable. Distintas implementaciones dan lugar a distintos valores de b y c. Dado que el valor de c no influye en el orden del algoritmo consideraremos $f(n) = bn$. En este caso, la ecuación de recurrencia queda, suponiendo $n = 3^k$, como $t_k = 3t_{k-1} + b3^k$, y $t_k = c_1 3^k + c_2 k 3^k$, y $t(n) = c_1 n + c_2 n log_3\ n$. Planteando el sistema:

$$\begin{array}{rcccc} t(1) & = & a & = & c_1 \\ t(3) & = & 3a + 2b & = & 3c_1 + 3c_2 \end{array}$$

obtenemos $c_1 = a$ y $c_2 = b$, y $t(n) \in o\,(bnlog_3 n)$, condicionado a que n sea potencia de tres. La restricción se puede quitar al igual que en la ordenación por mezcla normal pues $nlog_3 n$ es creciente en los positivos y es 3-armónica. Y t es creciente, lo que se demuestra por inducción pues $t(2) = 2t(1) + b2 > t(1) = a$, y si es creciente hasta n se puede demostrar que $t(n+1) \geq t(n)$ pues el problema de tamaño $n+1$ se resuelve con llamadas a tres subproblemas, dos de ellos del mismo tamaño que los tamaños resuelto para n y otro de tamaño un dato más.

Para poder comparar la ordenación por mezcla normal y la del presente problema hay que determinar el valor de la constante b en cada caso. Para la ordenación por mezcla se puede considerar que se utiliza un array auxuliar para la mezcla o que no se utiliza dicho array, con lo que se evita la copia final del array auxiliar al original. Consideraremos la versión en que no se utiliza array auxiliar. En este caso el número de asignaciones en una mezcla de n elementos es n, y el número de comparaciones está entre $\frac{n}{2}$ y $n-1$, dado que normalmente estaremos más cerca del caso más desfavorable que del más favorable, consideraremos una buena aproximación al coste de la mezcla como $f(n) = (a + c)n$, siendo a el coste de una asignación y c el de una comparación. Por tanto el tiempo de ejecución aproximado es $(a + c)nlog_2\ n$.

Si la combinación que se nos propone la hacemos realizando una mezcla normal del primer tercio de los n datos con el segundo tercio, con coste aproximado $\frac{2(a+c)}{3}n$, dejando los datos en un array auxiliar, y después se mezclan los $\frac{2n}{3}$ del array auxiliar con los del tercer tercio del array original dejando el resultado en el array original, el coste de esta segunda mezcla es aproximadamente $(a + c)n$, con lo que el coste total de la combinación es aproximadamente $\frac{5(a+c)}{3}n$, y el tiempo de ejecución aproximado del algoritmo de ordenación es $\frac{5(a+c)}{3}nlog_3\ n$.

El método de ordenación por mezcla dividiendo en tres subproblemas sería mejor que el método normal si $\frac{5(a+c)}{3}nlog_3\ n < (a+c)nlog_2\ n$, lo que ocurre si $5log_3\ 2 < 3$, lo que no es verdad, por lo que el método normal de ordenación por mezcla es mejor que el método dividiendo en tres subproblemas con la combinación propuesta.

Podemos hacer la combinación de otra manera con un esquema similar al de la mezcla de dos subarrays ordenados. Utilizaremos tres índices i, j y k para indicar por dónde vamos comparando en los tres subarrays, y un índice l para indicar dónde vamos escribiendo en un array auxiliar. Consideraremos que no es necesario copiar del array auxiliar al original, tal como estamos suponiendo en la ordenación por mezcla normal. Para que esto sea así habría que modificar el esquema que se nos da, igual que se modifica el esquema de la ordenación por mezcla normal para evitar la copia.

combinar(a,$i1$,$i2$,$i3$,$i4$):
    $i = i1$
    $j = i2 + 1$
    $k = i3 + 1$
    $l = i1$
    while $i \leq i2$ and $j \leq i3$ and $k \leq i4$
        if $a[i] < a[j]$
            if $a[i] < a[k]$
                $b[l] = a[i]$
                $i = i + 1$
            else
                $b[l] = a[k]$
                $k = k + 1$
        else
            if $a[j] < a[k]$
                $b[l] = a[j]$
                $j = j + 1$
            else
                $b[l] = a[k]$
                $k = k + 1$
        endif
        $l = l + 1$
    endwhile
    if $i > i2$
        mezclar(a,j,$i3$,k,$i4$,b,l)
    elsif $j > i3$
        mezclar(a,i,$i2$,k,$i4$,b,l)
    else
        mezclar(a,i,$i2$,j,$i3$,b,l)
    endif

donde lo que se hace es obtener el menor de tres datos, uno de cada subarray, haciendo dos comparaciones, mientras en ninguno de los subarrays se ha llegado al final. Cuando en uno de los subarrays se ha llegado al final se trabaja con los datos restantes de los dos subarrays con los que no se ha acabado y se hace una mezcla normal de a sobre el array auxiliar que estamos suponiendo.

El código de la mezcla sería:

mezclar(a,$i1$,$f1$,$i2$,$f2$,b,ib):
    while $i1 \leq f1$ and $i2 \leq f2$
        if $a[i1] < a[i2]$
            $b[ib] = a[i1]$

```
            i1 = i1 + 1
        else
            b[ib] = a[i2]
            i2 = i2 + 1
        endif
        ib = ib + 1
    endwhile
    if i1 > f1
        for i = i2 to f2
            b[ib] = a[i]
            ib = ib + 1
        endfor
    else
        for i = i1 to f1
            b[ib] = a[i]
            ib = ib + 1
        endfor
    endif
```

Para estudiar el coste hay que tener en cuenta que siempre se copian n datos, y que si la copia se hace obteniendo el menor de tres elementos este menor se obtiene con dos comparaciones, si se hace obteniendo el menor de dos con una comparación, y si se hace compiando el final de un array cuando ya se han acabado los otros dos se hace sin comparaciones. En el caso más desfavorable cada elemento (menos los dos últimos, lo que no influye en el orden) se copia realizando dos comparaciones. Por tanto, podemos considerar como aproximación del coste de la combinación $(a+2c)n$, y el coste de la ordenación es aproximadamente $(a+2c)nlog_3\ n$.

La ordenación dividiendo en tres subproblemas será mejor que la normal dividiendo en dos si $(a+2c)nlog_3\ n < (a+c)nlog_2\ n$ (sólo comparamos operaciones sobre elementos del array a ordenar), lo que ocurre cuando $(a+2c)log_3\ 2 < a+c$, que ocurre aproximadamente cuando el coste de una asignación es mayor que 0.7 veces el coste de una comparación.

Problema 3.14 Consideramos el problema de obtener de n números los $\frac{n}{10}$ menores ordenados.
a) Hacer un programa según el esquema divide y vencerás para resolver el problema.
b) Estudiar el coste de dicho programa.
c) Estudiar el coste si en vez de obtener los $\frac{n}{10}$ menores ordenados lo que se pretende es obtener los 100 menores ordenados.

Solución:
a) La idea puede consistir en hacer una ordenación por mezcla tomando como casos base los menores o iguales que $\frac{n}{10}$ y quedándonos en las mezclas con los $\frac{n}{10}$ primeros. Suponemos que los datos están en un array global a y que la dimensión del array (n)

también es un dato global.

ordenar_primeros(izq,der:índices):
si $der - izq < \frac{n}{10}$
quicksort(izq,der)
en otro caso
$m = \frac{izq+der}{2}$
ordenar_primeros(izq,m)
ordenar_primeros($m+1$,der)
mezclar_primeros(izq,$m+1$,der,$\frac{n}{10}$)
finsi

Vemos que en el caso base utilizamos el método quicksort pues es el más rápido en promedio. Si no estamos en el caso base dividimos el array en dos de igual tamaño y hacemos llamadas recursivas. La única diferencia con la ordenación por mezcla normal estará a la hora de realizar la mezcla, donde nos quedaremos siempre con un número de datos menor o igual a $\frac{n}{10}$, lo que viene indicado por el cuarto parámetro. mezcla_primeros podría ser:

mezcla_primeros(izq,cen,der,tot):
$i = izq$
$j = cen$
$k = 1$
mientras $i < cen$ y $j \leq der$ y $k \leq tot$
si $a[i] < a[j]$
$b[k] = a[i]$
$i++$
en otro caso
$b[k] = a[j]$
$j++$
finsi
$k++$
finmientras
si $k < tot$
si $i < cen$
mientras $i < cen$ y $k \leq tot$
$b[k] = a[i]$
$k++$
$i++$
finmientras
en otro caso
mientras $j \leq der$ y $k \leq tot$
$b[k] = a[j]$
$k++$

$j++$
finmientras
finsi
finsi
$i = izq$
$k = 1$
mientras $k \leq tot$ y $i \leq der$
$a[i] = b[k]$
$i++$
$k++$
finmientras

Observamos que se utiliza un array auxiliar b, lo que se podría evitar si queremos reducir el tiempo de ejecución; se va copiando el menor de los elementos que se comparan en el array b, pero sólo hasta completar la totalidad de los elementos que queremos obtener ordenados (en este caso $\frac{n}{10}$); y finalmente se copian esos elementos de b en la posición correspondiente de a.

b) El tiempo de ejecución se obtiene según la fórmula $t(n) = 2t\left(\frac{n}{2}\right) + \frac{n}{10}$, ya que en las mezclas se toman siempre los $\frac{n}{10}$ menores datos (si tuvieramos menos se tomarían menos, pero como el caso base es $\frac{n}{10}$ siempre vamos a tener esa cantidad de datos). Expandiendo la recurrencia tendremos $t(n) = 2^k t\left(\frac{n}{2^k}\right) + \frac{n}{10}\left(2^k - 1\right)$, y como el caso base es cuando $\frac{n}{2^k} = \frac{n}{10}$, tenemos $k = \log 10$, y el tiempo será $n \log n - n \log 10 + \frac{9}{10}n$.

c) En el caso de querer obtener los 100 menores ordenados el esquema sería similar, pero el caso base tendría tamaño 100 y el valor del parámetro tot en la llamada a las mezclas sería 100. La ecuación quedaría $t(n) = 2t\left(\frac{n}{2}\right) + 100$, y el caso base sería $\frac{n}{2^k} = 100$, con lo que $k = \log n - \log 100$, y el tiempo expandiendo la recurrencia queda $t(n) = 2^k t\left(\frac{n}{2^k}\right) + 100\left(2^k - 1\right)$ y sustituyendo el caso base $t(n) = \frac{n}{100}t(100) + 100\left(\frac{n}{100} - 1\right) = n \log 100 + n - 99$, que es un tiempo lineal, a diferencia del caso anterior, donde era del orden $O(n \log n)$.

Problema 3.15 a) Consideramos la ordenación por mezcla recursiva con datos en un array a donde en las llamadas a la mezcla se utiliza un array auxiliar b al que se asigna memoria en cada llamada y que se libera al salir de la mezcla:

mezcla(tipo a,int izq,int med,int der)
tipo *b
b=(tipo *) malloc(sizeof(tipo)*DATOS)
...
free(b)

Estudiar la ocupación de memoria de la ordenación por mezcla en el caso en que el número de DATOS que se reserva cada vez sea n y en el caso en que sea el número de datos que se están mezclando (DATOS=der-izq+1).

b) Calcular θ del número promedio de comparaciones que se hacen en la ordenación por mezcla dentro de la función de mezcla.

Solución:

a) El esquema de la ordenación por mezcla es:

mergesort(a,izq,der):
 if caso_base(izq,der)
  ordenar_directo(a,izq,der)
 else
  $m = \frac{izq+der}{2}$
  mergesort(a,izq,m)
  mergesort(a,$m+1$,der)
  mezcla(a,izq,m,der)
 endif

Supondremos que el array a donde están los datos a ordenar contiene n elementos y que en las llamadas recursivas lo que se pasa es la dirección de memoria de a (tal como se hace en C), con lo que la ocupación de cada llamada recursiva es constante correspondiente a los índices izq, der, a la dirección de a y a la variable local m. Como el número máximo de llamadas recursivas ejecutándose al mismo tiempo es $\log_2 n$ si el caso base es de un elemento, la ocupación de memoria por las llamadas recursivas a mergesort es como mucho $k \log_2 n$, con k la constante correspondiente a la ocupación de m, izq, der y la dirección de a.

En el caso en que en la mezcla la cantidad de memoria que se reserva para b sea siempre n, cuando se llega al caso base y se vuelve de la llamada recursiva a ese caso se tendrá una ocupación de $2n + k \log_2 n - k$, por la ocupación de a, b y los parámetros de la recursión. Este es el momento de máxima ocupación de memoria porque a partir de ese punto se libera la memoria de b, que se puede volver a usar en otra llamada a la mezcla, y se sube un nivel en la recursión. El orden de la ocupación de memoria es por tanto $o(2n)$.

En el caso de reservar para b la memoria necesaria para trabajar con los elementos que se están mezclando $(der - izq + 1)$, en la llamada más profunda de la recursión en la que se hace la mezcla (mezclar dos datos si el caso base es uno) la ocupación es $n + k \log_2 n - k + 2$, de a, los parámetros y variables locales y 2 de los dos datos de b, al mezclar 4 datos es $n + \log_2 n - 2k + 4$ pues se reserva espacio para parámetros de una llamada menos pero se mezclan el doble de datos; y en general, cuando se mezclan 2^m datos la memoria es $n + k \log_2 n - mk + 2^m$. La mayor ocupación la tendremos cuando $2^m = n$, que es cuando se realiza la última mezcla para tener los datos ordenados. El coste de la ocupación de memoria es en este caso también $o(2n)$, pero es menor que el coste anterior en términos de orden $\log_2 n$.

b) Contaremos el número de comparaciones de elementos del array, no de índices. La ecuación de recurrencia es:

$$c(n) = \begin{cases} 1 & si \quad n = 2 \\ 2c\left(\frac{n}{2}\right) + an + b & si \quad n > 2 \end{cases}$$

donde a y b son $\frac{1}{2}$ y 0, respectivamente, en el caso más favorable, que es cuando todos los elementos de una de las dos partes a mezclar son menores que todos los de la otra; y en el caso más desfavorable, que es cuando los dos últimos elementos de las dos zonas a mezclar son los dos últimos una vez mezclados, son 2 y -1, respectivamente.

La ecuación es idéntica a la de la ordenación por mezcla, por lo que el orden será $\theta(n \log n)$. No vamos a hacer el cálculo pues es repetir el de la ordenación por mezcla con los valores que aparecen en este caso.

Problema 3.16 a) Dar un algoritmo para multiplicar una matriz triangular superior por una completa (considerar matrices cuadradas), y estudiar su tiempo de ejecución.

b) Comentar cómo se implementaría con un método divide y vencerás similar al de Strassen la multiplicación de una matriz triangular por una cuadrada. Habrá que indicar qué funciones sería necesario implementar y qué habría que hacer para ahorrar memoria en las llamadas recursivas. Estudiar también el tiempo de ejecución.

Solución:

a) La multiplicación directa de matrices tiene la forma:

```
for i = 1 to n
    for j = 1 to n
        s = 0
        for k = 1 to n
            s = s + a[i,k] * b[k,j]
        endfor
        c[i,j] = s
    endfor
endfor
```

En el caso de multiplicar una matriz triangular superior por una completa tenemos que desde $k = 1$ hasta $i - 1$ los valores de la primera matriz son ceros, por lo que no hay que realizar las operaciones y la multiplicación queda:

```
for i = 1 to n
    for j = 1 to n
        s = 0
        for k = i to n
            s = s + a[i,k] * b[k,j]
        endfor
        c[i,j] = s
    endfor
endfor
```

Para estudiar el tiempo de ejecución contaremos las operaciones en coma flotante que se realizan (las sumas y multiplicaciones de elementos de las matrices), con lo que se obtiene $\sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=i}^{n} 2 = n^3 + n^2$.

b) Habrá que ver cómo quedan las fórmulas de la multiplicación de Strassen. A_{21} es la matriz nula, y A_{11} y A_{22} son matrices triangulares superiores que llamaremos T_{11} y T_{22}. Las fórmulas para calcular P, Q, R, S, T, U y V son:

$$\begin{array}{rcc}
P & = & (T_{11}+T_{22})(B_{11}+B_{22}) \\
Q & = & T_{22}B_{11} \\
R & = & T_{11}(B_{12}-B_{22}) \\
S & = & T_{22}(B_{21}-B_{11}) \\
T & = & (T_{11}+A_{12})B_{22} \\
U & = & -T_{11}(B_{11}+B_{12}) \\
V & = & (A_{12}-T_{22})(B_{21}+B_{22})
\end{array}$$

Para calcular las submatrices de C se utilizan las mismas fórmulas pues todas las matrices que intervienen son completas.

Habrá que implementar la suma y resta de matrices completas, pero también la suma de matrices triangulares o de una triangular y una completa, y habrá que hacer llamadas recursivas a la función que multiplica matrices triangulares superiores por completas (cálculo de P, Q, R, S y U) e implementar un método eficiente para multiplicar matrices completas (por ejemplo el método de Strassen para obtener T y V).

La matriz para la que se necesita más memoria en las llamadas recursivas es V pues es la única en la que las multiplicaciones son de matrices completas y antes hay que haces sumas o restas de matrices que hay que almacenar en alguna zona de memoria. Se necesitaría $\frac{3n^2}{4}$ para V y las dos matrices temporales, pero como V sólo se utiliza en C_{11}, se puede calcular la primera y almacenar en C_{11} directamente. De este modo las necesidades de memoria son $\frac{2n^2}{4}$.

La otra matriz que se obtiene multiplicando matrices completas es T, pero sólo se necesita una matriz temporal, por lo que se pueden usar las zonas de las matrices temporales de V.

De las otras matrices la única que necesita dos temporales es P, que se puede calcular después de V y T y almacenar directamente en C_{22}, y sumarla a C_{11}, con lo que también es suficiente con las zonas usadas en el cálculo de V.

Como en el resto de las matrices sólo es necesaria una matriz temporal, las dos matrices temporales de dimensión $\frac{n}{2} \times \frac{n}{2}$ reservadas en el cálculo de V son suficientes.

Para obtenes el tiempo de ejecución hay que tener en cuenta que para resolver un problema de dimensión n se resuelven 5 del mismo tipo de dimensión la mitad, se hacen 2 multiplicaciones de matrices completas de dimensión la mitad, se hacen 5 sumas o restas de matrices completas, y 3 sumas o restas de matrices donde al menos una de las matrices es triangular. La ecuación de recurrencia teniendo en cuenta sólo las operaciones de mayor orden queda $t(n) = 5t\left(\frac{n}{2}\right) + 2a\left(\frac{n}{2}\right)^{2.81} + 5\frac{n^2}{4} + 3\frac{n^2}{8}$, donde a

representa la constante que aparece en la multiplicación de Strassen en el término de mayor orden.

La ecuación característica queda $(x-5)(x-7)(x-4)=0$, con lo que el tiempo queda $t(n)=t_k=c_1 5^k+c_2 7^k+c_3 4^k$, y para obtener las constantes habrá que plantear un sistema con t_0, t_1 y t_2. El valor de c_2 que se obtiene es $\frac{a}{2}$, con lo que esta manera de hacer la multiplicación será el doble de rápida (sin contar los términos de menor orden) que si se hace con el método de Strassen.

Capítulo 4

ALGORITMOS VORACES

4.1 Método general

Los algoritmos **voraces** o de **avance rápido** (greedy en inglés) se utilizan normalmente para obtener soluciones iniciales aproximadas en problemas de optimización, realizándose después una búsqueda local para obtener una solución óptima. Pero en algunos casos los métodos voraces pueden dar lugar a soluciones óptimas. Se trata por tanto de obtener una solución que debe cumplir unas ciertas condiciones dadas por ecuaciones y minimizar o maximizar una función de coste. Una solución se obtiene tomando de entre una serie de entradas unas determinadas en un orden determinado, y esto se hace en varios pasos, decidiendo en cada uno de los pasos la inclusión en la solución de una entrada, teniendo en cuenta que esta inclusión no debe hacer que la nueva solución parcial lleve a transgredir las condiciones que debe cumplir una solución, que la selección del elemento a incluir se realizará por medio de una función de selección que se intentará que asegure el acercamiento a soluciones óptimas, y que no se desanda el camino andado (de ahí el nombre de avance rápido).

Un esquema general sería:

Algoritmo 4.1 *Esquema general de la técnica de* **avance rápido**

```
PROCEDURE voraz(a:ARRAY[1..n] OF tipo)
VAR
  i:indice
  x:tipo
BEGIN
  FOR i = 1 TO m
    x=seleccion(a)
    IF posible(solucion,x)
      añadir(x,solucion)
    ENDIF
  ENDFOR
```

ENDvoraz

donde

- solucion es el array global donde almacenamos las soluciones parciales que vamos obteniendo y al final tendremos la solución total,
- se realizan un número fijo de pasos que en el algoritmo aparece como m, aunque no es necesario que se realicen todos los pasos, sino que es posible que antes de llegar a realizarse todos se cumpla alguna condición que nos permita acabar la ejecución,
- *seleccion* selecciona la siguiente entrada candidata a entrar en la solución,
- *posible* comprueba que el añadir la entrada seleccionada a la solución actual nos sigue dando lugar a una solución posible,
- *añadir* añade la nueva entrada seleccionada a la solución actual.

4.1.1 Devolución de monedas

Un ejemplo típico de resolución de un problema por una técnica voraz es el de devolución de una cierta cantidad con un número mínimo de monedas. Tenemos monedas de unas cantidades determinadas c_1, c_2, ... y queremos devolver una cantidad C de manera que el número de monedas devueltas sea mínimo. En este caso la restricción viene dada por $\sum_{i=1}^{n} n_i c_i = C$ y la función de costo que hay que minimizar es $\sum_{i=1}^{n} n_i$, siendo n_i el número de monedas de valor c_i que se devuelven. La función de selección consistirá en tomar la moneda de mayor valor de las que no se han gastado todavía, la inclusión será posible si la suma de las monedas que tenemos hasta ese momento no excede a C.

Este método no lleva siempre a soluciones óptimas, como se ve tomando unas cantidades ilimitadas de monedas de valor 25, 12 y 1 pesetas, y $C = 36$. Se daría una moneda de 25 y once de 1 peseta, mientras que la solución óptima es dar 3 monedas de 12 pesetas.

Tampoco se puede asegurar que el problema tenga solución, por ejemplo si tenemos monedas de cantidad 25 y 5 y $C = 36$.

4.1.2 Paseo del caballo

Otro ejemplo es el algoritmo de Warnsdorff para resolver el problema de recorrer todas las casillas de un tablero de ajedrez sin pasar dos veces por una misma casilla y utilizando un caballo.

La solución viene dada en este caso por un conjunto ordenado de pares que indican las sucesivas posiciones por las que pasa el caballo en su paseo. El algoritmo utiliza como función de selección la que consiste en tomar como siguiente posición una de las casillas a las que puede acceder el caballo desde su posición actual y que es accesible desde menos casillas todavía no recorridas. Se puede ver que dependiendo del tamaño del tablero el algoritmo nos lleva a una solución óptima o no encuentra solución, como se ve en la figura:

1	20	9	14	3
10	15	2	19	24
21	8	23	4	13
16	11	6	25	18
7	22	17	12	5

1		5	10
6	9	2	
	4	11	8
12	7		3

donde el número de las casillas determina el orden en que se recorren.

4.2 Problema de la mochila

4.2.1 Planteamiento

Tenemos n objetos y una mochila. La mochila tiene capacidad M (admite un peso M) y cada objeto i tiene un peso w_i y un beneficio asociado p_i que se obtendrá si se mete ese objeto en la mochila. Si suponemos que los objetos se pueden partir y que la porción que se mete de un objeto en la mochila es x_i con $0 \leq x_i \leq 1$, resolver el problema consistirá en maximizar $\sum_{i=1}^{n} p_i x_i$, sujeto a $\sum_{i=1}^{n} w_i x_i \leq M$, $0 \leq x_i \leq 1$ (además p_i y w_i deben ser positivos). Una solución será una ordenación de x_i que satisfaga las restricciones, y será óptima si maximiza el beneficio.

Un primer algoritmo voraz puede consistir en ordenar los objetos en orden decreciente de beneficio e ir metiendo en la mochila objetos mientras nos quepan enteros, y cuando no quepa uno entero añadir la porción que quepa del siguiente objeto.

Vemos que no seguimos exactamente el esquema propuesto, sino que se realiza un preproceso (la ordenación) antes de aplicar un esquema similar al propuesto.

Ejemplo 4.1 Con este ejemplo veremos que el método anterior no nos asegura que lleguemos a una solución óptima.

Consideramos $n = 3$, $M = 20$, $p = (25, 24, 15)$ y $w = (18, 15, 10)$.

Con el método anterior la solución será $\left(1, \frac{2}{15}, 0\right)$ que tiene beneficio $25+2\frac{24}{15} = 28.2$. Esta solución no es óptima pues la solución $\left(\frac{5}{18}, 1, 0\right)$ tiene beneficio $24 + 25\frac{5}{18} = 30.9$.

De lo anterior no se puede deducir que no se pueda resolver de manera óptima el problema de la mochila por avance rápido, sino simplemente que quizás no hemos dado con una función de selección apropiada.

4.2.2 Solución óptima

Con el siguiente teorema se obtiene un método de avance rápido con el que se encuentra una solución óptima.

Teorema 4.1 *Si se ordenan los objetos con* $\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \ldots$ *y se meten en la mochila enteros en este orden mientras quepan y cuando no quede capacidad para uno entero se mete la fracción correspondiente, la solución es óptima.*

Demostración:

Sea $X = (x_1, x_2, \ldots, x_n)$ una solución obtenida de esta manera. Si todos los $x_i = 1$ la solución es óptima, por lo que supondremos que no son todos 1.

Sea j el menor tal que $x_j \neq 1$.

Suponemos una solución $Y = (y_1, y_2, \ldots, y_n)$, y k el menor tal que $y_k \neq x_k$.

Se pueden presentar tres casos:

- $k < j \Rightarrow x_k = 1 \Rightarrow y_k < x_k$,
- $k = j \Rightarrow$ (como $x_i = y_i$ con $i < k$) es $y_k < x_k$ pues sino tendríamos $\sum w_i y_i > M$,
- $k > j \Rightarrow \sum w_i y_i > \sum w_i x_i = M$, lo que no es posible,

por tanto, tendremos que $k \leq j$ y $y_k < x_k$.

Podemos obtener otra solución incrementando y_k hasta x_k y decrementando $y_{k+1}, \ldots, y_n$ para no exceder la capacidad de la mochila. Llamamos a esta nueva solución $Z = (z_1, z_2, \ldots, z_n)$, siendo la cantidad de peso que añadimos igual a la que quitamos, por lo que:

$$(z_k - y_k)\, w_k = \sum_{i=k+1}^{n} (y_i - z_i)\, w_i \tag{4.1}$$

y además

$$\begin{aligned}
\sum_{i=1}^{n} p_i z_i = \sum_{i=1}^{n} p_i y_i + (z_k - y_k)\, p_k - \sum_{i=k+1}^{n} (y_i - z_i)\, p_i = \\
\sum_{i=1}^{n} p_i y_i + (z_k - y_k)\, p_k \frac{w_k}{w_k} - \sum_{i=k+1}^{n} (y_i - z_i)\, p_i \frac{w_i}{w_i} \geq \\
\sum_{i=1}^{n} p_i y_i + \left((z_k - y_k)\, w_k - \sum_{i=k+1}^{n} (y_i - z_i)\, w_i \right) \frac{p_k}{w_k} = \sum_{i=1}^{n} p_i y_i
\end{aligned} \tag{4.2}$$

en cuyo caso hemos obtenido Z con beneficio mayor o igual que el de Y que tiene los valores iguales a los de X hasta la posición k. En un número finito de pasos llegaríamos a la solución X.

Ejemplo 4.2 Aplicando al ejemplo 4.1 el método dado en el teorema 4.1 se ordenan los objetos de la forma $p = (24, 15, 25)$, $w = (15, 10, 18)$, y la solución óptima es $x = \left(1, \frac{1}{2}, 0\right)$, con beneficio $24 + 15\frac{1}{2} = 31.5$.

4.2.3 Algoritmo

Obtener un algoritmo una vez ordenados los objetos de acuerdo con el teorema 4.1 es bien sencillo:

Algoritmo 4.2

```
PROCEDURE MochilaVoraz(p, w, x:ARRAY[1..n] OF REAL;M:REAL)
VAR
   i:INTEGER
   resto:REAL
BEGIN
   FOR i = 1 TO n
      x[i] = 0
   ENDFOR
   resto = M
   i = 1 (*indica el elemento que introducimos*)
   WHILE w[i] ≤ resto AND i ≤ n
      x[i] = 1
      resto = resto − w[i]
      i = i + 1
   ENDWHILE
   IF i ≤ n
      x[i] = resto/w[i]
   ENDIF
ENDMochilaVoraz
```

El tiempo de ejecución depende (si no contamos la parte de inicialización a cero del vector de soluciones y de ordenación de los objetos) del número de veces que se ejecute el WHILE y este número será cero en el mejor de los casos (cuando no podemos meter ningún elemento en la mochila) y n en el peor (cuando podemos meterlos todos).

En este caso los términos "mejor" y "peor" no son muy apropiados pues no podemos considerar lo "mejor" no poder meter ningún elemento en la mochila y por tanto tener beneficio cero.

4.3 Secuenciamiento de trabajos a plazos

4.3.1 Planteamiento

Tenemos n trabajos cuya ejecución lleva una unidad de tiempo y que tienen que ser ejecutados en una misma máquina sin poder ésta ser compartida. Cada trabajo i tiene asociado un plazo d_i y un beneficio p_i de manera que si la ejecución del trabajo i se empieza antes del plazo d_i se tiene un beneficio p_i. Una solución será una ordenación de los trabajos iniciados dentro de sus plazos, y de entre todas las posibles soluciones tratamos de encontrar la que maximice $\sum_{i \in I} p_i$, siendo I el conjunto de los trabajos que empiezan a ejecutarse dentro de sus plazos.

Una posible manera de resolver el problema sería tomar todas las posibles soluciones (para lo que habría que generar las n! posibles ordenaciones de los trabajos y ver las que corresponden a soluciones), calcular los beneficios asociados y obtener la de mayor beneficio.

Ejemplo 4.3 Vemos cómo trabaja este método con los siguientes valores: $n = 4$, $p = (100, 10, 15, 27)$ y $d = (2, 1, 2, 1)$.

Habría que generar $n! = 24$ posibles soluciones, pero como el mayor plazo es 2 es suficiente con tomar permutaciones de los cuatro trabajos de dos en dos, con lo que habrá que generar 12 pares, comprobar los que son solución, obtener su beneficio y quedarse con el de mayor beneficio:

par	solución	beneficio
1,2	no	
1,3	si	115
1,4	no	
2,1	si	110
2,3	si	25
2,4	no	
3,1	si	115
3,2	no	
3,4	no	
4,1	si	**127**
4,2	no	
4,3	si	42

en donde se ve que la solución óptima es la que corresponde a la ordenación 4,1.

El método descrito es obviamente muy malo pues tiene un coste $\theta(n!)$.

4.3.2 Solución óptima

Se puede utilizar un método voraz consistente en añadir en cada paso el trabajo de mayor p_i de modo que el conjunto sea realizable, donde que un conjunto sea realizable significa que existe alguna ordenación del conjunto con la que los trabajos se pueden ejecutar dentro de sus plazos. Este método voraz da lugar a una solución óptima, y para demostrar esto utilizaremos un lema previo.

Lema 4.1 *Si J es un conjunto de k tareas y $\sigma = (s_1, s_2, \ldots, s_k)$ es una permutación de J tal que $d_{s_1} \leq d_{s_2} \leq \ldots \leq d_{s_k}$, entonces J es realizable $\Leftrightarrow$ σ lo es.*

Demostración:

La implicación $\Rightarrow$ se cumple por la definición de que un conjunto sea realizable.

Veremos la implicación en el otro sentido.

J es realizable si $\exists \rho = (r_1, \ldots, r_k)$ con $d_{r_i} \geq i \; \forall i = 1, 2, \ldots, k$. Veremos que se puede modificar ρ hasta llegar a σ en un número finito de pasos de manera que la ordenación que obtenemos en cada paso corresponde a una solución.

Sea a el índice más pequeño con $s_a \neq r_a$.

Si r_a está antes que s_a en σ, sería $r_a = s_b$ con $b < a$, y a no sería el menor tal que $s_a \neq r_a$. Por tanto, r_a está tras s_a en σ y $d_{r_a} \geq d_{s_a} = d_{r_b}$, y se pueden intercambiar r_a y r_b en ρ obtniendo una nueva solución.

Intercambiando r_a y r_b se obtiene una nueva ordenación correspondiente a una solución siendo esta solución y σ iguales al menos hasta la posición a.

Repitiendo el proceso llegamos en un número finito de pasos a que σ es realizable.

Teorema 4.2 *El método voraz consistente en incluir en cada paso el trabajo con mayor p_i de modo que la solución parcial obtenida sea realizable lleva a una solución óptima.*

Demostración:

Suponemos que el método da lugar a un conjunto I con una realización S_I, y que J es una solución óptima con realización S_J.

Tal como se hacía en el lema 4.1 se pueden hacer transformaciones en S_I y S_J de manera que los trabajos comunes estén en el mismo sitio.

Si en una posición de I tenemos un trabajo a y en la correspondiente posición de J no tenemos nada se podría poner a en J y este no sería óptimo.

Si en una posición de J se ejecuta a no es posible que la correspondiente posición de I esté vacía.

Si tenemos en una misma posición de I y J distintos a y b, $a \in I$ y $b \in J$ pueden ocurrir varios casos:

- si $p_a > p_b$ se podría mejorar J,
- si $p_a < p_b$ el algoritmo habría tomado b y no a,

- por tanto, es $p_a = p_b$.

y como en cada posición los beneficios son iguales I es también óptima.

4.3.3 Algoritmo

Basándonos en las observaciones anteriores podemos diseñar un algoritmo voraz óptimo.

Algoritmo 4.3

PROCEDURE trabajos(d:ARRAY[1..n] OF REAL;VAR s:ARRAY[0..n] OF 0..n)

(*En d tendremos los plazos asociados a los trabajos, los trabajos estarán ordenados de mayor a menor beneficio, y la solución la almacenaremos en s indicando el número de trabajo que se ejecuta*)

VAR
    $k : 1..n$
    l, r, i:INTEGER
BEGIN
    $d[0] = 0$
    $s[0] = 0$
(*$d[0]$ y $s[0]$ actúan de centinelas*)
    $k = 1$
    $s[1] = 1$
(*incluye el primer trabajo*)
    FOR $i = 2$ TO n
        $r = k$

(*k indica la cantidad de trabajos incluidos, y r tendrá la posición donde se ejecuta el nuevo trabajo*)

        WHILE $d[s[r]] > d[i]$ AND $d[s[r]] \neq r$
            $r = r - 1$
        ENDWHILE

(*se busca mantener la solución ordenada crecientemente por d. Esto se consigue moviendo a la derecha los trabajos siempre que queden dentro de su plazo*)

        IF $d[s[r]] \leq d[i]$ AND $d[i] > r$
(*si están en orden y se puede incluir el trabajo i*)
            FOR $l = k$ DOWNTO $r + 1$
                $s[l + 1] = s[l]$
            ENDFOR
            $s[r + 1] = i$
(*ha puesto i en su lugar de ejecución*)
            $k = k + 1$
        ENDIF
    ENDFOR

ENDtrabajos

Para el cálculo del tiempo hay que tener en cuenta que suponemos que los trabajos están ordenados por el beneficio.

El caso más favorable (en cuanto a tiempo de ejecución) será que sólo se pueda ejecutar el primer trabajo, pues en este caso el cuerpo del WHILE y del IF no se ejecuta nunca, con lo que tendríamos un tiempo $a+\sum_{i=2}^{n} b = a+b(n-1)$, y sería $\Omega(n)$.

El caso más favorable suponiendo que se pueden ejecutar todos los trabajos lo tenemos cuando estén en orden de ejecución, pues en este caso tampoco se ejecuta el cuerpo del WHILE ni del IF y tendríamos $\Omega(n)$.

El caso más desfavorable corresponde a que se ejecuten todos los trabajos lo más alejados posible de su posición inicial, pues en este caso el cuerpo del WHILE se ejecuta $i-1$ veces y el FOR interno al IF se ejecuta con índices de $i-1$ a 1, con lo que el tiempo de ejecución será:

$$a+\sum_{i=2}^{n}\left(b+i-1+\sum_{j=1}^{i-1} c\right)=$$
$$a+\sum_{i=2}^{n}(b+(c+1)(i-1)) = a+b(n-1)+(c+1)\frac{n(n-1)}{2} \tag{4.3}$$

por lo que el algoritmo tiene un orden $O\left(n^2\right)$.

En realidad, el estudio suponiendo que se ejecutan todos los trabajos se reduce al estudio de una ordenación secuencial, por lo que el cálculo del tiempo promedio con dicha suposición nos daría un orden $O\left(n^2\right)$. Para demostrar esto, vemos que el trabajo i habrá que ponerlo en un lugar j con j entre 1 e i con probabilidad $\frac{1}{i}$, y el poner el trabajo i en el lugar j conlleva un coste $i-j$, por lo que el tiempo promedio es:

$$\sum_{i=2}^{n}\left(\frac{1}{i}\sum_{j=1}^{i}(i-j)\right) = \sum_{i=2}^{n}\frac{1}{i}(i-1)\frac{i}{2} = \sum_{i=2}^{n}\frac{i-1}{2} = \frac{(n-1)^2}{4} \in \theta\left(n^2\right) \tag{4.4}$$

donde no hemos considerado las constantes por cuestión de claridad.

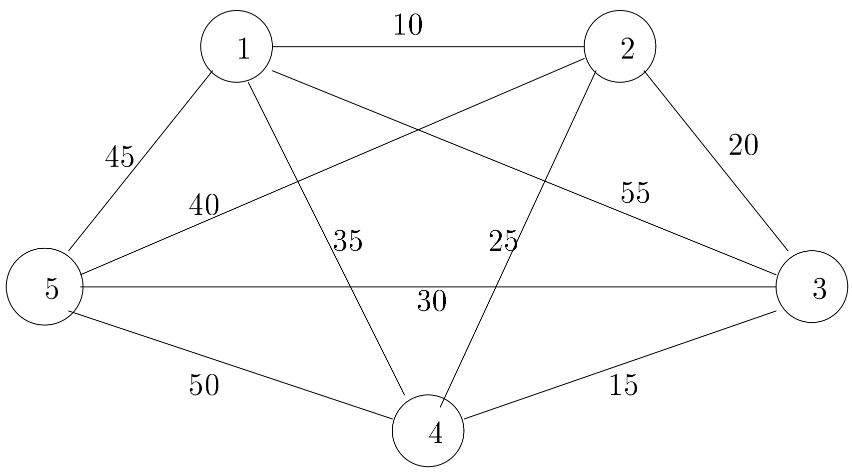
4.4 Heurísticas voraces

Hemos visto dos ejemplos en que un método de avance rápido lleva a una solución óptima, pero hay problemas con los que se puede usar esta técnica para obtener una solución no óptima intentando que esté próxima a una óptima para encontrar ésta utilizando una búsqueda local en el espacio de las soluciones. De esta manera se reduce el espacio de búsqueda de soluciones utilizando previamente un método voraz cuyo tiempo de ejecución será lineal pues se ejecuta un número n de pasos. La función de selección en estos casos se determina de modo heurístico, lo que quiere decir que nos basamos en el conocimiento (muchas veces intuitivo y no formalizado) que tengamos del problema para considerar que una función nos puede acercar más que otra a una solución óptima.

4.4.1 Problema del viajante

Vemos el problema del viajante, donde tenemos n ciudades con carreteras que las unen dos a dos y por las que se puede circular en las dos direcciones, y el viajante quiere realizar un recorrido por las n ciudades partiendo y finalizando en una de ellas que llamaremos 1, sin pasar dos veces por una misma ciudad y minimizando el número de kilómetros a recorrer.

Ejemplo 4.4 Como ejemplo, sea el grafo:

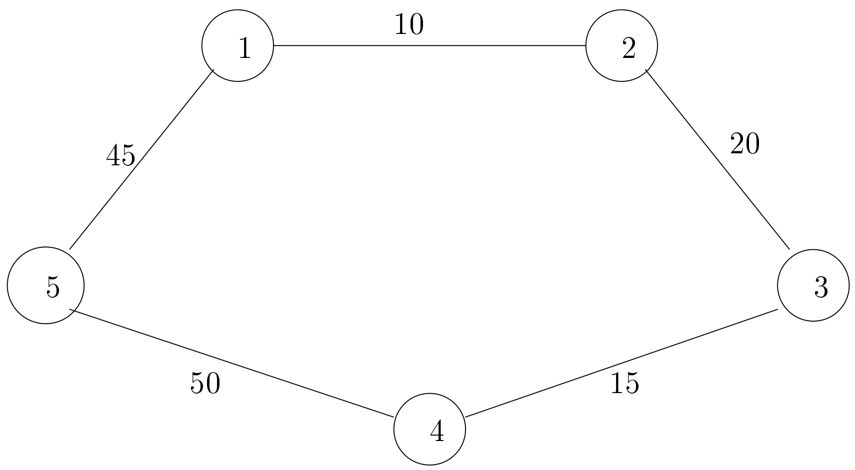


que representa el mapa de carreteras que unen las cinco ciudades a recorrer por el viajante. Un método de avance rápido para encontrar una solución que no sea óptima puede ser ordenar de menor a mayor peso las aristas, e ir incluyendo las $n-1$ aristas necesarias para tener una solución de manera que la inclusión de una arista no produzca que se cierre un ciclo (salvo la última arista que tendrá necesariamente que cerrarlo) ni que de un nodo salgan tres aristas. El número de pasos a dar es como mucho a siendo a el número de aristas del grafo, pero esto no quiere decir que el orden sea $\theta(a)$, ya que para cada arista hay que comprobar que no produce que de un nodo salgan tres aristas (lo que se puede hacer en tiempo constante) y que no cierra un ciclo, lo que se puede hacer llevando conjuntos de nodos conectados y la comprobación de que dos nodos no están en el mismo conjunto no tiene porqué poder hacerse en un tiempo constante.

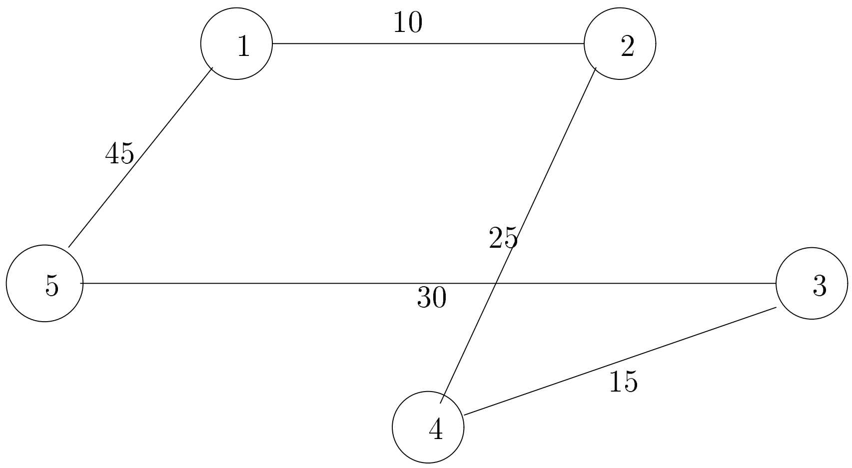
La ordenación de las aristas en este ejemplo será:

arista	peso
1,2	10
3,4	15
2,3	20
2,4	25
3,5	30
1,4	35
2,5	40
1,5	45
4,5	50
1,3	55

y el grafo solución dado por este método es:

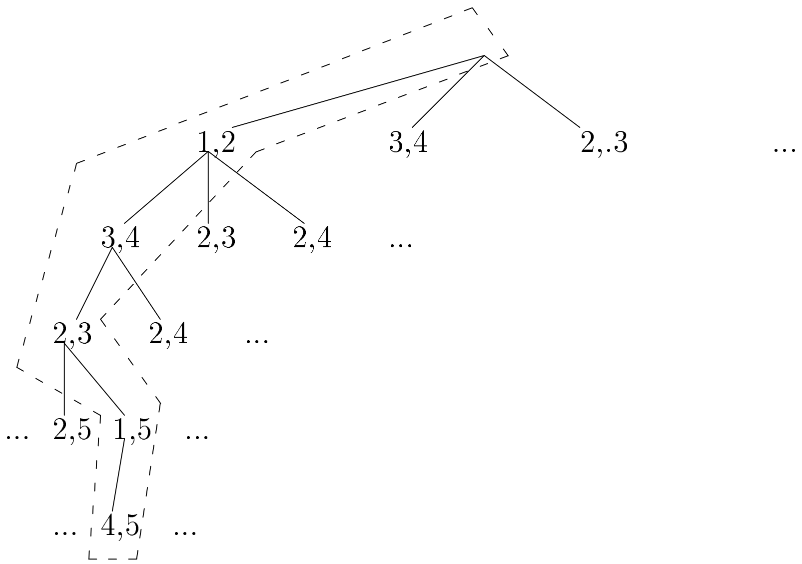


A partir de esta solución se pueden encontrar soluciones mejores por búsqueda local haciendo varios pasos de modo que en cada uno de ellos intentemos quitar la arista de mayor peso. Por ejemplo, podemos intentar quitar la arista 4,5 poniendo la 4,1, lo que produce que tengamos que quitar una arista del nodo 1, que puede ser la arista 1,2, pero no es posible poner la 1,3 ni la 1,4 ni la 1,5, por lo que no es posible sustituir la arista 4,5 por la 1,4. Haciendo sucesivas pruebas vemos que una posibilidad es quitar la 4,5 y poner la 4,2, quitar la 2,3 y poner la 3,5. De este modo se obtiene la solución:



que es de menor costo que la solución anterior.

Podemos considerar que el avance rápido nos ha dado la solución que marcamos en el árbol de soluciones:



y una búsqueda local a partir de esta solución puede ser un backtracking empezando desde ese nodo, o desde algún nivel superior a ese nodo, siendo la búsqueda tanto menos local cuanto más se suba de nivel.

4.5 Problemas

Problema 4.1 Tenemos una tabla:

	a	b	c
a	2	1	1
b	4	3	2
c	2	3	1

donde a, b, y c son trabajos que no se pueden ejecutar simultáneamente. Los números en la tabla indican el beneficio que se tendría ejecutando el trabajo de la vertical y a continuación el de la horizontal (ba tendría beneficio 4 y ab beneficio 1). Idear un algoritmo basado en el método de avance rápido para maximizar el beneficio. Tendremos que ejecutar los trabajos a, b y c un número n_a, n_b y n_c de veces, respectivamente. Mostrar que el algoritmo no es óptimo.

Problema 4.2 Dado un grafo multietapa con pesos en las aristas $G = (V, A)$, con V el conjunto de los vértices y A el de las aristas, consideramos que cumple:

el conjunto de vértices está dividido en una serie de subconjuntos V_1, V_2, ..., V_m,

V_1 y V_m contienen un único vértice,

cada uno de los subconjuntos V_2, ..., V_{m-1} consta del mismo número de vértices (v),

cada arista de G con origen en V_i tiene destino en V_{i+1} (estamos en un grafo multietapa).

a) Diseñar un algoritmo por avance rápido para el problema de obtener el camino mínimo del vértice de V_1 al de V_m. Calcular su tiempo de ejecución en función de v y del tamaño del problema $n = 2 + v(m - 2)$.

b) Dado que el algoritmo del apartado a) no obtendrá la solución óptima, indicar cómo se podrían diseñar algoritmos de avance rápido para este problema aumentando el coste de cada una de las decisiones pero de manera que se aumenten las posibilidades de llegar a una solución óptima.

Solución:

a) Un esquema para solucionar el problema por avance rápido podría ser:

```
s[1] = 1
seguir = true
i = 2
while seguir and i ≤ m
    x =menor(s[i − 1])    (*función siguiente*)
    if x ≠ ∞    (*función posible*)
        s[i] = x    (*función añadir*)
        i = i + 1
    else
        seguir = false
    endif
endwhile
```

Donde supondremos que $V_1 = 1$; y que la solución se guardará en el array s de m posiciones, indicando $s[i] = j$ que del i-ésimo conjunto se ha tomado el vértice j. Si se llega a $i = m$ se ha encontrado una solución.

La forma de la función `menor` dependerá de la representación del grafo. Si suponemos que tenemos una tabla *nodo* de 1 a n, habiendo en $nodo[i]$ un puntero *lista* a una lista de nodos que representan las aristas que salen del vértice i, y apuntando *lista* a un registro con tres campos:

arista=record
  destino:integer (*es el nodo vértice destino de la arista*)
  peso:integer (*es el peso de la arista*)
  siguiente:puntero a *arista* (*puntero a la siguiente arista que sale del vértice*)
endrecord

Con esta representación `menor` podría ser:

`menor`($vertice$):
  $lista = nodo[vertice]$
  $min = \infty$
  $x = \infty$
  while $lista \neq NIL$
    if $lista \to valor < min$
      $min = lista \to valor$
      $x = lista \to nodo$
    endif
    $lista = lista \to siguiente$
  endwhile
  return x

Si el grafo se almacena en una matriz de adyacencia t de n filas y columnas, representando $t[i, j]$ el peso de la arista del vértice i al j, y un ∞ que no existe la arista; la función `menor` podría ser:

`menor`($vertice$):
  $min = \infty$
  $x = \infty$
  for $i = 1$ to n
    if $t[vertice, i] < min$
      $min = t[vertice, i]$
      $x = i$
    endif
  endfor
  return x

Si suponemos que se encuentra solución y que por lo tanto se ejecuta $m - 1$ veces la función `menor`, y que de cada vértice salen un máximo de v aristas. El coste en el caso de la primera representación será:

$$t(n,v) = 1 + (m-2)(v+1) + 1 = 2 + (m-2)v + m - 2 = n + \frac{n-2}{v} \in O(n)$$

Con la representación de matriz el coste será:

$$t(n,v) = 1 + (m-1)n = 1 + \left(\frac{n-2}{v} + 1\right) n \in O\left(\frac{n^2}{v}\right)$$

b) Si cada una de las decisiones se basa en explorar dos niveles en vez de uno el coste de tomar cada decisión será mayor, pero esto se puede ver compensado por una probabilidad mayor de llegar a una solución óptima. En este caso la función `menor` variaría para explorar dos niveles salvo en la última decisión en la que sólo queda un nivel por explorar. Con la representación con matrices de adyacencia `menor` podría ser:

```
menor(vertice,nivel):
  if nivel = m
    el código de la función en el caso anterior
  else
    min = ∞
    x = ∞
    for i = 1 to n
      if t[vertice, i] ≠ ∞
        for j = 1 to n
          if t[vertice, i] + t[i, j] < min
            min = t[vertice, i] + t[i, j]
            x = i
          endif
        endfor
      endif
    endfor
    return x
  endif
```

El coste en este caso será $t(n,v) \in O\left(\frac{n^3}{v}\right)$, y de manera análoga se haría con la representación de listas.

Problema 4.3 Consideramos el problema de la mochila modificado, donde tenemos una mochila de capacidad M, n objetos con beneficios $b = (b_1, b_2, \ldots, b_n)$ y pesos $p = (p_1, p_2, \ldots, p_n)$, y que cada objeto puede meterse dentro de la mochila, no meterse, o meterse la mitad del objeto obteniendo la mitad del beneficio. Programar un esquema de avance rápido para resolver este problema. Justificar si se encuentra la solución óptima o no. Estudiar su tiempo de ejecución.

Solución:

Como el problema no 0/1 se resuelve de manera óptima ordenando los objetos de mayor a menor $\frac{b}{p}$, en este caso los ordenaremos también así considerando que si no llegamos a una solución óptima sí llegaremos a una cercana. Después de eso se irán introduciendo elementos en la mochila, metiendo el elemento entero si cabe y si no cabe entero se intenta con la mitad. Si un elemento no se puede meter ni entero ni en parte no podemos descartar que alguno de los siguientes se pueda meter, por lo que habrá que continuar con todos los elementos. De esta forma el programa puede ser:

$ben = 0$
$cap = M$
$s \leftarrow 0$
ordenar objetos de mayor a menor $\frac{b}{p}$
for $i = 1$ to n
  if $p[i] \leq cap$
    $s[i] = 1$
    $cap = cap - p[i]$
    $ben = ben + b[i]$
  else if $\frac{p[i]}{2} \leq cap$
    $s[i] = \frac{1}{2}$
    $cap = cap - \frac{p[i]}{2}$
    $ben = ben + \frac{b[i]}{2}$
  endif
endfor

El algoritmo no da la solución óptima. Por ejemplo, con $b = (10, 6)$, $p = (8, 5)$, y $M = 5$, la solución que se obtiene es $0.5, 0$, que da beneficio 5, y la $0, 1$ da beneficio 6.

En cuanto al tiempo de ejecución, la ordenación se hace en un tiempo $n \log_2 n$, y después se entra en un bucle con n pasos, con el coste de cada paso acotado superior e inferiormente por una constante, con lo que el orden será el de la ordenación.

Problema 4.4 Dada una tabla de números, como por ejemplo:

7	3	2	5
3	1	3	2
1	2	7	8
2	1	3	2

se trata de encontrar, de entre todas las sucesiones de números que se forman a partir de la tabla empezando por la fila uno y tomando un número de cada fila estanto un número de una fila adyacente en vertical o diagonal con el de la fila anterior, la sucesión de números con suma máxima.

a) Programar un método de avance rápido para resolver el problema (no obtendrá la solución óptima pero se intentará que se acerque a ella), indicando cómo funciona con la tabla ejemplo.

b) Estudiar el tiempo de ejecución del programa.
c) Indicar alguna idea para mejorar la solución obtenida pero siguiendo utilizando un esquema de avance rápido, indicar cómo trabajaría con el ejemplo y cómo afectaría al tiempo de ejecución.
Solución:
a) En un método de avance rápido se toman una serie de decisiones. En nuestro caso la primera decisión será elegir de la primera fila el mayor elemento y de las filas siguientes el mayor de las tres (o de las dos si estamos en el borde de la tabla) casillas adyacentes en diagonal y vertical al elemento tomado de la fila anterior. Una solución vendrá representada por un array s de 1 a n (número de filas) con valores de 1 a m (número de columnas), donde $s[i] = j$ indica que de la fila i se toma el elemento en la columna j. El esquema del algoritmo será:

```
for i = 1 to n
    seleccionar(s,i)
endfor
```

donde en seleccionar se toman los elementos según hemos explicado:

```
seleccionar(s,i):
    max = 0 /*O un valor mínimo*/
    if i = 1 /*Primera fila*/
        for j = 1 to m
            if t[1, j] > max /*t representa la tabla*/
                s[1] = j
                max = t[1, j]
            endif
        endfor
    else
        if t[i, s[i − 1]] > max /*Elemento en la misma columna*/
            s[i] = s[i − 1]
            max = t[i, s[i]]
        endif
        if s[i − 1] > 1 /*No en la primera columna*/
            if t[i, s[i − 1] − 1] > max /*Elemento en la columna anterior*/
                s[i] = s[i − 1] − 1
                max = t[i, s[i]]
            endif
        endif
        if s[i − 1] < m /*No en la última columna*/
            if t[i, s[i − 1] + 1] > max /*Elemento en la columna siguiente*/
                s[i] = s[i − 1] + 1
            endif
        endif
```

endif

En la tabla ejemplo el mayor de la primera fila es el 7, por lo que $s[1] = 1$; de los dos adyacentes a él en vertical y diagonal el mayor es el 3, en la columna 1, por lo que $s[2] = 1$; el mayor adyacente es el 2, en la columna 2, por lo que $s[3] = 2$; y el mayor adyacente es el 3, en la columna 3, por lo que $s[4] = 3$. La solución es $(1, 1, 2, 3)$ y su valor es 15, que no es óptimo.

b) Con $i = 1$ seleccionar tiene un coste m, y con i entre 2 y n se toma el máximo de dos o tres elementos, por lo que tenemos un coste constante en cada paso, y el tiempo será $t(n, m) = am + b(n - 1) \in \theta(n + m)$. Si la tabla es cuadrada el coste será lineal.

c) No haremos un programa en este caso pues no nos lo piden. La idea podría ser tomar las decisiones en cada fila no con la información de esa fila sino de ella y de la siguiente: no ir por la casilla de mayor valor sino por la que se obtiene mayor valor con ella y la mayor a la que se puede acceder desde ella. En el ejemplo desde 7 se puede acceder a 3 y 1, por lo que el valor que se obtiene en dos pasos tomando el 7 es 10; desde 3 se puede acceder a 3, 1 y 3, por lo que el valor es 6; desde 2 se puede acceder a 1, 3 y 2, por lo que el valor es 5; y de 5 se puede acceder a 3 y 2, con lo que el valor es 8. Por tanto la primera decisión es $s[1] = 1$. Desde 7 se puede acceder a 3 o a 1; desde 3 se puede acceder a 1 y 2, por lo que su valor es 5; y desde 1 se puede acceder a 1, 2 y 7, por lo que su valor es 8. La segunda decisión será $s[2] = 2$. Las siguientes decisiones serán $s[3] = 3$ y $s[4] = 3$. La solución es $(1, 2, 3, 3)$ y su valor es 18, que es mejor que el obtenido con el primer método pero sigue sin ser óptimo.

El tiempo de ejecución sigue siendo del mismo orden anterior, pues cada decisión se toma haciendo el máximo de unos valores que se obtienen en un tiempo constante, pues para cada elemento es el máximo de él mismo sumado con las posiciones de la siguiente fila adyacentes en diagonal y vertical.

Capítulo 5

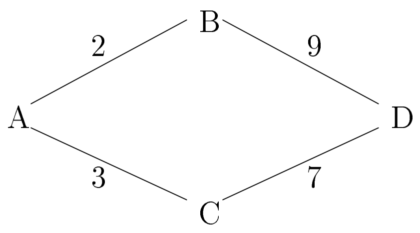
PROGRAMACIÓN DINÁMICA

5.1 Descripción

En algunos problemas que se resuelven con llamadas recursivas a subproblemas de menor tamaño se repiten llamadas a problemas idénticos, con lo que se repiten cálculos innecesariamente pues se podrían haber guardado las soluciones obtenidas anteriormente. Esto ocurre por ejemplo si calculamos los números de Fibonacci por la ecuación $f(n) = f(n-1) + f(n-2)$. Con la técnica de la **programación dinámica** se trata de evitar estos cálculos repetidos guardando una tabla de resultados parciales, es por tanto un método ascendente, donde partimos de problemas de tamaño mínimo y vamos obteniendo resultados de problemas de tamaño cada vez mayor hasta llegar al tamaño del problema a resolver. Se diferencia de la técnica divide y vencerás en que este método es descendente.

Se aplica a problemas de optimización en los que una solución viene dada como una secuencia de decisiones (en el problema de la mochila podría ser decidir si se incluye un elemento o no, o qué elemento se incluye primero) y se utiliza el **principio de optimalidad** que dice que cuando se ha tomado una secuencia óptima de decisiones cada subsecuencia debe ser óptima, por lo que si se han tomado una serie de decisiones la siguiente subsecuencia de decisiones debe ser óptima para el problema en que nos encontremos en ese momento.

El principio de optimalidad no siempre se cumple por lo que, para aplicar este método, deberemos comprobar que sí se puede aplicar en nuestro problema. Por ejemplo, en el problema de calcular el camino mínimo entre las ciudades A y D con el mapa de carreteras de la figura:



una primera decisión sería tomar dirección a B o C. En este momento es óptima $A - B$, pero tomar esta decisión nos lleva a una solución no óptima. Da la impresión de que se cumple el principio de optimalidad ya que en la solución $A - B - D$ las dos decisiones que se han tomado son óptimas en un cierto sentido: la decisión de tomar $A - B$ es óptima pues es la menor de las dos posibilidades en ese momento, y la decisión de tomar $B - D$ es óptima pues es la única posibilidad en ese momento. Pero sin embargo no llegamos a una solución óptima por lo que no puede cumplirse el principio de optimalidad.

Si enfocamos el problema considerando que una solución es óptima si se compone de dos decisiones óptimas en el sentido: la primera solución es óptima para pasar de A a B o C y la segunda decisión es óptima para pasar de B o C a D; es cierto que una solución compuesta de dos soluciones óptimas de los subproblemas es una solución óptima global, pero no podemos asegurar que una solución óptima se pueda descomponer en soluciones óptimas de subproblemas con el tipo de subproblemas que hemos considerado, por lo que no se cumple el principio de optimalidad.

Pero sí se cumple el principio de optimalidad si consideramos que una solución óptima para ir de A a D debe estar compuesta de dos soluciones óptimas para ir de A a B o para ir de A a C y para ir de B a D o para ir de C a D. De esta manera se ve que para resolver algunos problemas es necesario resolver subproblemas de menor tamaño y que tengamos que guardar las soluciones de estos subproblemas para poder reconstruir a partir de ellas la solución del problema inicial. Como no sabemos a priori cuáles de estas soluciones óptimas de subproblemas van a dar lugar a soluciones óptimas del problema global tenemos que guardar toda la información, lo que hace que se necesite mucha memoria y que se hagan cálculos que a la postre se van a revelar como innecesarios.

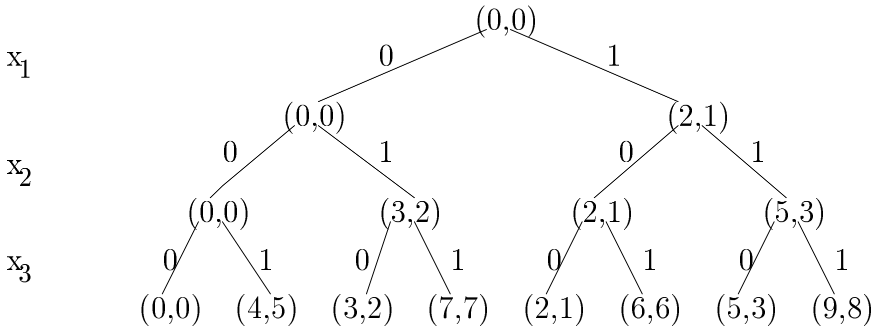
5.2 Problema de la mochila

5.2.1 Planteamiento

Consideramos en este caso el problema de la mochila 0/1 (no se pueden introducir en la mochila porciones de elementos) con lo que cada x_i es 0 ó 1 dependiendo de que no se introduzca o sí se introduzca el elemento en la mochila.

En principio tenemos n objetos numerados de 1 a n y una capacidad de la mochila M, y podemos llamar a este problema $Mochila(1, n, M)$. Para resolverlo tendremos que resolver problemas más pequeños que llamaremos $Mochila(i, j, Y)$, que corresponden al problema de la mochila 0/1 con los objetos numerados de i a j y con capacidad de la mochila Y, por lo que $Mochila(i, j, Y)$ será maximizar $\sum_{k=i}^{j} p_k x_k$ sujeto a $\sum_{k=i}^{j} w_k x_k \leq Y$ y $x_k = 0$ ó $x_k = 1\ \forall k,\ i \leq k \leq j$.

Para resolver el problema suponemos que hay que tomar n decisiones correspondientes a si se mete o no un elemento en la mochila. Por lo tanto, una solución será una secuencia de n ceros y unos, y la solución óptima se podría encontrar por búsqueda exhaustiva, tal como se puede ver en la figura:



correspondiente al ejemplo $n = 3$, $p = (1, 2, 5)$, $w = (2, 3, 4)$, $M = 6$. En este árbol cada nivel corresponde a una decisión sobre si incluir o no un elemento, y los pares de los nodos corresponden al peso y beneficio acumulado, los nodos terminales corresponden a soluciones y la solución óptima será la correspondiente al nodo con peso seis y mayor beneficio. En este ejemplo el beneficio máximo es 6 y corresponde a incluir en la mochila los elementos 1 y 3.

5.2.2 Solución por programación dinámica

Para resolver el problema por un método de programación dinámica llamaremos $f_i(X)$ al valor de la solución óptima del problema $Mochila(1, i, X)$, y se cumplirá la fórmula:

$$f_i(X) = max \{f_{i-1}(X), f_{i-1}(X - w_i) + p_i\} \tag{5.1}$$

según que no se incluya o sí se incluya el elemento i-ésimo. Para resolver $Mochila(1, n, M)$ habrá que calcular $f_n(M)$ en función de $f_{n-1}(M)$ y $f_{n-1}(M - w_n)$, f_{n-1} en función de f_{n-2} y así sucesivamente, por lo que se podría hacer recursivamente con la consiguiente repetición de cálculos que es lo que queremos evitar. Por lo tanto, el proceso en un método de programación dinámica irá en sentido contrario, calculando la función f_0, la f_1, y así sucesivamente hasta llegar a la f_n cuyo valor $f_n(M)$ nos da el óptimo que vamos buscando. De este modo habremos obtenido en el camino valores

$f_i(X)$ innecesarios para la solución que buscamos, y habrá que almacenar en una tabla todos los resultados parciales con el consiguiente desperdicio de memoria.

Ejemplo 5.1

Analizaremos esta técnica con el ejemplo anterior. Para que la fórmula 5.1 sea aplicable para los distintos valores de i y x es necesario imponer unos valores en unos casos base:

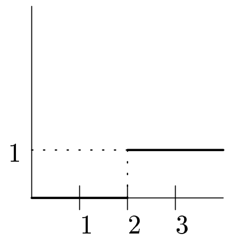
$f_0(x) = 0$ si $x \geq 0$, lo que quiere decir que cuando no se mete ningún elemento el beneficio es 0 sea cual sea la capacidad de la mochila.

$f_i(x) = -\infty$ si $x < 0$, pues este caso no corresponde a un problema real y no hay solución, con lo que cualquier solución lo mejora.

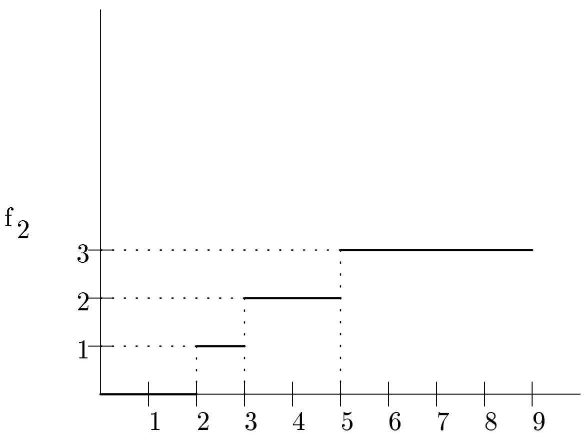
Un primer paso sería calcular f_1:

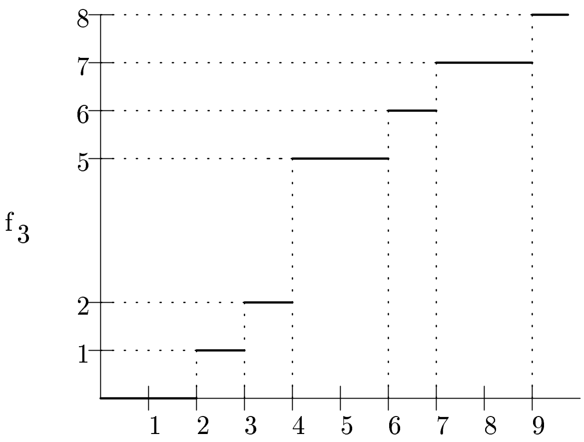
$$\begin{array}{rcl} f_1(0) = & max\{f_0(0), f_0(-2)+1\} & = 0 \\ f_1(1) = & max\{f_0(1), f_0(-1)+1\} & = 0 \\ f_1(2) = & max\{f_0(2), f_0(0)+1\} & = 1 \\ f_1(x) = & max\{f_0(x), f_0(x-2)+1\} & = 1 \; si \; x > 2 \end{array}$$

por lo que la gráfica de f_1 es:



Del mismo modo se calculan f_2 y f_3 y se obtienen las gráficas:





En la gráfica de f_3 se ve que la solución óptima para $M = 6$ es 6, pero de este modo sólo tenemos el beneficio, pero no los elementos que se introducen en la mochila. Para poder reconstruir el camino que nos lleva a la solución óptima necesitaremos dos tablas, una para $f[i,x]$ y otra, que llamaremos $e[i,x]$, para almacenar si se introduce un elemento o no. De este modo tendríamos en memoria las tablas:

f	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1
2	0	0	1	2	2	3	3
3	0	0	1	2	5	5	6

e	0	1	2	3	4	5	6
1	0	0	1	1	1	1	1
2	0	0	0	1	1	1	1
3	0	0	0	0	1	1	1

y para reconstruir la solución: como $e[3,6] = 1$, se ha introducido el objeto tres, por lo que $f_3(6) = f_2(2) + 5$; como $e[2,2] = 0$ no se introduce el elemento 2, y $f_2(2) = f_1(2)$; como $e[1,2] = 1$ sí se introduce el elemento 1, con lo que la solución es (1,0,1).

Vemos que la ocupación de memoria es del orden de nM, y el tiempo de ejecución dependerá del tiempo de cálculo del máximo, siendo en este caso este tiempo constante, por lo que también es del orden de nM.

Se puede ahorrar algo de memoria si se almacenan las funciones f_i como conjuntos de pares (w_j, p_j), donde w_j es un valor de x donde f_i da un salto, y $p_j = f_i(w_j)$. De este modo, en este ejemplo tenemos:

$$\begin{aligned}
f_0(x) &= \{(0,0)\} \\
f_1(x) &= \{(0,0),(2,1)\} \\
f_2(x) &= \{(0,0),(2,1),(3,2),(5,3)\} \\
f_3(x) &= \{(0,0),(2,1),(3,2),(4,5),(6,6),(7,7),(9,8)\}
\end{aligned}$$

Si llamamos S_i al conjunto de los pares que describen f_i y S_{i+1}^1 se obtiene sumando a los pares de S_i el par (w_{i+1}, p_{i+1}), S_{i+1} será $S_{i+1}^1 \bigcup S_i$.

Tendremos:

$$\begin{array}{cc}
S_0 = \{(0,0)\} & S_1^1 = \{(2,1)\} \\
S_1 = \{(0,0),(2,1)\} & S_2^1 = \{(3,2),(5,3)\} \\
S_2 = \{(0,0),(2,1),(3,2),(5,3)\} & S_3^1 = \{(4,5),(6,6),(7,7),(9,8)\} \\
\multicolumn{2}{l}{S_3 = \{(0,0),(2,1),(3,2),(5,3),(4,5),(6,6),(7,7),(9,8)\}}
\end{array}$$

donde vemos que el par (5,3) se puede eliminar (no aparece en $f_3(x)$) pues corresponde a tener beneficio 3 llenando una capacidad 5 de la mochila, y tenemos el par (4,5) que corresponde a tener beneficio 5 con capacidad 4, y por el principio de optimalidad el par (5,3) no puede formar parte de una solución óptima pues no es óptimo para el problema con capacidad cinco (este par corresponde a un punto por debajo de la gráfica de la función). Los puntos (7,7) y (9,8) también se pueden eliminar por exceder la capacidad de la mochila.

El proceso de construcción de la solución es igual al visto en el caso anterior.

5.2.3 Algoritmo

Un esquema de algoritmo sería:

Algoritmo 5.1

```
PROCEDURE Mochila(p, w:ARRAY[1..n] OF REAL;M:REAL)
BEGIN
/*Construcción de los conjuntos que determinan las funciones*/
   S_0 = {(0,0)}
   FOR i = 1 TO n − 1
      S_i^1 = {(x + w_i, y + p_i) /(x, y) ∈ S_{i−1}}
      S_i = unir (S_{i−1}, S_i^1)
      S_i = eliminar (S_i)
   ENDFOR
/*Reconstrución de la solución*/
   FOR i = n DOWNTO 1
      par1 = ultimo (S_{i−1})
      par2 = ultimo1 (S_{i−1})
      IF mayor(par1, par2)
         x[i] = 0
      ELSE
         x[i] = 1
         M = actualizar(M)
      ENDIF
   ENDFOR
ENDMochila
```

donde *unir* y *eliminar* son los procedimientos descritos antes y *ultimo* y *ultimo1* obtienen el último par del conjunto y el último par, si lo hay, tal que $x + w_i$ no excede de M, y le suma (w_i, p_i); y *actualizar* obtiene la capacidad de la mochila en el nivel anterior (ver el árbol de búsqueda al principio de esta sección).

En la implementación de este algoritmo se presenta el problema de decidir la representación de los conjuntos de manera que las operaciones que se hacen sobre ellos (*unir*, *eliminar*, *ultimo* y *ultimo1*) se realicen de manera eficiente.

5.3 Problemas

Problema 5.1 Consideramos el problema de la devolución de una cantidad C con un número mínimo de monedas, teniendo monedas de n tipos distintos cuyo valor está almacenado en un array de índices 1 a n, y disponiendo de un número ilimitado de monedas de cada tipo. Para resolver este problema por programación dinámica se puede usar la fórmula de recurrencia:

$$M(X,i) = min_{k=0,\dots,\lfloor\frac{X}{a[i]}\rfloor}\{k + M(X - ka[i], i-1)\}$$

explicar cómo se rellenarían las dos tablas que se utilizan en la programación dinámica (la de cantidad de monedas y la que se utiliza en la reconstrucción de la solución) con el ejemplo: $c = (1, 2, 3)$ y $C = 7$. Encontrar una cota superior del tiempo de ejecución de este algoritmo.

Solución:

Las tablas quedarían:
la de número de monedas:

	0	1	2	3	4	5	6	7
1	0	1	2	3	4	5	6	7
2	0	1	1	2	2	3	3	4
3	0	1	1	1	2	2	2	3

y la que ayuda a recomponer la solución:

	0	1	2	3	4	5	6	7
1	0	1	2	3	4	5	6	7
2	0	0	1	1	2	2	3	3
3	0	0	0	1	0	1	2	1

Una estimación del tiempo en función de C y n será $C\sum_{i=1}^{n}\lfloor\frac{C}{a[i]}\rfloor \in O\left(C^2 n\right)$.

Problema 5.2 Indicar cómo se podría resolver por programación dinámica el problema de devolución de una cantidad con un número mínimo de monedas suponiendo que se dispone de n monedas cuyo valor se tiene en un array de índices de 1 a n. El problema difiere del anterior porque ahora no disponemos de un número ilimitado de monedas con cada valor.

Problema 5.3 Un político importante debe hacer un viaje urgente desde Santiago de Compostela a Alicante y decide hacerlo en avión. Tiene la oportunidad de volar de forma directa o haciendo una o más escalas. Los aeropuertos en los que podría hacer escala son Madrid, Barcelona y Sevilla. La siguiente tabla muestra los tiempos de vuelo entre ciudades:

	Santiago	*Madrid*	*Barcelona*	*Sevilla*	*Alicante*
Santiago		6	2	5	12
Madrid	6		2	2	3
Barcelona	2	2		5	4
Sevilla	5	2	5		4
Alicante	12	3	4	4	

Aplicar el método de programación dinámica para encontrar la ruta más corta para el problema de n ciudades sabiendo que los tiempos vienen dados en horas y que cada escala supone una hora adicional de retraso. Obtener una fórmula y resolver el problema particular.

Problema 5.4 Consideramos un laberinto formado por un damero donde en cada casilla un valor 1 indica que se puede pasar por esa casilla y un valor 0 que no se puede pasar por ella, y queremos resolver el problema de encontrar un camino de longitud mínima para llegar de una casilla origen a otra destino. Sólo pueden hacerse movimientos en horizontal y en vertical. Resolver el problema por programación dinámica. Explicar cómo funcionaría con el laberinto:

O	1	1	1	1
1	0	1	0	0
0	1	1	0	1
0	1	0	1	0
1	1	1	1	D

Solución:

Consideraremos el laberinto como un grafo donde cada casilla representa un nodo del grafo, siendo los nodos pares (x, y) con x representando la fila y y la columna, por lo que si el laberinto tiene n filas y m columnas tendremos $0 \leq x \leq n$ y $0 \leq y \leq m$. Las casillas origen y destino, que están señaladas con O y D en el laberinto ejemplo, son casillas por las que se puede pasar, por lo que consideraremos que su contenido en el laberinto (que llamaremos l) es 1. Además, el origen y destino pueden ser dos casillas cualesquiera que llamaremos (x_1, y_1) y (x_2, y_2). En este grafo todas las aristas tendrán peso 1, y habrá una arista entre dos nodos vecinos en vertical u horizontal si las casillas correspondientes tienen un 1:

$d((a, b), (a + 1, b)) = 1 \ si \ l(a, b) = l(a + 1, b) = 1$

$d((a, b), (a - 1, b)) = 1 \ si \ l(a, b) = l(a - 1, b) = 1$

$d((a, b), (a, b + 1)) = 1 \ si \ l(a, b) = l(a, b + 1) = 1$

$d((a, b), (a, b - 1)) = 1 \ si \ l(a, b) = l(a, b - 1) = 1$

$d((a, b), (c, d)) = +\infty \ en \ otro \ caso$

El problema se resuelve encontrando el camino de longitud mínima de (x_1, y_1) a (x_2, y_2), que se obtendrá resolviendo un problema menor consistente en llegar a una de las cuatro casillas vecinas a la (x_2, y_2) que tenga contenido 1:

$$\begin{aligned}C((x,y),(z,t)) = min\{ & C((x,y),(z-1,t)) + d((x,y),(z-1,t)),\\ & C((x,y),(z+1,t)) + d((x,y),(z+1,t)),\\ & C((x,y),(z,t-1)) + d((x,y),(z,t-1)),\\ & C((x,y),(z+1,t)) + d((x,y),(z+1,t))\}\end{aligned}$$

Por programación dinámica se resuelve construyendo dos tablas. En la primera de ellas, que llamaremos $t1$, se tienen los caminos mínimos desde el origen a los demás nodos, y las filas indicarían el número de pasos y las columnas las distintas casillas; y en la otra tabla, que llamamos $t2$, se almacena el nodo desde el que se ha obtenido ese camino mínimo, por lo que los valores de esta tabla pueden ser Iz, De, Ar y Ab. Dado que en un determinado número de pasos no se puede llegar a todos los nodos sino a los que distan del nodo origen menos de esa cantidad de pasos en vertical y horizontal en el laberinto, las tablas $t1$ y $t2$ las consideraremos del mismo tamaño que el laberinto.

Para ver la evolución de la ejecución lo haremos con el ejemplo. Se muestra a continuación el contenido de las dos tablas a lo largo de la ejecución:

0	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	∞	∞	∞

$-$	$-$	$-$	$-$	$-$
$-$	$-$	$-$	$-$	$-$
$-$	$-$	$-$	$-$	$-$
$-$	$-$	$-$	$-$	$-$
$-$	$-$	$-$	$-$	$-$

0	1	∞	∞	∞
1	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	∞	∞	∞

$-$	Iz	$-$	$-$	$-$
Ar	$-$	$-$	$-$	$-$
$-$	$-$	$-$	$-$	$-$
$-$	$-$	$-$	$-$	$-$
$-$	$-$	$-$	$-$	$-$

0	1	2	∞	∞
1	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	∞	∞	∞

$-$	Iz	Iz	$-$	$-$
Ar	$-$	$-$	$-$	$-$
$-$	$-$	$-$	$-$	$-$
$-$	$-$	$-$	$-$	$-$
$-$	$-$	$-$	$-$	$-$

0	1	2	3	∞
1	∞	3	∞	∞
∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	∞	∞	∞

–	Iz	Iz	Iz	–
Ar	–	Ar	–	–
–	–	–	–	–
–	–	–	–	–
–	–	–	–	–

0	1	2	3	4
1	∞	3	∞	∞
∞	∞	4	∞	∞
∞	∞	∞	∞	∞
∞	∞	∞	∞	∞

–	Iz	Iz	Iz	Iz
Ar	–	Ar	–	–
–	–	Ar	–	–
–	–	–	–	–
–	–	–	–	–

0	1	2	3	4
1	∞	3	∞	∞
∞	5	4	∞	∞
∞	∞	∞	∞	∞
∞	∞	∞	∞	∞

–	Iz	Iz	Iz	Iz
Ar	–	Ar	–	–
–	De	Ar	–	–
–	–	–	–	–
–	–	–	–	–

0	1	2	3	4
1	∞	3	∞	∞
∞	5	4	∞	∞
∞	6	∞	∞	∞
∞	∞	∞	∞	∞

–	Iz	Iz	Iz	Iz
Ar	–	Ar	–	–
–	De	Ar	–	–
–	Ar	–	–	–
–	–	–	–	–

0	1	2	3	4
1	∞	3	∞	∞
∞	5	4	∞	∞
∞	6	∞	∞	∞
∞	7	∞	∞	∞

–	Iz	Iz	Iz	Iz
Ar	–	Ar	–	–
–	De	Ar	–	–
–	Ar	–	–	–
–	Ar	–	–	–

0	1	2	3	4
1	∞	3	∞	∞
∞	5	4	∞	∞
∞	6	∞	∞	∞
8	7	8	∞	∞

–	Iz	Iz	Iz	Iz
Ar	–	Ar	–	–
–	De	Ar	–	–
–	Ar	–	–	–
De	Ar	Iz	–	–

0	1	2	3	4
1	∞	3	∞	∞
∞	5	4	∞	∞
∞	6	∞	∞	∞
8	7	8	9	∞

–	Iz	Iz	Iz	Iz
Ar	–	Ar	–	–
–	De	Ar	–	–
–	Ar	–	–	–
De	Ar	Iz	Iz	–

0	1	2	3	4
1	∞	3	∞	∞
∞	5	4	∞	∞
∞	6	∞	10	∞
8	7	8	9	10

$-$	Iz	Iz	Iz	Iz
Ar	$-$	Ar	$-$	$-$
$-$	De	Ar	$-$	$-$
$-$	Ar	$-$	Ab	$-$
De	Ar	Iz	Iz	Iz

Con lo que el camino de longitud mínima tiene longitud 10 y el camino se obtiene utilizando la tabla $t2$ empezando por el nodo (5,5):

$$(5,5),(4,5),(3,5),(2,5),(2,4),(2,3),(3,3),(3,2),(3,1),(2,1),(1,1)$$

Problema 5.5 Dada una serie de trabajos a, b, c, ... y una tabla:

	a	b	c	$\dots$
a	2	3	1	$\dots$
b	4	1	3	$\dots$
c	2	2	1	$\dots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	

donde la i-ésima fila, j-ésima columna, representa el beneficio que se obtiene de ejecutar el i-ésimo trabajo seguido del j-ésimo. Se quiere encontrar la sucesión de n trabajos que dé beneficio máximo. Idear un algoritmo por programación dinámica que resuelva el problema.

Solución:

Se almacenarán resultados parciales en una tabla con filas de 2 a n y columnas a, b, c, ... indicando la entrada $tabla[i,j]$ la solución óptima para sucesiones de i trabajos acabadas en el trabajo j. Con la tabla ejemplo y $n=4$ quedaría:

tabla

	a	b	c
2	4	3	3
3	7	7	6
4	11	10	10

tabla1

	a	b	c
2	b	a	b
3	b	a	b
4	b	a	b

Donde los datos de la tabla se obtienen con $tabla(2,x) = max_{k=a,b,c,\dots}\{B(k,x)\}$, y $tabla1(2,x) = k$ para el que se alcanza el máximo; y $tabla(i,x) = max_{k=a,b,c,\dots}\{tabla(i-1,k)+B(k,x)\}$, $tabla1(i,x) = k$ para el que se alcanza el máximo, donde $tabla(i-1,k)+B(k,x)$ corresponde al beneficio máximo con sucesiones de $i-1$ trabajos acabadas en k y $B(k,x)$ es el beneficio de realizar x a continuación de k.

El máximo beneficio del problema será $max_{k=a,b,c,\dots}\{tabla(n,k)\}$, y la sucesión de trabajos será $s_1, s_2, \dots, s_n$, donde s_n es el k para el que obtenemos el máximo y para $i = n, n-1, \dots, 2$ es $s_{i-1} = tabla1[i, s_i]$.

En el ejemplo:

$$
\begin{aligned}
s_4 &= a \\
s_3 &= tabla1[4, a] = b \\
s_2 &= tabla1[3, b] = a \\
s_1 &= tabla1[2, a] = b
\end{aligned}
$$

con lo que la solución óptima es *baba*.

Problema 5.6 Dada una tabla $n \times n$ de valores números naturales. Se pretende resolver el problema de obtener el camino de la casilla $(1, 1)$ a la (n, n) que minimice la suma de los valores en las casillas por las que se pasa, teniendo en cuenta que en cada casilla tenemos dos posibles movimientos: hacia la derecha y hacia abajo. Resolver el problema por programación dinámica, indicando cómo son las tablas que se usarían para llegar a la solución óptima y para recomponer el camino que da esa solución óptima, cómo sería la función de recurrencia que se usaría para completar esas tablas, y cuáles serían los valores de la función en los casos base. Aplicar el método a la tabla:

3	2	6
5	1	4
5	1	3

Solución:

Está claro que para llegar a una posición (i, j) por un camino óptimo hay que venir desde $(i, j-1)$ (desde arriba) o desde $(i-1, j)$ (desde la izquierda), habiendo llegado a esas casillas también por un camino óptimo. Así, la ecuación de recurrencia puede ser: $M(i, j) = min\ \{M(i, j-1) + T[i, j], M(i-1, j) + T[i, j]\}$, donde T representa la tabla de valores, y el par (i, j) representa la posición final del movimiento. Necesitaremos como caso base $M(1, 1) = T(1, 1)$.

Para resolver el problema por programación dinámica se utilizará una tabla $M : array[1..n, 1..n]$ y otra auxiliar $S : array[1..n, 1..n]$ que servirá para obtener el camino una vez que se ha calculado $M(n, n)$. $S[i, j]$ será 0 si se llega a (i, j) desde la izquierda y 1 si se llega desde arriba. El valor de $S[1, 1]$ será irrelevante.

Aplicando el método al ejemplo se rellenarían las tablas M y S por filas, de la fila 1 a la n, y dentro de cada fila por columnas de la 1 a la n, y quedaría:

M	1	2	3
1	3	5	11
2	8	6	10
3	13	7	10

S	1	2	3
1		0	0
2	1	1	0
3	1	1	0

Y para recomponer la solución sabemos que se acaba en $(3, 3)$ y $S[3, 3] = 0$, con lo que se llega desde $(3, 2)$; $S[3, 2] = 1$, y se llega desde $(2, 2)$; $S[2, 2] = 1$, y se llega desde

$(1,2)$; $S[1,2]=0$ y se llega desde $(1,1)$. Por tanto, se ha obtenido la solución óptima siguiendo el camino: $(1,1)$, $(1,2)$, $(2,2)$, $(3,2)$, $(3,3)$.

Problema 5.7 Resolver por Programación Dinámica el problema de minimizar el número de monedas a devolver para dar una cantidad C si tenemos monedas de n tipos, estando los tipos de las monedas en un array *tipos*: array$[1,\dots,n]$ of integer, y teniendo de cada tipo una cierta cantidad de monedas, estando estas cantidades almacenadas en un array *cantidad*: array$[1,\dots,n]$ of integer (de la moneda de tipo $tipos[i]$ podemos dar una cantidad entre 0 y $cantidad[i]$). No habrá que programar la resolución, pero sí habrá que dar la ecuación recurrente con la que se resuelve el problema, indicar qué tablas se utilizan y cómo se rellenan, cómo se recompone la solución, cuáles son y qué valores tienen los casos base, y estudiar el tiempo de ejecución.

Solución:

Llamamos $M(i,X)$ a la solución del problema de devolver una cantidad X con las i primeras monedas. El problema que queremos resolver es $M(n,C)$, y la ecuación de recurrencia:

$$M(i,X) = min_{k=0,1,\dots,min\left\{\left\lfloor\frac{X}{tipos[i]}\right\rfloor,cantidad[i]\right\}}\left\{M(i-1,X-k*tipos[i])+k\right\}$$

donde se indica que el número de monedas a dar de tipo i es como mucho el mínimo de la cantidad de monedas de ese tipo que hay y la máxima cantidad que se puede dar sin dar una cantidad mayor a X. Si se dan k monedas de tipo i, con las $i-1$ primeras monedas se hay que devolver una cantidad $X-k*tipos[i]$.

Para que la ecuación de recurrencia sea válida para todos los posibles valores tendremos como casos base: $M(i,X)=\infty$ si i es negativo, que indica que no existe solución al problema de devolver una cierta cantidad sin tener monedas de ningún tipo, lo mismo si X es negativo, pues tampoco se puede resolver el problema de devolver una cantidad negativa, y $M(i,0)=0$ que indica que para devolver una cantidad 0 la solución óptima es no dar ninguna moneda.

Se usan dos tablas M y s de n filas y C columnas. La tabla s se utiliza para recomponer la solución. Las tablas se rellenan de la primera a la última fila y dentro de cada fila de la primera a la última columna. En M se almacenan los valores de la función M obtenida con la ecuación de recurrencia, y en s se almacena el valor k con que se ha obtenido el valor mínimo de M. Cuando al hacer los cálculo se tenga que tomar uno de los casos base se pondrá el valor que hemos indicado anteriormente, pero no se accederá a la tabla pues los casos base no están almacenados en M.

El número de monedas a devolver lo encontramos en $M[n,C]$, y para obtener el número de monedas de cada tipo que contiene la solución hay que recorrer todas las filas de s desde la n a la 1:

for $k=n$ to 1
 $monedas[k]=s[k,C]$

$C = C - monedas[k] * tipos[k]$

endfor

Para estudiar el tiempo de ejecución hay que tener en cuenta que éste dependerá de varios parámetros, a saber: número de tipos de monedas (n), cantidad de monedas a devolver (C), valores de las monedas ($tipos$), y cantidad de monedas de cada tipo ($cantidad$). Debido a esta gran cantidad de parámetros, podemos estudiar sólo la cota superior y en función de un número más reducido de parámetros.

Si consideramos que el mínimo en la función recursiva se calcula variando k entre 0 y X (lo que obviamente es una cota superior muy pesimista) tendremos un coste:

$$\sum_{i=1}^{n}\left(a+\sum_{j=1}^{C}(b+j)\right) = an + bCn + \frac{C}{2}n + \frac{C^2}{2}n \in \theta\left(C^2 n\right)$$

con lo que el tiempo de ejecución es del orden $O\left(C^2 n\right)$.

Problema 5.8 Consideramos el problema del recorrido del caballo modificado donde se pretende llegar en un damero desde una casilla origen a otra destino en el menor número de pasos utilizando los movimientos del caballo de ajedrez y habiendo algunas casillas marcadas en el tablero por las que no se puede pasar (en la figura no se podría pasar por las casillas marcadas con una X).

	X		X	X
X		X	X	
X				X
	X	X	X	
X	X		X	

Indicar cómo se resolvería el problema por programación dinámica, detallando la fórmula de recursión a utilizar, cómo serán las tablas que se utilizan, cuáles son y qué valor tienen los casos base, cómo se rellenan las tablas y cómo se recompone la solución. Estudiar también el tiempo de ejecución. Explicar el funcionamiento del algoritmo con el ejemplo de la figura suponiendo que se parte de la casilla superior izquierda y se quiere llegar a la inferior derecha, y comparar la solución con la obtenida con un método de avance rápido (habrá que explicar el método de avance rápido que se usa).

Solución:

Se quiere minimizar el número de pasos a dar para llegar de una casilla origen (x_o, y_o) a otra destino (x_d, y_d) con un número máximo de pasos p. La solución se representará por $M(p, x_o, y_o, x_d, y_d)$. Dado que el mínimo camino con un máximo de p pasos para llegar de (x, y) a (z, t) puede coincidir con el mínimo camino entre ellos con $p-1$ pasos o se puede obtener dando un paso desde otra casilla intermedia hasta (z, t), y hay ocho casillas desde las que se puede llegar a (z, t), la fórmula de recursión puede ser $M(p,x,y,z,t) = min\{M(p-1,x,y,z,t), M(p-1,x,y,x-1,y-2)+1, M(p-1,x,y,x-1,y+2)+1, M(p-1,x,y,x+1,y-2)+1, M(p-1,x,y,x+1,y+2)+1, M(p-1,x,y,x-2,y-1)+1, M(p-1,x,y,x-2,y+1)+1, M(p-1,x,y,x+2,y-1)+1, M(p-1,x,y,x+2,y+1)+1\}$.

Para que la fórmula tenga sentido hay que determinar unos casos base que serán: $M(p,x,y,z,t) = \infty$ si alguno de los puntos está fuera del tablero, lo que ocurre si x, y, z o t son mayores que n, o si en el tablero en la posición (x,y) o (z,t) hay una X. Estos valores indican soluciones no posibles, con lo que cualquier solución posible los mejora. El otro caso base corresponde a ir de una casilla a sí misma, lo que se logra con cero pasos: $M(p,x,y,x,y) = 0$.

Dado que el número máximo de pasos que puede dar el caballo para llegar de la casilla origen a la destino es n^2-1, con n el tamaño del tablero si es cuadrado, y hay también n^2 casillas por las que puede pasar, y que en la fórmula de recurrencia el problema con un máximo de p pasos se obtiene en función de problemas con un máximo de $p-1$ pasos, y el problema de llegar a una casilla se pone en función de otras casillas anteriores en el tablero; habría que utilizar una tabla con n^2 filas, indicando cada fila el número de pasos, y con n^2 columnas, indicando cada columna una casilla:

	$(1,1)$	$(1,2)$	$\dots$	$(2,1)$	$\dots$	(n,n)
0						
1						
$\vdots$						
n^2-1						

El problema que se quiere resolver es $M\left(n^2-1, x_o, y_o, x_d, y_d\right)$, con lo que será un problema de la última fila y de la columna indicada por (x_d, y_d), determinándose los valores de la fila 0, que son casos base, por la casilla origen. Asociada a esta tabla habría otra de las mismas dimensiones para recomponer la solución, y apuntándose en esta tabla en cada casilla desde qué casilla se llega para obtener el valor mínimo obtenido en la ecuación de recursión. De esta manera el tiempo de ejecución sería del orden $\theta\left(n^4\right)$, pues hay que rellenar n^4 posiciones de la tabla y cada una de ellas se obtiene haciendo el mínimo de un número constante de valores. La solución se recompone utilizando la tabla auxiliar obteniendo en la fila n^2-1 el par que se encuentra en la columna (x_d, y_d), en la fila n^2-2 el par que se encuentra en la columna del valor obtenido en la fila n^2-1, y así sucesivamente hasta llegar a la fila 0 al nodo origen. El número de pasos para recomponer la solución es n^2, por lo que no aumenta el orden del algoritmo.

En este problema en concreto se podría hacer de una manera mejor pues mucha de la información almacenada en las tablas anteriores es innecesaria pues corresponde a nodos por los que no se puede pasar y por tanto con valor ∞, y además cuando se obtiene un número de pasos en una columna en las filas sucesivas no se puede disminuir ese número de pasos, lo que quiere decir que cuando pasamos por una casilla se pone en ella el número de pasos con que hemos llegado y no es necesario volver a evaluar la casilla. Del mismo modo, aunque el número máximo de pasos es $n^2 - 1$, cuando se llega a (x_d, y_d) en un determinado número de pasos se ha obtenido el camino mínimo, por lo que no es necesario completar todas las filas de la tabla.

Por tanto, se pueden usar dos tablas del mismo tamaño que el tablero, una para almacenar el número mínimo de pasos y otra para almacenar la casilla desde donde se ha llegado. Inicialmente tendremos la primera tabla con valores ∞ en casillas por las que no se puede pasar y con 0 en la origen, y la segunda tabla con un valor especial $(-1, -1)$ en la casilla origen:

0	∞		∞	∞
∞		∞	∞	
∞				∞
	∞	∞	∞	
∞	∞		∞	

$(-1,-1)$				

En el primer paso se aplica la fórmula de recursión a todas las casillas o, alternativamente y de forma menos costosa se obtienen las casillas con valor el paso por el que vamos y se pone valor uno más en las casillas a las que se puede acceder y que no tienen ningún valor. Con el ejemplo los pasos sucesivos serían:

0	∞		∞	∞
∞		∞	∞	
∞	1			∞
	∞	∞	∞	
∞	∞		∞	

$(-1,-1)$				
	$(1,1)$			

0	∞	2	∞	∞
∞		∞	∞	
∞	1			∞
	∞	∞	∞	
∞	∞	2	∞	

$(-1,-1)$		$(3,2)$		
	$(1,1)$			
		$(3,2)$		

0	∞	2	∞	∞
∞		∞	∞	
∞	1			∞
	∞	∞	∞	
∞	∞	2	∞	

$(-1,-1)$		$(3,2)$		
	$(1,1)$			
		$(3,2)$		

0	∞	2	∞	∞
∞		∞	∞	3
∞	1		3	∞
3	∞	∞	∞	3
∞	∞	2	∞	

$(-1,-1)$		$(3,2)$		
				$(1,3)$
	$(1,1)$		$(1,3)$	
$(5,3)$				$(5,3)$
		$(3,2)$		

0	∞	2	∞	∞
∞	4	∞	∞	3
∞	1	4	3	∞
3	∞	∞	∞	3
∞	∞	2	∞	4

$(-1,-1)$		$(3,2)$		
	$(3,4)$			$(1,3)$
	$(1,1)$	$(2,5)$	$(1,3)$	
$(5,3)$				$(5,3)$
		$(3,2)$		$(3,4)$

Después de este paso se para pues ya hemos llegado a la casilla destino. El número de pasos para llegar a la solución es de sólo cuatro, mucho menor que n^2. El camino seguido se obtiene accediendo a la casilla (5,5) en la segunda tabla, después a la (3,4), después a la (1,3), a la (3,2), a la (1,1), y ahí se acaba pues nos encontramos con el valor especial (-1,-1).

El coste de este algoritmo también tiene cota superior n^4 pues el número máximo de pasos es n^2 y en cada paso hay que analizar n^2 casillas. Pero el tiempo de ejecución será mucho menor que con el esquema anterior en el que se usa una tabla con n^4 entradas.

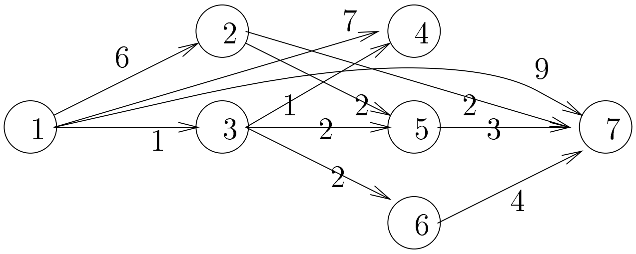
Se puede acabar la ejecución sin encontrar camino si es que este no existe. Esto ocurrirá cuando en un paso no se actualiza el contenido de ninguna casilla.

Un método de avance rápido no nos va a asegurar que resolvamos el problema, pero podemos seleccionar la casilla destino desde una dada como aquella a la que se puede llegar a más casillas todavía no recorridas y no marcadas con X. De esta manera intentamos evitar llegar a casillas desde las que no se puede salir. El recorrido en el ejemplo sería:

0	X	4	X	X
X	8	X	X	5
X	1	6	3	X
7	X	X	X	
X	X	2	X	

Vemos que no se llega a solución pues desde 8 no se puede ir a ninguna casilla, y sin embargo habíamos estado muy cerca de la solución en la casilla marcada con 3. El método de avance rápido podría mejorarse, pero de cualquier manera no nos asegura que obtengamos un camino ni que en caso de obtenerlo sea mínimo.

Problema 5.9 Consideramos el problema de un grafo multietapa modificado donde puede haber aristas de un nodo a todos los nodos de las etapas siguientes, según el ejemplo que se muestra en la figura. Se pretende resolver por programación dinámica el problema de encontrar el camino mínimo del nodo origen al destino (en el ejemplo del nodo 1 al 7).



a) Obtener la fórmula que nos permita resolver el problema por programación dinámica e indicar cuáles son los casos base y cuáles son sus valores.
b) Indicar cómo se irán rellenando las tablas y cómo se recomponen las soluciones.
c) Explicar el funcionamiento con el grafo ejemplo.
d) Estudiar la ocupación de memoria.
e) Estudiar el tiempo de ejecución.

Solución:

a) Llamaremos $C(p, D)$ al valor del camino mínimo con p pasos, origen 1 y destino D. El valor que queremos obtener es $C(niveles - 1, n)$, con $niveles$ el número de niveles del grafo y n el número de nodos. En el ejemplo será $C(3, 7)$.

La longitud de los caminos mínimos de p pasos se obtendrá en función de las longitudes mínimas de caminos de $p-1$ pasos añadiendo un paso, que se dará por una arista que una directamente dos nodos. Las aristas las consideramos almacenadas en una matriz d, siendo $d[i, j]$ la longitud de la arista de origen i y destino j. Si no existe tal arista el valor $d[i, j]$ será infinito. Para que la fórmula de recurrencia que vamos a utilizar funcione en todos los casos consideraremos $d[i, i] = 0$.

La fórmula será $C(p, D) = min_{k=1,\ldots,n}\{C(p-1, k) + d(k, D)\}$.

Los casos base serán $C(0, k) = 0$.

b) Podemos considerar que tenemos una tabla para las longitudes de los caminos, C:array[0,..,$niveles$ − 1;1,..,n], correspondiendo la fila 0 a los casos base, por lo que esta fila estará inicializada a 0. El resto de las filas se van completando con un par de bucles:

```
for i = 1 to niveles − 1
    for j = 1 to n
        calcular el mínimo y ponerlo en C[i, j]
    endfor
endfor
```

La última fila no es necesario calcularla entera, sino que podría calcularse sólo $C[niveles - 1, n]$, que es el valor en el que estamos interesados.

Se tendrá otra tabla Aux:array[1,..,$niveles-1$;1,..,n] donde se almacena el valor de k para el que se ha obtenido el mínimo. Esta tabla servirá para recomponer el camino seguido hasta obtener el valor mínimo.

Para obtener el camino, lo almacenaremos en un array $camino$:array[1..$niveles$], donde obviamente $camino[1] = 1$ y $camino[niveles] = n$. Inicialmente $destino = n$, y el nuevo destino se obtiene consultando como $destino = Aux[niveles-1, destino]$, y después $destino = Aux[niveles-2, destino]$, etc. Por tanto, necesitaremos un bucle con índices de $niveles - 1$ hasta 1:

$camino[niveles] = n$
for $i = niveles - 1$ to 1
  $camino[i] = Aux[i, camino[i+1]]$
endfor

Si en el camino encontramos nodos repetidos se pasa una única vez por ellos.

c) Los valores de las tablas para este grafo serán:

La tabla C:

niv, nod	1	2	3	4	5	6	7
1	0	6	1	7	∞	∞	9
2	0	6	1	2	3	3	8
3	0	6	1	2	3	3	6

Y la tabla Aux:

niv, nod	1	2	3	4	5	6	7
1	1	1	1	1	1	1	1
2	1	1	1	3	3	3	2
3	1	1	1	3	3	3	5

El camino será $camino[4] = 7$, $camino[3] = Aux[3,7] = 5$, $camino[2] = Aux[2,5] = 3$, y $camino[1] = Aux[1,3] = 1$.

d) Según el método explicado la ocupación de memoria será la que ocupan las tablas y el *camino* pues además de esta memoria sólo se utiliza alguna para variables auxiliares. La ocupación será $niveles \times n$, de la tabla C, $niveles \times n$, de Aux, y $niveles$ de $camino$; y el total es $niveles(2n+1)$.

e) El tiempo de ejecución vendrá dado por:

$$\sum_{i=1}^{niveles-1} \sum_{j=1}^{n} \sum_{k=1}^{n} 1 + \sum_{i=niveles-1}^{1} 1$$

donde los tres primeros sumatorios corresponden a la obtención de las posiciones de las tablas obteniendo mínimos y el último a la formación del camino. Hemos puesto el valor uno dentro de los bucles porque estamos interesados en el orden. El orden será $(niveles-1)n^2 + niveles - 1$

Observación adicional:

El problema se puede resolver con menos tiempo de ejecución y ocupación de memoria si tenemos en cuenta que en cada paso tenemos las distancias definitivas de nodos de un nuevo nivel, por lo que no hay que volver a calcularlas, y que para calcular las distancias de un nodo estas se obtienen haciendo el mínimo no de todos los nodos sino de nodos que están en niveles anteriores al suyo.

Así, inicialmente tendríamos:

C							Aux						
0	6	1	7	∞	∞	9	1	1	1	1	1	1	1

Con lo que ya tenemos las distancias a nodos del nivel uno y dos. Después se calculan las distancias a nodos del nivel tres en adelante utilizando las mismas tablas:

C							Aux						
0	6	1	2	3	3	8	1	1	1	3	3	3	2

Y finalmente las distancias a nodos del nivel cuatro en adelante:

C							Aux						
0	6	1	2	3	3	6	1	1	1	3	3	3	5

De este modo vemos que la ocupación de memoria es proporcional a n.

Algo más complicado es obtener el tiempo de ejecución. Haremos una aproximación llamando $l = niveles$. Suponiendo que tenemos el mismo número de nodos en cada nivel salvo en el primero y el último, este número será $\frac{n-2}{l-2}$.

La primera vez que se rellena la tabla se hace en un tiempo n pues sólo hay que poner las distancias de 1 a cada nodo. La última vez se calcula un mínimo en función de $n-1$ nodos, por lo que el tiempo es $n-1$.

El resto de las veces ($i = 3$ hasta $l-1$) se calculan $\frac{n-2}{l-2}$ mínimos que es el número de nodos en ese nivel, y cada mínimo se calcula en función de los nodos de niveles anteriores, por lo que será de $1 + \frac{n-2}{l-2}(i-2)$.

El tiempo promedio será:

$$n + \sum_{i=3}^{l-1} \frac{n-2}{l-2} \left(1 + \frac{n-2}{l-2}(i-2)\right) + n - 1 \in O(nl)$$

Capítulo 6

BACKTRACKING

6.1 Descripción

El **backtracking** y el **branch and bound** son métodos para recorrer de una manera sistemática un árbol de soluciones. El backtracking se caracteriza por hacer una búsqueda en profundidad en un árbol en el que algunos nodos (o todos) representan soluciones del problema.

Ejemplo 6.1

Para resolver el problema de obtener los subconjuntos del conjunto de números $\{13, 11, 7\}$ cuya suma es 20 se puede representar una solución como:

- Una secuencia de tres ceros o unos, indicando un 0 en la posición i que el i-ésimo número no está en el conjunto y un 1 que sí está. Cada una de las posibles combinaciones representa una solución distinta.

- Una secuencia de tres valores entre 0 y 3, indicando un valor j, con $1 \leq j \leq 3$ que el número j-ésimo forma parte del conjunto, y un 0 indica que no se toma ningún número. De esta manera una solución viene dada por una secuencia de tres números siendo estos distintos si son distintos de cero. Además, a partir del primer cero en la secuencia todos los valores siguientes son cero. Como lo que interesa es determinar los números en el conjunto pero no el orden dentro de éste, dos secuencias con los mismos números cambiados de orden representan la misma solución (2,3,0 y 3,2,0), por lo que podemos considerar que los números en la secuencia están ordenados de menor a mayor hasta llegar al primer cero.

De esta manera, el árbol de soluciones sería, con la primera representación de las soluciones el de la figura 6.1, y con la segunda el de la figura 6.2.

Analizando estos dos árboles podemos obtener las características principales de un algoritmo por bactracking.

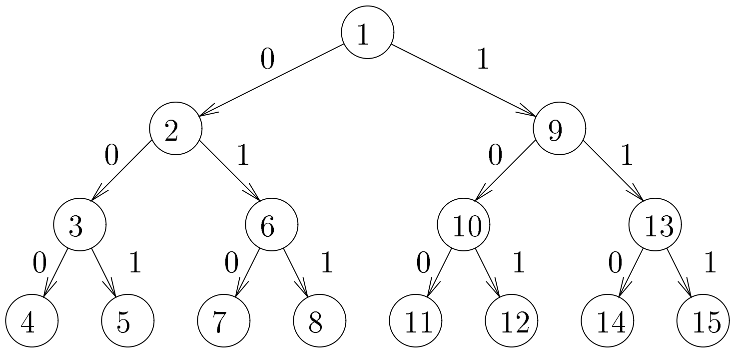


Figura 6.1: Árbol de búsqueda con soluciones representadas como secuencias de 0 y 1

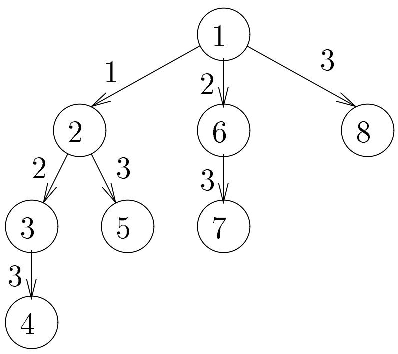


Figura 6.2: Árbol de búsqueda con soluciones representadas como secuencias de números entre 0 y 3

- En los dos árboles se representan los nodos con un número en su interior, representando este número el orden en que se genera ese nodo en el recorrido del árbol de soluciones. Vemos que el recorrido es un **recorrido sistemático**, siendo en los dos casos **en profundidad** y generando en cada nodo sus descendientes de izquierda a derecha. En el backtracking el recorrido se hace en profundidad, pero el orden en que se generan los nodos hijos de un nodo dado puede variar (por ejemplo, se podrían generar los hijos de derecha a izquierda). Este recorrido sistemático se hará generando los posibles valores de las soluciones de una manera sistemática. Esta generación será la que proporcionará el recorrido del árbol, que hay que tener en cuenta que es un árbol lógico, no físico, y que por lo tanto no está almacenado en memoria. Por esto, hay que tener una **función de generación de valores solución** que dependerá del nodo por el que vayamos (de la solución total o parcial generada hasta ese momento). En el primer caso los valores a generar por esta función serán 0 y 1 en este orden, y en el segundo caso serán del 1 al 3, pero empezando en el siguiente al último valor en la solución.

- La representación de las soluciones determina la forma del árbol: la cantidad de los descendientes de un nodo, la profundidad del árbol, si el árbol es completo o no, ... Y determina la cantidad de nodos del árbol y por tanto posiblemente la eficiencia del algoritmo, pues el tiempo de ejecución dependerá del número de nodos que se generen.

- Tendremos tantos niveles como valores tenga una secuencia solución. En el segundo árbol en algunos casos no llegamos hasta el nivel tres, pero podemos asumir que los nodos que no están en el nivel tres completan la solución con ceros.

- En un nodo hay que poder deteminar si es solución del problema (o posible solución), lo que se hará con una **función de determinación de si un nodo es solución**. En el primer caso sólo son nodos solución los terminales (en los que la suma de los números sea 20), pero en el segundo caso todos los nodos (en los que la suma sea 20) son solución.

- Cuando como consecuencia de haber recorrido todos los descendientes de un nodo se vuelve a ese nodo, hay que generar un hermano de ese nodo si lo hay, y si no lo hay se vuelve al nivel superior. Por tanto, necesitaremos una **función para determinar si un nodo tiene hermanos que todavía no se han generado**.

- Por último, podemos ver que en los dos árboles hemos representado todos los posibles nodos de acuerdo con la representación de las soluciones escogida, pero vemos que a partir de algunos nodos (el nodo 13 en el primer caso y el 3 en el segundo caso) puede no haber solución (en este ejemplo porque la suma de la solución parcial que llevamos sea mayor que 20). Es conveniente tener una **función que determine si a partir de un nodo se puede llegar a una**

solución, de manera que utilizando esta función se puede evitar el recorrido de algunos nodos y por lo tanto reducir el tiempo de ejecución.

Con este ejemplo hemos analizado las funciones que debe tener un esquema para un algoritmo por backtracking. En la subsección siguiente analizaremos un esquema que utiliza estas funciones.

Otro factor a tener en cuenta es el tipo de problema que estamos queriendo resolver. En el problema del ejemplo hay solución, y ésta es única, con lo que (si esto lo sabemos a priori) el programa se ejecutaría hasta encontrar la única solución. Pero se pueden presentar otros casos:

- Hay alguna solución y sólo se quiere encontrar una. En este caso el programa parará cuando encuentre la primera.
- Puede no haber solución pero si la hay sólo queremos una. El programa parará cuando encuentre una solución o cuando vuelva al nodo raíz, en cuyo caso no hay solución.
- Se quieren todas las soluciones. El programa acaba cuando vuelve al nodo raíz, y conforme ha ido encontrado soluciones las ha ido almacenando. En este caso también puede que no hubiera ninguna solución.
- El problema es de optimización y queremos optimizar una función sujeta a unas restricciones (el conjunto de restricciones puede ser vacío). En este caso cuando se encuentra una solución parcialmente óptima (óptima entre las que se han encontrado hasta ese momento cumpliendo las restricciones) se almacena la solución y su valor, de modo que al final tengamos la solución óptima, o ninguna solución si ninguna de las posibles cumple las restricciones. De este modo, el programa acabará después de haber analizado todas las posibilidades o, lo que es lo mismo, cuando vuelva al nodo raíz.

6.1.1 Esquema

Un esquema utilizando las funciones antes enumeradas y en el caso en que hay soluciones y sólo se quiere obtener una podría ser el siguiente:

Algoritmo 6.1 *Esquema del* **backtracking** *cuando hay solución y sólo se quiere encontrar una.*

PROCEDURE Backtracking(n:INTEGER)
VAR
$nivel$:INTEGER
fin:BOOLEAN

```
BEGIN
   nivel = 1
   fin=FALSE
   s[1] = generar(nivel)
   REPEAT
      IF solucion(nivel)
         fin =TRUE
      ELSIF criterio(nivel)
         nivel = nivel + 1
         s[nivel] = generar(nivel)
      ELSIF mas_hermanos(nivel)
         s[nivel] = generar(nivel)
      ELSE
         WHILE NOT mas_hermanos(nivel)
            retroceder(nivel)
         ENDWHILE
         s[nivel] = generar(nivel)
      ENDIF
   UNTIL fin
ENDBacktracking
```

donde:

- La variable *nivel* indica el nivel por el que se va recorriendo el árbol.
- La variable fin indicará si se ha acabado ya el recorrido del árbol. En este caso se pone a TRUE cuando se encuentra una solución, pero en problemas de otro tipo se pone a TRUE en diferentes casos, tal como ya habíamos señalado.
- Con la función *generar* se generará el siguiente hijo de un nodo dado, que puede ser el primer hijo u otro hijo distinto si volvemos al nodo después de un retroceso.
- La función *solucion* devuelve TRUE si el nodo por el que vamos es solución del problema. En caso de querer encontrar varias soluciones habría que almacenar la solución en otra secuencia y seguir buscando soluciones desde la secuencia actual. En el caso de un problema de optimización devolverá TRUE cuando cumple las restricciones, después de esto comparará con la solución óptima almacenada hasta ese momento y se quedará con la mejor de las dos y seguirá buscando por la secuencia actual.
- La función *criterio* devuelve TRUE si a partir de un nodo se puede llegar a una solución. Puede que no podamos asegurar que se llegue, pero que no tengamos ningún motivo para descartarlo.

- La función *mas_hermanos* devolverá TRUE si hay hermanos de ese nodo que todavía no han sido generados.
- Con la función *retroceder* se retrocede en el árbol de soluciones. Habrá que disminuir en uno el valor de *nivel*, y posiblemente actualizar la solución para dejarla en un valor inicial.

En el algoritmo se hace un REPEAT hasta que se encuentra una solución (la condición de fin puede ser otra, tal como hemos dicho), y se supone que la solución se almacena en el array global s. El recorrido del árbol (¡lógico!) se hace poniendo distintos valores en la solución. Cuando se genera un nuevo nodo se comprueba si es solución, en cuyo caso se acaba la ejecución, si no es solución se comprueba si se puede excluir que a partir de él se puedan encontrar soluciones, si no se puede excluir se pasa al siguiente nivel y se genera un nuevo nodo, si sí se puede excluir se comprueba si hay nodos hermanos suyos todavía no generados, si hay alguno se genera uno de ellos, y si no hay se pasa al nivel anterior hasta encontrar una posición donde se puedan generar nuevos nodos.

Basándose en este esquema, y con pequeñas modificaciones, se pueden diseñar esquemas para los problemas de los tipos antes enumerados.

6.2 Problema de las reinas

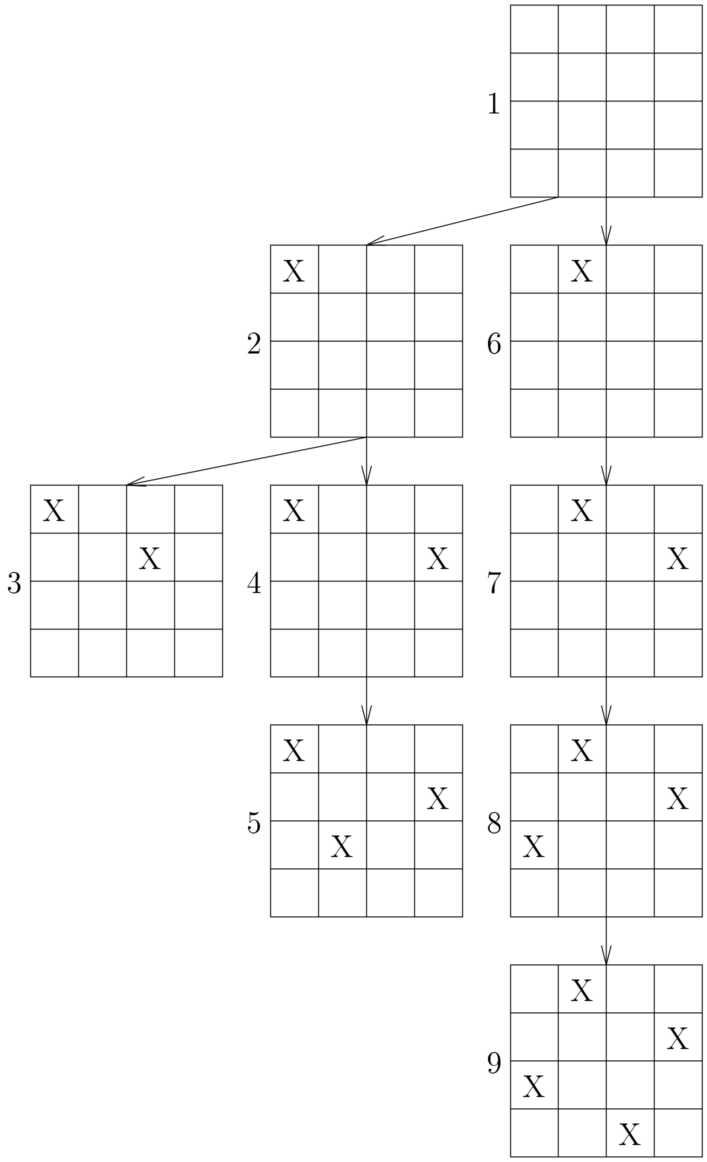
6.2.1 Planteamiento

El problema de las n reinas consiste el encontrar las posiciones (todas o una) de n reinas en un tablero ajedrezado de tamaño $n \times n$, sin que una reina pueda comerse a otra.

El problema se puede resolver tomando una secuencia de n decisiones sobre en qué columna situar la reina que está situada en cada una de las filas. No puede haber dos reinas en la misma fila, por lo que una solución s consistirá de una secuencia de valores de 1 a n, indicando $s[i] = j$ que la reina en la fila i ocupa la columna j. Además de la restricción de no poder estar en la misma fila o columna tenemos la restricción de que no puede haber dos reinas en la misma diagonal o antidiagonal (debido a que las reinas mueven y comen en horizontal, vertical y diagonal).

El problema se puede resolver recorriendo un árbol de posibles soluciones utilizando la técnica del backtracking. De esta manera, como hay que tomar n decisiones, el árbol tiene profundidad n y el coste de su recorrido es prohibitivo, pues el número de nodos del árbol es $1+n+n(n-1)+n(n-1)(n-2)+\ldots+n!$. De todas maneras, cuando no es posible una solución por un método más eficiente hay que recurrir a estos métodos de alto coste e intentar reducir al máximo el número de nodos que se recorren y el coste de la generación y recorrido (análisis) de cada nodo.

Representamos el árbol de búsqueda de la solución con $n = 4$:



En la figura cada tablero representa un nodo del árbol de búsqueda; al lado de cada nodo hay un número que indica el orden en que se recorren los nodos; en cada tablero las posiciones de las reinas se representan con una X; y el nodo raíz corresponde a un tablero vacío.

Observamos que, aunque el número de nodos total del árbol de búsqueda es $1+4+4^2+4^3+4^4=341$, el número de nodos que se recorre para encontrar una solución es 9. Vemos, por tanto, que teniendo en cuenta las características del problema se puede reducir el espacio de búsqueda haciendo el programa más eficiente:

- Si se quieren encontrar todas las posibles configuraciones, podemos considerar que hay configuraciones simétricas, con lo que en la primera fila sólo es necesario situar reinas en las columnas 1 y 2, pero no en las 3 y 4, pues por cada solución que encontremos con la primera reina en la columna 1 ó 2 tendremos una simétrica con la primera reina en la columna 4 ó 3. De esta manera el espacio de búsqueda se divide por dos, siendo el número total de nodos posibles 171.

- Utilizando una función *criterio* apropiada se puede determinar nodos desde los que no se llega a una solución (por ejemplo, de los 4 posibles hijos del nodo 2, en los dos primeros las reinas están en la misma fila o la misma diagonal, con lo que se violan las restricciones) por lo que no es necesario generar esos nodos. De este modo llegamos a que de los 171 nodos sólo es necesario generar 9, aunque esto se hace a costa de un mayor coste computacional en cada nodo, coste que corresponde principalmente a la ejecución de la función *criterio*. Por tanto, hay que encontrar funciones *criterio* con bajo coste computacional de manera que no sea mayor el coste añadido en cada nodo que lo que ahorramos al recorrer menos nodos.

- En el backtracking se hace un recorrido sistemático, pero si tuvieramos algún criterio para decidir en qué orden recorrer los nodos (no siempre de izquierda a derecha) de manera que esa decisión nos acerque a una solución, puede que necesitemos recorrer menos nodos. En el ejemplo, si generamos el nodo 6 antes que el dos llegamos a una solución recorriendo 5 nodos y no 9. Esta idea se utilizará en los métodos Branch and Bound que veremos en el capítulo siguiente.

6.2.2 Algoritmo

Analizaremos un algoritmo que corresponde a una pequeña modificación del esquema presentado en el algoritmo 6.1. Se pretende buscar una única solución. En el algoritmo representamos qué partes corresponden a cada una de las funciones del esquema del backtracking.

Algoritmo 6.2 *Problema de las n reinas.*

PROCEDURE reinas
VAR
  $nivel$:INTEGER
  fin:BOOLEAN
BEGIN
  $nivel = 1$
  $s[1] = 0$
(*es una generación que no corresponde a generar ninguna posible solución pero que posibilita que sumando 1 a $s[1]$ se vayan generando las distintas posibilidades*)
  fin=FALSE

REPEAT
$s[nivel] = s[nivel] + 1$ (*generar*)
WHILE $s[nivel] \leq n$ AND NOT $correcto(nivel)$

(*la comprobación $s[nivel] \leq n$ corresponde a la función *mas_hermanos*, y la función *correcto* a la *criterio*. Por tanto, se evalúa si un nodo no cumple el criterio y tiene hermanos no generados, y se saldrá del WHILE porque cumpla el criterio, con lo que se puede descender de nivel, o porque no haya más hermanos, con lo que hay que subir de nivel*)

$s[nivel] = s[nivel] + 1$ (*generar*)
ENDWHILE
IF $s[nivel] \leq n$

(*si se ha salido del WHILE porque la configuración es correcta*)

IF $nivel = n$

(*si tenemos una solución. La función *solucion* será TRUE cuando estemos en la última fila con una configuración correcta, o lo que es lo mismo en nodos terminales del árbol*)

fin=TRUE
ELSE

(*la configuración es correcta pero todavía no hemos llegado a una solución*)

$nivel = nivel + 1$
$s[nivel] = 0$

(*no generamos una columna pero inicializamos para que en la primera generación empiece por la columna 1*)

ENDIF
ELSE

(*no hemos encontrado ningún nodo de entre todos los hermanos que hemos tratado en el WHILE en el que la configuración sea correcta, por lo que hay que subir de nivel*)

$s[nivel] = 0$
$nivel = nivel - 1$

(*la función *retroceder* consiste en ascender de nivel habiendo dejado el valor de la solución en el nivel actual a su valor inicial, pues se deshace la decisión que se había tomado*)

ENDIF
UNTIL fin
END reinas

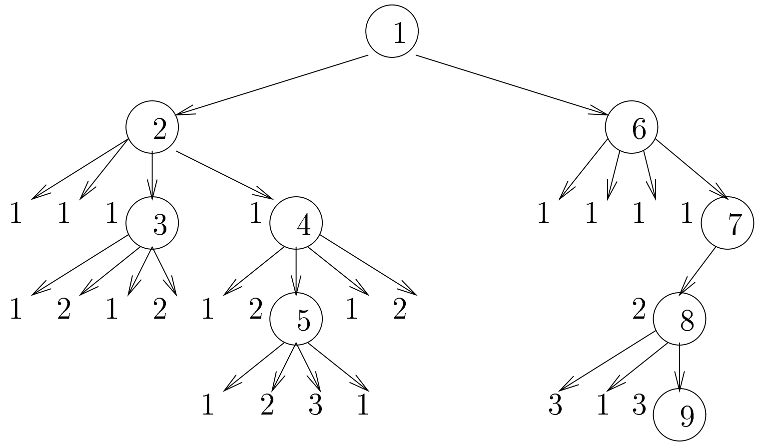
PROCEDURE correcto($nivel : 0 \dots n$):BOOLEAN
VAR
i:INTEGER
$valor$:BOOLEAN
BEGIN

$i = 1$
valor=TRUE
WHILE $i < nivel$ AND *valor*
IF $s[nivel] = s[i]$ OR $|s[i] - s[nivel]| = |i - nivel|$
valor=FALSE
ENDIF
$i = i + 1$
ENDWHILE
RETURN *valor*
END correcto

En el procedimiento *correcto* se comprueba, para cada una de las reinas en las filas anteriores a la que se quiere colocar (desde $i = 1$ y mientras $i < nivel$) si está en la misma columna que la que queremos colocar ($s[nivel] = s[i]$) o si está en la misma diagonal ($s[i] - s[nivel] = i - nivel$) o en la misma antidiagonal ($s[i] - s[nivel] = -i + nivel$). Obviamente, esta función *correcto* puede ser programada de manera más eficiente, pues una cota superior de su coste es el nivel por el que vamos, pero si se utilizan arrays de booleanos indicando las columnas, diagonales y antidiagonales ocupadas, el coste sería constante, pues sólo habría que comprobar que la columna, diagonal y antidiagonal donde estamos introduciendo la reina actual estén libres. Esta mejora en la función **criterio** conlleva una modificación de las funciones de generación y de retroceso pues en cada caso hay que actualizar los tres arrays de booleanos.

6.2.3 Evaluación de la eficiencia

En el algoritmo 6.2 se observa que la función *criterio* en cada nodo no tiene un coste constante, sino que el coste es proporcional al nivel por el que vamos, con lo que una cota superior más realista del tiempo de ejecución se obtiene multiplicando el número de nodos en cada nivel por el nivel en el que están menos 1. De este modo una cota superior del coste sería $1 + n + n(n-1) + 2n(n-1)(n-2) + \ldots + (n-1)n!$. En el caso de $n = 4$ vemos en la siguiente figura que aunque se han generado 9 nodos el coste de la generación de cada uno de ellos y de la comprobación de que un descendiente no cumple el criterio no es constante.



En esta figura los nodos representan los tableros válidos y las flechas representan intentos de poner una reina, cuando la flecha llega a un nodo el intento ha tenido éxito y cuando no llega a un nodo no ha tenido éxito. Al lado de cada flecha o nodo hay un número que indica la cantidad de veces que se ha tenido que pasar por el cuerpo del WHILE en el procedimiemto *correcto*. De esta forma, aunque se recorren 9 nodos, se ha pasado 36 veces por el cuerpo del WHILE, con lo que podemos considerar que el coste es proporcional a 45. De todas maneras, aunque el utilizar una función *criterio* de coste no constante nos aumenta el costo, seguimos estando lejos de los 171 nodos que habría que generar si no se utiliza un criterio de eliminación de nodos.

Además de tener en cuenta el coste de la función criterio para evaluar el algoritmo hay que estimar también el número de nodos que se generan. En la estimación del costo de la función criterio se puede normalmente encontrar una cota de este costo, pero suele ser mucho más difícil hacer una estimación del número de nodos generados. Se puede hacer una estimación tomando una expresión que puede ser una posible solución y ver cuántos nodos se generan con esta expresión. Tomando más expresiones se puede hacer una media.

Por ejemplo, en el problema con 4 reinas podemos considerar que se intentan generar todos los descendientes de un nodo, con lo que el número máximo de nodos es 341. Si generamos la secuencia (2,3,4,1) llegamos hasta el nivel 2 y se generan en media 1+4+16=21 nodos, si generamos la secuencia (1,4,2,3) obtenemos una solución, se llega al nivel 4 y se generan en media 341 nodos, y si generamos la secuencia (2,4,3,1) llegamos al nivel 3 y se generan 1+4+16+64=85 nodos. Haciendo la media de los nodos generados con estas estimaciones y dividiendo por el número de nodos totales obtenemos el valor 0.43, con lo que la estimación que hemos hecho indica que en la ejecución del algoritmo se recorren menos de la mitad de los nodos posibles.

6.3 Problema de la mochila

Volvemos a considerar el problema de la mochila 0/1:

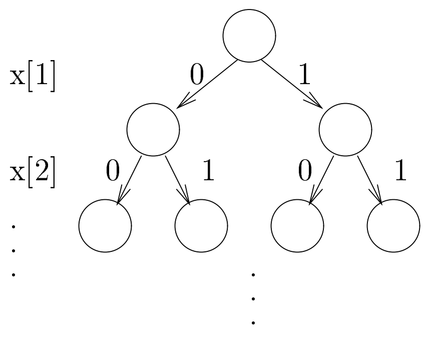
$$
\begin{array}{ll}
\textit{maximizar} & \sum_{i=1}^{n} p_i x_i \\
\\
\textit{sujeto a} & \sum_{i=1}^{n} w_i x_i \leq M \\
 & x_i = 0, 1 \ \forall i = 1, 2, \ldots, n
\end{array}
$$

donde M es la capacidad de la mochila, cada objeto se mete entero o no se mete ($x_i = 0, 1$), p_i y w_i son los beneficios y los pesos, respectivamente, de los objetos.

Para obtener un algoritmo por backtracking para resolver este problema lo haremos viendo cuáles serían las funciones *solucion*, *criterio*, *mas_hermanos*, *generar* y *retroceder* del esquema dado en el algoritmo 6.1, cómo se representan las soluciones y cuál es la condición de fin.

- La condición de fin en este caso es que la variable *nivel* vuelva a valer 0, pues en ese momento hemos vuelto al nodo raíz e intentamos encontrar un hermano suyo, lo que quiere decir que hemos recorrido todo el árbol. Al ser un problema de optimización habrá que obtener todas las posibles soluciones (las que cumplen las restricciones) y tener almacenada una solución óptima parcial que se va actualizando al encontrar una nueva solución con mayor beneficio.

- Una posible representación de las soluciones la obtenemos de manera obvia del planteamiento del problema. Se pueden representar en un array de 1 a n con valores 0 ó 1, indicando que no se mete un objeto o que sí se mete. Este modo de representar las soluciones corresponde a un árbol de búsqueda de la forma:

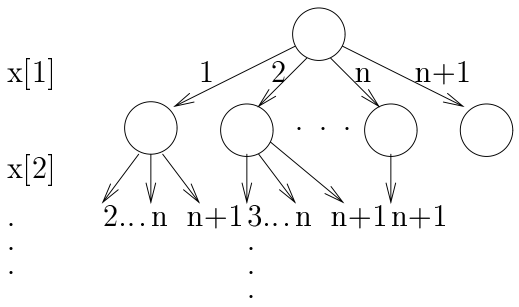


Pero hay otras representaciones posibles. Podemos considerar que una solución viene dada por un array x de 1 a n y valores entre 1 y $n+1$, indicando $x[i] = j$

que el objeto j se mete en la mochila, y $x[i] = n + 1$ que no se mete un objeto (sólo tenemos n objetos). Los valores también podrían ser de 0 a n, indicando un 0 que no se mete ningún objeto, pero la representación que hemos elegido puede ser mejor si la función *generar* la implementamos sumándole 1 al valor actual de $x[i]$, de modo que lo último que se intentará con los objetos es no meterlos en la mochila. Como no puede haber objetos repetidos y además no importa el orden en que se introduzcan los objetos, y una vez que se decide no introducir objetos no se sigue intentando introducir más, el array donde se representan las soluciones debe cumplir:

$$x[i] \leq x[i+1] \ \forall i = 1, \ldots, n-1$$
$$x[i] = x[i+1] \Leftrightarrow x[i] = n+1$$

Esta representación de las soluciones corresponde a un árbol de la forma:



donde podríamos no considerar los nodos correspondientes al valor $n+1$ porque a partir de intentar poner el valor n lo único que queda por hacer en el array solución es poner todos los demás valores a $n+1$.

- La función *solucion* sería:
  - Con la primera representación, son nodos posible solución sólo los terminales, por tanto es:

    RETURN (nodo terminal y *criterio*)
  - Y con la segunda representación todos los nodos representan una posible solución (sin más que completar el resto del array con el valor $n+1$), por lo que la función *solucion* es simplemente:

    RETURN *criterio*

- Una posible función *criterio* devolverá TRUE si: los pesos acumulados hasta ese momento no exceden la capacidad de la mochila y, una vez obtenida una solución, el beneficio máximo alcanzable a partir de ese nodo puede llegar a ser mayor que el beneficio de la solución óptima actual.

  Los pesos acumulados se obtienen con la fórmula:

$$\sum_{i=1}^{nivel} w_i x_i \tag{6.1}$$

  con la primera representación de las soluciones, y con la segunda:

$$\sum_{i=1}^{nivel} w_{x_i} \tag{6.2}$$

  Estos pesos acumulados se obtienen en cada nodo en un tiempo constante utilizando el valor del peso acumulado hasta el nodo padre.

  La otra cuestión es ¿cómo obtener una cota superior del beneficio obtenible a partir de un nodo? Podemos considerar que el problema de la mochila 0/1 es un caso particular del no 0/1, por lo que una solución del problema 0/1 no puede ser mejor que la óptima del no 0/1. Como tenemos un método de avance rápido para resolver el problema no 0/1 de manera óptima, utilizamos esta técnica en cada nodo para obtener la cota superior.

- La función *mas_hermanos* es:

  RETURN $x[nivel] < 1$ ó RETURN $x[nivel] < n + 1$

  dependiendo del árbol.

- La función *generar* puede ser en los dos casos $x[nivel] = x[nivel] + 1$, pero inicializando $x[i]$ a -1 en el primer árbol al pasar al primer descendiente de un nodo, e inicializando $x[i] = x[i-1] + 1$ en el segundo árbol.

- La función *retroceder* será hacer $nivel = nivel - 1$. Se podría poner la solución (x) a su valor inicial, pero si consideramos que se inicializa al generar el primer hijo de un nodo (tal como hemos indicado en la función *generar*) no haría falta poner al valor inicial.

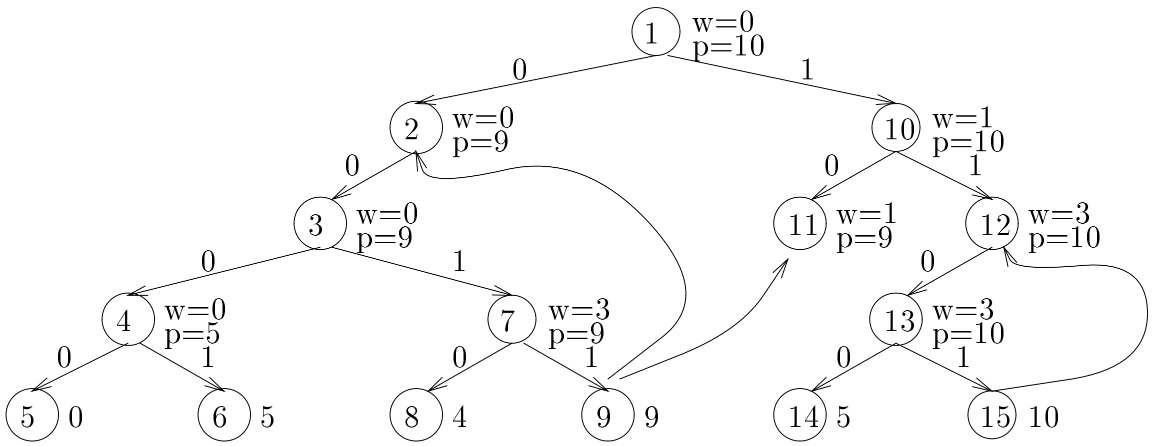
Ejemplo 6.2

Utilizamos el ejemplo con valores $n = 4$, $M = 7$, $p = (2, 3, 4, 5)$ y $w = (1, 2, 3, 4)$ para ver cómo influyen los distintos factores del esquema del backtracking en el número de nodos que se generan y por lo tanto en el tiempo de ejecución. Los objetos están ordenados de mayor a menor $\frac{p}{w}$, por lo que se puede aplicar el método de avance rápido en el orden en el que están para obtener la cota superior del beneficio en cada nodo tal como hemos dicho.

Si no se poda el árbol de búsqueda, con la primera representación habría que generar 31 nodos, y con la segunda 32 ó 16, según que consideremos que se generan los nodos etiquetados con $n+1$ o que no se generan (el coste de su generación es muy pequeño en comparación con el coste de los demás nodos).

Si con el primer árbol se utiliza como función *criterio* que los pesos acumulados no excedan de la capacidad de la mochila no se poda ningún nodo, pues todos los que podrían podarse son terminales, con lo que el número de nodos generados sigue siendo 31.

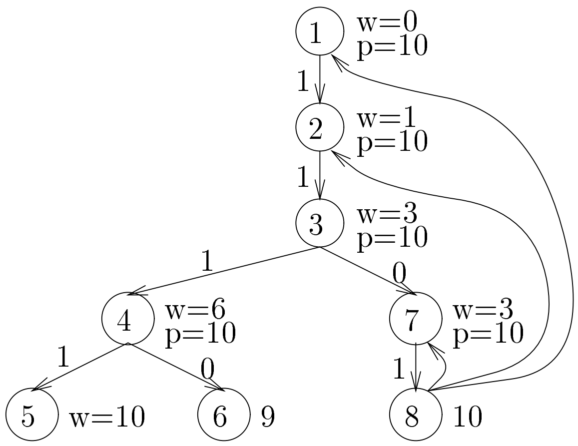
Si utilizamos el *criterio* completo el árbol de búsqueda quedaría:



donde los números dentro de los nodos indican el orden en que se recorren, los valores de w y p al lado de los nodos indican, respectivamente, el peso acumulado y la cota superior del beneficio máximo que se puede alcanzar a partir de ese nodo utilizando el método de avance rápido solucionando el problema como no 0/1, y quedándonos con la parte entera ya que en nuestro problema los números son enteros. Una flecha hacia arriba indica cuando un nodo se poda cuál es la solución que produce la poda. Aunque hemos almacenado el peso acumulado en cada nodo (w), no se ha podado ningún nodo por esta razón. Los nodos terminales son soluciones por lo que sólo aparece al lado de ellos el valor de la solución que representan. La solución óptima la tenemos en el nodo 15. El nodo 11 se poda la primera vez que pasamos por él, pues en ese momento ya tenemos una solución (el nodo 9) con beneficio 9, y desde el nodo 11 no se puede obtener beneficio mayor que 9. Pero los nodos 2 y 12 no se podan la primera vez que pasamos por ellos sino al volver a ellos haciendo retroceso. En el nodo 2 la primera vez que pasamos no se ha generado todavía ninguna solución, pero la segunda vez se ha generado una solución con beneficio 9. Lo mismo ocurre con el nodo 12, que la segunda vez que pasamos por él se ha generado ya el nodo 15 con beneficio 10.

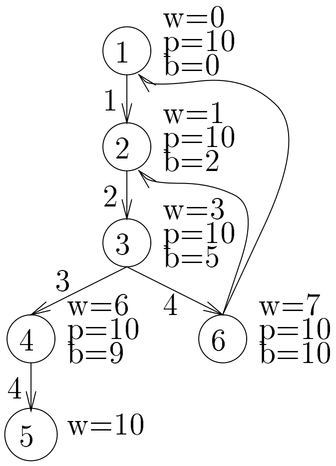
Se han generado un total de 15 nodos, mientras que antes generábamos 31. Vemos, por tanto, que el uso de una buena función *criterio* nos puede reducir considerablemente el número de nodos generados. Pero ¿qué pasa con el tiempo de ejecución? El tiempo de análisis de cada nodo ha aumentado al utilizar esta función *criterio*, por lo que no sabemos cuál de los dos métodos va a ser más rápido. En general es preferible podar la mayor cantidad de nodos posibles si el problema es grande, ya que el número de nodos crece exponencialmente con el tamaño del problema, mientras que el coste de la función *criterio* suele crecer de una manera mucho más moderada.

Todavía podemos reducir más el número de nodos si tenemos en cuenta que la solución del problema debe estar cerca de la solución del problema no 0/1. En el problema no 0/1 se empieza introduciendo elementos en la mochila, mientras que generando los valores del array solución en el orden 0,1 estamos empezando no introduciendo los objetos en la mochila. Puede ser preferible organizar el árbol con función *generar* en la que se reste uno al valor de la solución en vez de sumar 1, e inicializando a 2 en vez de a -1. De este modo el árbol quedaría:



donde hemos pasado de 15 a 8 nodos cambiando el orden de generación de las soluciones, de manera que el encontrar antes la solución óptima nos produce podas más productivas. Aquí vemos que el nodo terminal número 5 no es solución porque su peso acumulado sobrepasa la capacidad de la mochila.

Para llegar a una solución tenemos que llegar hasta un nodo terminal, pero con la otra representación de las soluciones que hemos visto todos los nodos representan una solución, con lo que podemos empezar a podar antes. Utilizando el otro árbol de soluciones se generan 6 nodos en vez de 8. En este caso al lado de cada nodo hay tres valores: el peso acumulado, el beneficio máximo alcanzable, y el beneficio de la solución que representa el nodo.



6.4 Problemas

Problema 6.1 Representamos un laberinto por una matriz cuadrada $M(i,j)$ con $1 \leq i \leq n$ y $1 \leq j \leq n$, donde cada componente puede tomar el valor A o C, indicando una A que la posición está abierta y una C que está cerrada. Por una casilla abierta se puede pasar y por una cerrada no. Desarrollar un algoritmo que genere todos los caminos que empezando por $M(1,1)$ acaben en $M(n,n)$. Hay sólo dos posibles movimientos: hacia abajo y hacia la derecha. Representar el árbol de búsqueda correspondiente a:

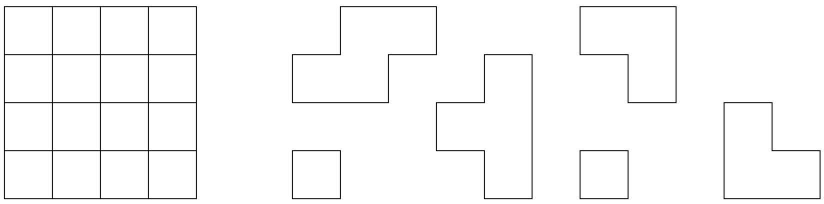
A	A	A	C	C
A	C	A	A	A
A	C	A	C	A
A	C	A	A	A
A	A	A	C	A

Problema 6.2 Diseñar un programa para, dado un conjunto de números enteros positivos $\{x_1, x_2, \ldots, x_n\}$, obtener todos los subconjuntos que sumen otra cantidad dada C.

Problema 6.3 Diseñar un programa que encuentre los n enteros positivos $x_1, x_2, \ldots, x_n$ que, dado otro entero positivo N, minimicen $\sum_{i=1}^{n} x_i^2$ y cumplan que $\sum_{i=1}^{n} x_i = N$.

Problema 6.4 Si tenemos n números enteros $x_1, x_2, \ldots, x_n$ y otro entero N, hacer un algoritmo que encuentre un subconjunto $\{y_1, y_2, \ldots, y_m\}$ de $\{x_1, x_2, \ldots, x_n\}$ que minimice el valor $|N - \sum_{i=1}^{m} y_i|$.

Problema 6.5 Se quiere hacer un programa por Backtracking que resuelva un rompecabezas consistente en rellenar una plantilla cuadriculada con unas ciertas piezas (dibujo). Explicar cómo se podría representar una solución, cómo sería el árbol de soluciones, cuál sería la condición de final, y cómo serían los procedimientos "generar", "criterio", "solucion" y "mas_hermanos".



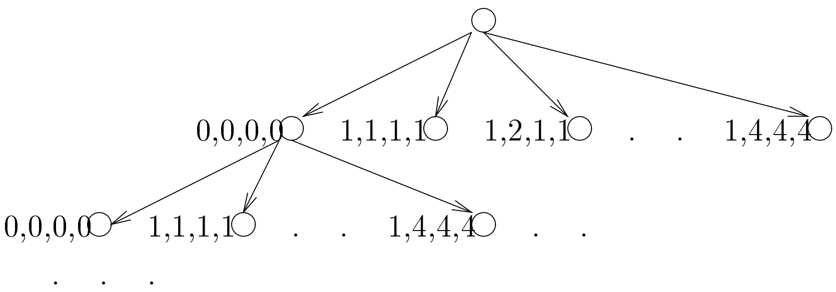
Solución:

El árbol de soluciones podría ser de profundidad 6 (en general n si n es el número de piezas de que disponemos) y en cada nivel se podría decidir la posición en que se pone una pieza (en el nivel 1 la pieza 1, en el nivel 2 la 2, ...). Puede ocurrir que no todas las piezas intervengan en la construcción del puzzle, en cuyo caso en cada nivel se incluiría un nodo más que representara esta posibilidad.

Cada pieza se puede intentar poner en una de las 16 casillas del damero y en una de las cuatro posibles posiciones de giro, lo que nos dará 64 posibilidades. Si consideramos que todas las piezas intervienen en la solución cada nodo no terminal tendrá 64 hijos, y si consideramos que hay piezas que pueden no intervenir 65 hijos.

Una solución vendrá dada por un array con 6 registros, indicando el registro i-ésimo cómo se ha colocado la pieza i-ésima. Por tanto, los registros tendrán cuatro campos: el primero indicará si se ha colocado o no, el segundo la posición de giro, y el tercero y cuarto la posición de colocación en el tablero (esta posición tiene que estar referida a un cuadrado distinguido que deberá tener cada pieza).

Así, un árbol sería, suponiendo que consideramos que pueda no incluirse una pieza:



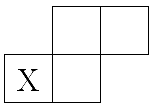
Tendremos un contador que indique por qué nivel vamos y otro que indicará por cuál de los 65 nodos. Le llamaremos *nivel* al primero y *nodo* al segundo.

nodo estará inicializado a -1, y la función *generar* lo que hará será sumarle 1 y calcular a partir del valor de *nodo* a qué nodo del árbol corresponde, y calcular la representación en forma de cuádrupla, que es lo que necesitamos para poder almacenar la solución:

$generar(nivel, nodo)$:
  $nodo = nodo + 1$
  si $nodo = 0$
    $solucion[nivel] = (0, 0, 0, 0)$
  en otro caso
    $solucion[nivel].campo1 = 1$
    $solucion[nivel].campo2 = (nodo - 1)\ mod\ 4 + 1$
    $solucion[nivel].campo3 = (nodo - 1)\ DIV\ 16 + 1$
    $solucion[nivel].campo4 = (nodo - 1)\ DIV\ 4 + 1$
  finsi

mas_hermanos comprobará si hay más hermanos que generar, por lo que será TRUE si $nodo < 65$.

La función *criterio* determinará si a partir de un nodo se puede llegar a solución. Para esto necesitamos saber si la pieza que estamos intentando encajar se sale del tablero o se pone encima de alguna de las puestas. Necesitamos un array 4×4 ($n \times n$) que represente la situación del tablero, indicando un 0 que no hay nada en esa posición y un 1 que sí hay. Si una pieza la representamos con un número indicando el número de cuadrados de la pieza distintos del cuadrado distinguido, y pares de números indicando el desplazamiento de los cuadros respecto al distinguido, la representación de



sería $\{3, (1, 0), (1, 1), (2, 1)\}$.

Según estas consideraciones, para comprobar si se cumple el criterio habría que hacer:

  $nuevapieza = girar(solucion[nivel].campo2, pieza)$
  $criterio = TRUE$
  para $i = 0$ hasta $numero\ de\ cuadrados$
    si NOT $encaja(i, pieza)$
      $criterio = FALSE$
    finsi
  finpara

donde *girar* recibirá el número de giro a hacer y la representación de la pieza y dará la representación con ese giro, y *encaja* comprobará si el cuadrado i-ésimo de la pieza encaja en el tablero (no se sale de él ni se solapa con una posición ocupada).

solucion nos diría si estamos en un nodo solución, y esto ocurre si $nivel = 6$ y se cumple el criterio.

La condición de fin será cuando *solucion* devuelva TRUE, si consideramos que con las piezas que nos dan se va a poder contruir el rompecabezas, o también que *nivel* sea 0 si no tiene porqué poderse construir el rompecabezas.

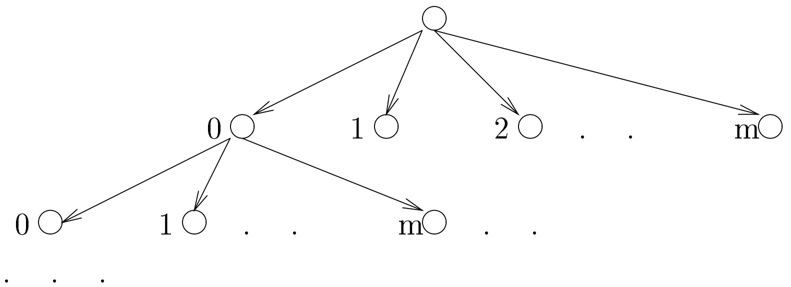
Problema 6.6 En una tabla con n filas y m columnas, con $m \leq n$, se representan las posibilidades de que unos determinados trabajadores realicen unos trabajos. Un 0 en la fila i columna j representa que el trabajador i-ésimo no puede realizar el trabajo j-ésimo, y un 1 en dicha posición representa que sí lo puede realizar. Diseñar un algoritmo por backtracking que resuelva el problema de asignación de trabajos a los trabajadores teniendo en cuenta que a cada trabajador se le puede asignar un trabajo o ninguno, y que cada trabajo se asigna a un único trabajador.

Solución:

La solución se almacenará en un array $solucion[1..n]$ de valores entre 0 y m, siendo $solucion[i]$ el trabajo asignado al trabajador i-ésimo, si es 0 no se ha asignado ningún trabajo a ese trabajador, por lo que inicialmente *solucion* tiene todos sus valores a 0, o a -1 para que la primera vez que se aplique la función *generar* se genere $solucion[i] = 0$.

Se puede utilizar un array $asignado[1..m]$ de booleanos para indicar si un trabajo está asignado o no.

El árbol de soluciones será:



donde en el nivel i-ésimo decidimos el trabajo a realizar por el trabajador i-ésimo.

Las funciones del esquema del backtracking serán:

$generar(solucion, nivel)$:

devolver $solucion[nivel] + 1$

para lo que necesitaremos (para empezar por 0) que *solucion* esté inicializado a -1.

$mas\_hermanos(nivel)$:

devolver $solucion[nivel] < m$

pues si toma el valor m el nodo no tiene más hermanos y no se pueden generar más hijos del nodo padre del actual.

$criterio(solucion, nivel)$:

devolver $tabla[nivel, solucion[nivel]] = 1$ y $asignado[solucion[nivel]] = FALSE$

(si no utilizáramos *asignado* tendríamos que recorrer el array *solucion* hasta el nivel actual para ver si se ha asignado el trabajo).

$solucion(solucion, nivel)$:

devolver $correcto(solucion, nivel)$ y $nivel = n$

La condición de fin en este caso sería que una variable booleana fin sea TRUE (tomará este valor cuando $solucion(solucion, nivel) = TRUE$) o $nivel = 0$ (en cuyo caso no habría solución para este problema).

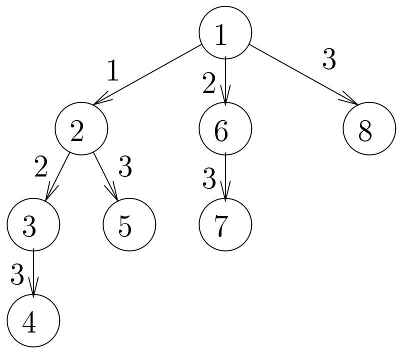
Problema 6.7 Dado un conjunto de pares de números naturales $S = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$, resolver por backtracking el problema de obtener de entre todos los subconjuntos (no vacíos) de S el de menor diferencia en valor absoluto entre lo que suman las primeras componentes y lo que suman las segundas. Es decir, si un subconjunto es $\{(u_1, v_1), (u_2, v_2), \ldots, (u_m, v_m)\}$, hay que minimizar $|\sum_{i=1}^{m} u_i - \sum_{i=1}^{m} v_i|$.

Se utilizará un esquema similar al no recursivo visto en clase, con funciones "generar", "mas_hermanos", "solucion" y "criterio", y se indicará cómo se representan las soluciones y cuál es la condición de fin.

Solución:

Supondremos que el conjunto de pares se representa por medio de dos arrays x e y con índices de 1 a n, siendo $x[i] = x_i$ y $y[i] = y_i$.

La solución se va a representar con un array s con índices entre 1 y n. Este array se considera inicializado a cero, y al final de la ejecución un valor distinto de cero, $s[i] = j$, indicará que se ha tomado el par j-ésimo. Dado que la solución no se representa con un array de valores binarios (0 ó 1), el árbol de búsqueda será de la forma



y en el nivel uno se decidirá cuál es el primer par de la solución, en el dos cuál es el segundo, ...

El esquema del backtraking visto en clase es:

PROCEDURE Backtracking(n:INTEGER)

VAR

$nivel$:INTEGER

```
    fin:BOOLEAN
  BEGIN
    nivel = 1
    fin=FALSE
    s[1] = generar(nivel)
    REPEAT
      IF solucion(nivel)
        fin =TRUE
      ELSIF criterio(nivel)
        nivel = nivel + 1
        s[nivel] = generar(nivel)
      ELSIF mas_hermanos(nivel)
        s[nivel] = generar(nivel)
      ELSE
        WHILE NOT mas_hermanos(nivel)
          retroceder(nivel)
        ENDWHILE
        s[nivel] = generar(nivel)
      ENDIF
    UNTIL fin
  ENDBacktracking
```

pero este esquema corresponde al caso de querer encontrar una solución sabiendo que hay alguna, por lo que tenemos que modificarlo para que se resuelva nuestro problema de optimización utilizando el árbol de búsqueda que utilizamos.

Dado que en nuestro árbol todos los nodos son posible solución, la función *solucion* simplemente devolvería true, y cuando se encuentra una solución no se para la búsqueda, sino que se compara con una solución óptima actual s_{opt} y si la nueva solución es mejor que s_{opt} se actualiza s_{opt}. Hay un único caso en que se puede parar la búsqueda, que es cuando la diferencia entre la suma de las primeras componentes y las segundas vale cero.

La función *citerio* nos indica si a partir del nodo que vamos explorando podemos llegar a una solución. Como todos los nodos son posibles solución, todos los nodos cumplirán el criterio salvo los terminales. De este modo:

$criterio(nivel)$:

  return $s[nivel] \neq n$

La función *mas_hermanos* devuelve true si el nodo tiene más hermanos a la derecha, y esto ocurre en el tipo de árbol que hemos escogido si el nodo no es terminal, por lo que esta función será igual que la *criterio*.

En cada nodo tendremos calculado el *valor* de la diferencia entre los x y los y incluidos, por lo que al movernos de nodo (*generar* y *retroceder*) se varía ese *valor*.

La función *generar* genera el siguiente nodo a explorar. Cuando se genera el primer hijo de un nodo se hace con $s[nivel] = s[nivel-1]+1$, pero cuando no es el primer hijo se hace con $s[nivel] = s[nivel] + 1$. Por tanto, debemos tener alguna manera de saber si estamos generando el primer hijo o no. Si se ha inicializado s a 0, tendremos que si $s[nivel] = 0$ vamos a generar el primer hijo, y si es distinto de cero no es el primer hijo. Además de la inicialización de s a cero es necesario que al retroceder se vuelva a dejar $s[nivel] = 0$, pues a partir de ahí se empezará por el primer hijo de otro nodo. El código de $generar$ puede ser:

$generar(nivel)$:
  si $s[nivel] = 0$
    $s[nivel] = s[nivel - 1] + 1$
  en otro caso
    $valor = valor - x[s[nivel]] + y[s[nivel]]$
    $s[nivel] = s[nivel] + 1$
  $valor = valor + x[s[nivel]] - y[s[nivel]]$

Para que esta función sirva para todos los niveles es necesario tener $s[0] = 0$, por lo que consideraremos s con índices entre 0 y n.

Antes de retroceder hay que actualizar $valor$ quitando la asignación hecha:

$retroceder(nivel)$:
  $valor = valor - x[s[nivel]] + y[s[nivel]]$
  $s[nivel] = 0$
  $nivel = nivel - 1$

Como estamos en un problema de optimización hay que recorrer todo el árbol, por lo que la condición de fin será que se vuelva al nodo raíz o que, como ya hemos dicho, se obtenga una solución con valor 0 que no se puede mejorar.

Con estas consideraciones podemos modificar el esquema general para obtener uno adecuado para nuestro problema:

PROCEDURE Backtracking(n:INTEGER)
VAR
  $nivel$:INTEGER
  fin:BOOLEAN
BEGIN
  $nivel = 1$
  fin=FALSE
  $s \leftarrow 0$
  $s_{opt} \leftarrow 0$
  $valor_{opt} = \infty$
  $valor = 0$
  $s[1] = generar(nivel)$
  REPEAT
    IF $solucion(nivel)$ /*Que devuelve true, por lo que este IF se puede quitar*/
      IF $mejor(s,s_{opt})$

```
                copiar(s,s_opt)
                valor_opt = valor
                IF valor_opt = 0
                    fin =TRUE
                ENDIF
            ENDIF
        ENDIF
        IF criterio(nivel)
```

/*Ya que *solucion* es siempre true hay que comprobar siempre si se cumple el criterio*/

```
            nivel = nivel + 1
            s[nivel] = generar(nivel)
        ELSIF mas_hermanos(nivel)
            s[nivel] = generar(nivel)
        ELSE
            WHILE nivel ≠ 0 AND NOT mas_hermanos(nivel)
                retroceder(nivel)
            ENDWHILE
            IF nivel ≠ 0
                s[nivel] = generar(nivel)
            ENDIF
        ENDIF
    UNTIL fin OR nivel = 0
ENDBacktracking
```

Donde *mejor* compararía los valores absolutos de s y s_{opt} y devuelve true si es menor el de s, y *copiar* copiaría el primer parámetro en el segundo.

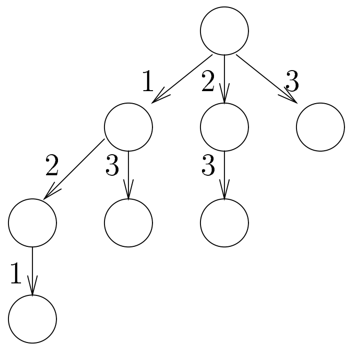
Problema 6.8 Supongamos que hay n hombres y n mujeres, y que tenemos dos matrices P y Q de dimensiones $n \times n$, donde $P[i,j]$ indica el grado de aversión que siente el hombre i por la mujer j, y $Q[i,j]$ el grado de aversión que siente la mujer i por el hombre j. Programar un algoritmo por backtracking que encuentre un emparejamiento entre los n hombres y las n mujeres de forma que la suma del producto de las aversiones sea mínimo. Nota: el grado de aversión es un valor mayor o igual a 0.

Problema 6.9 Dado un conjunto de números naturales $X = \{x_1, x_2, \ldots, x_n\}$ y otros dos números naturales N_1 y N_2, hacer un programa por backtracking que encuentre todos los subconjuntos de X cuya suma S sea $N_1 \leq S \leq N_2$. Programar las funciones "generar", "mas_hermanos", "solucion" y "criterio", e indicar cómo se representan las soluciones, cómo es el árbol de soluciones y cuál es la condición de final.

Problema 6.10 Dado un conjunto de números enteros positivos ($\{x_1, x_2, \ldots, x_n\}$) y otro número entero positivo N, resolver por backtracking, utilizando alguno de los esquemas no recursivos vistos en clase, el problema de obtener un subconjunto ($\{y_1, y_2, \ldots, y_m\}$) de dicho conjunto de números que minimice el valor $D = N - y_1 * y_2 * \ldots * y_m$, siendo $D \geq 0$. Dar el esquema que habría que utilizar, indicar cuál es la condición de fin, cómo es el árbol de búsqueda y cómo se representan las soluciones, y cómo son las funciones "generar", "mas_hijos", "criterio" y "solucion".

Solución:

Consideraremos un árbol del tipo:



donde $n = 3$. El árbol tendrá n niveles, y en cada nivel se decidirá el número a incluir en el conjunto.

La solución se almacenará en un array $s : array[1, \ldots, n]$ de valores entre 0 y n, indicando un 0 que no se añade ningún número y $s[i] = j$, con $j \neq 0$ que se incluye en el conjunto el número x_j. Además, será $a[i] < a[j]$ si $i < j$ y $a[j] \neq 0$, y si $a[j] = 0$ será $a[i] = 0$ $\forall i > j$. s estará inicializado a 0.

Como es un problema de optimización se acabará cuando se recorra todo el árbol, lo que ocurrirá cuando se regrese al nodo raíz. Pero, por las condiciones del problema, también se puede acabar cuando $D = 0$, en cuyo caso no se puede mejorar la solución. Además, tendremos un array $SOA : array[1, \ldots, n]$ donde se almacenará la solución óptima actual que estará inicializado a 0, indicando el conjunto vacío; y una variable VOA donde se almacenará el valor de la solución óptima actual, que estará inicializado a N. Cada vez que se llegue a un nodo que represente una solución (que son todos los nodos del árbol que cumplan la restricción del problema) se comprobará si se actualizan SOA y VOA.

Utilizaremos otra variable auxiliar (*producto*) donde se almacenará el producto de los números tomados hasta ese momento, y se inicializará a 1. Esta variable habrá que actualizarla cada vez que cambiemos de un nodo a otro en el árbol.

Las funciones del esquema del backtracking serán:

Como todos los nodos son posible solución sólo es necesario comprobar si cumplen la restricción:

solucion:

return $N - producto \geq 0$

Un nodo cumplirá el criterio si cumple la misma restricción y no es un nodo terminal:

criterio:

  return $N - producto \geq 0$ y $s[nivel] < n$

donde $nivel$ es la variable que indica el nivel por el que vamos.

Un nodo tiene más hermanos si no es terminal:

mas_hermanos:

  return $s[nivel] < n$

La generación de un nodo es distinta si es el primer hijo de otro nodo que si no es el primero. Para que el caso de generar el primer hijo de un nodo sea general necesitamos que el array s tenga índices desde 0 y que $s[0] = 0$:

```
generar:
    si s[nivel] = 0
        s[nivel] = s[nivel - 1] + 1
        producto = producto * x_{s[nivel]}
    en otro caso
        producto = producto/x_{s[nivel]}
        s[nivel] = s[nivel] + 1
        producto = producto * x_{s[nivel]}
    finsi
```

Y el esquema del algoritmo quedaría:

```
nivel = 1
SOA ← 0
VOA = N
s ← 0
producto = 1
generar
repetir
    si solucion
        si N - producto < VOA
            VOA = N - producto
            SOA ← s
        finsi
    finsi
    si criterio
        nivel = nivel + 1
        generar
    en otro caso si mas_hermanos
        generar
    en otro caso
        mientras no mas_hermanos y nivel ≠ 0
```

$producto = producto/x_{s[nivel]}$
$s[nivel] = 0$
$nivel = nivel - 1$
finmientras
si $nivel \neq 0$
generar
finsi
finsi
hasta $nivel = 0$ o $VOA = 0$

Problema 6.11 Resolver el problema del grafo multietapa por medio de un backtracking con un esquema similar al visto en clase: decir cómo sería el árbol, cómo se representan las soluciones, cuál es la condición de fin, cómo serían las funciones "generar", "solucion", "mas_hermanos" y "criterio", y dar el esquema completo para resolver el problema.

Solución:

Suponiendo la representación del grafo por matriz de adyacencia el árbol de soluciones será un árbol n-ario, aunque cada nodo tendrá un máximo de v hijos válidos, y los nodos de nivel $m-1$ un máximo de uno.

Como es un problema de optimización se acabará cuando se vuelva al nodo raíz, en cuyo caso se habrá recorrido todo el árbol.

La solución se almacenará en un array s inicializado a 0.

Consideraremos en el esquema que ∞ representa un número suficientemente grande que se puede sumar y restar sin problemas. Esto lo hacemos así para evitar el tratar de manera distinta los pesos que corresponden a aristas del grafo y el peso ∞ que corresponde a que no exista la arista.

El esquema del algoritmo será:

$nivel = 1$
$s[i] = 1$
$SOA \leftarrow 0$ (*array donde se almacena la solución óptima actual*)
$s \leftarrow 0$
$VOA = \infty$ (*valor de la solución óptima actual*)
$valor = 0$ (*contendrá el valor de la solución que estamos formando*)
repeat
if solucion($nivel$)
if $valor < VOA$
$SOA \leftarrow s$
$VOA = valor$
endif
endif
if criterio($nivel$)
$nivel = nivel + 1$

$s[nivel]$ =generar($nivel$)
else if mashermanos($nivel$)
$valor = valor - t[s[nivel-1], s[nivel]]$
$s[nivel]$ =generar($nivel$)
else
while not mashermanos($nivel$) and $nivel \neq 1$
$valor = valor - t[s[nivel-1], s[nivel]]$
$s[nivel] = 0$
$nivel = nivel - 1$
endwhile
if $nivel \neq 1$
$s[nivel]$ =generar($nivel$)
endif
endif
until $nivel = 1$

donde las funciones son:
solucion($nivel$):
return $nivel = m$
(*hemos llegado al último nivel*)
criterio($nivel$):
return $valor < VOA$
(*no hemos sobrepasado el VOA y podemos mejorar la solución óptima actual*)
generar($nivel$):
$valor = valor + t[s[nivel-1], s[nivel]+1]$
return $s[nivel]+1$
(*genera los hijos independientemente de que exista la arista o no*)
mashermanos($nivel$)
return $s[nivel] < n$

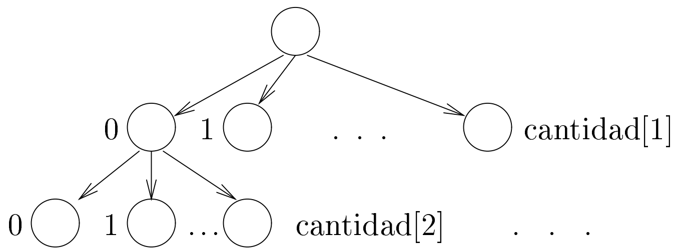
Obviamente las funciones se pueden implementar de muchas maneras distintas, quizás más eficientes que la que aquí se muestra, pues por cada nodo estamos generando n descendientes, mientras que hay un máximo de v y en el penúltimo nivel de uno. Como la función generar genera descendientes sin comprobar si hay una arista que une los dos vértices, las funciones solucion y criterio hacen el trabajo de eliminar los nodos que no representan aristas del grafo.

Problema 6.12 Resolver por Backtracking el problema de la devolución de monedas con el siguiente planteamiento: minimizar el número de monedas a devolver para dar una cantidad C si tenemos monedas de n tipos, estando los tipos de las monedas en un array *tipos*: array$[1, \ldots, n]$ of integer, y teniendo de cada tipo una cierta cantidad de monedas, estando estas cantidades almacenadas en un array *cantidad*: array$[1, \ldots, n]$ of integer (de la moneda de tipo $tipos[i]$ podemos dar una cantidad entre 0 y $cantidad[i]$).

Hay que decir qué estructuras de datos se utilizarían, cómo será el árbol de soluciones, cómo se representan las soluciones, cuál es la condición de fin, y programar las funciones "generar", "solucion", "mas_hermanos" (o "mas_hijos"), "criterio" y el esquema del método.

Solución:

El árbol de búsqueda de la solución tendrá n niveles (más el del nodo raíz), decidiéndose en el nivel i cuántas monedas del tipo i se dan, por lo que la cantidad de hijos de un nodo del nivel i será $cantidad[i+1]+1$, pues necesitamos un valor que indique que no se da ninguna moneda del tipo i. Por tanto los hijos de un nodo estarán numerados de cero hasta $cantidad[i+1]$, indicando ese número el número de monedas que se dan de ese tipo:



Las soluciones se almacenan en una tabla s con índices entre 1 y n, siendo $s[i]$ un valor entre 0 y $cantidad[i]$, indicando $s[i]=j$ que en la solución se dan j monedas de tipo i. El array s estará inicializado a -1 y en la generación se sumará uno a ese valor, por lo que el árbol de búsqueda se recorrerá de izquierda a derecha según la figura anterior.

Como es un programa de minimización, la condición de fin será que se haya recorrido todo el árbol, y por tanto que la variable *nivel*, que indica el nivel que vamos explorando, vuelva a valer cero.

Con el tipo de árbol que estamos utilizando sólo son nodos posible solución los nodos terminales y que sumen la cantidad C. La función solucion será:

solucion($nivel$):

return $nivel = n$ and $\sum_{i=1}^{nivel} tipos[i] * s[i] = C$

El sumatorio sirve para hacer la suma del valor de las monedas que damos, pero sería mejor (menor tiempo de ejecución) utilizar una variable global, que se podría llamar *suma*, donde se almacena la cantidad que se lleva dada. Esta variable tendrá que modificarse cada vez que se varíe de nodo en el árbol. La función solucion queda:

solucion($nivel$):

return $nivel = n$ and $suma = C$

Se pueden seguir generando nodos cuando no estemos en un nodo terminal y el valor de las monedas que damos hasta ese momento no supere a la cantidad a devolver. La función criterio será:

criterio($nivel$):
return $nivel \neq n$ and $suma \leq C$
Un nodo tendrá más hermanos cuando no estemos en el último nodo de ese nivel:
mas_hermanos($nivel$):
return $s[nivel] < cantidad[nivel]$

En la función generar modificaremos la variable $valor$ y, como queremos minimizar el número de monedas a dar, utilizaremos otra variable ($monedas$) que indique el número de monedas que contiene la solución, y que también se actualizará en generar. Si $s[nivel] = 0$ no se actualiza $monedas$, y si no es cero se le sumará una moneda más:

```
generar(nivel):
    s[nivel] = s[nivel + 1
    if s[nivel] ≠ 0
        suma = suma + tipos[nivel]
        monedas = monedas + 1
    endif
```

Para completar el esquema del algoritmo necesitamos una variable (SOA) que almacene la solución óptima actual, y que será un array con índices de 1 a n y valores iniciales -1, y una variable (VOA) que contenga el número de monedas que se dan en la solución óptima actual, y que inicialmente puede tener un valor ∞ que indique que no tenemos solución. Si al acabar la ejecución VOA sigue teniendo ese valor el problema no tiene solución. Cuando se llega a un nodo solución habrá que comparar con la solución óptima actual para ver si se mejora.

Además, en la parte de retroceso del esquema habrá que actualizar las variables $suma$ y $monedas$ y volver a poner $s[nivel]$ a -1 para que al volver a bajar en el árbol empecemos por el valor cero.

El esquema sería:

```
monedas = 0
suma = 0
SOA ← −1
VOA = ∞
s ← −1
nivel = 1
generar(nivel)
repeat
    if solucion(nivel)
        if monedas < VOA
            VOA = monedas
            SOA ← s
        endif
    endif
    if criterio(nivel)
```

```
        nivel = nivel + 1
        generar(nivel)
    else if mas_hermanos(nivel)
        generar(nivel)
    else
        while not mas_hermanos(nivel) and nivel > 0
            monedas = monedas − s[nivel]
            suma = suma − s[nivel] ∗ tipos[nivel]
            s[nivel] = −1
            nivel = nivel − 1
        endwhile
        if nivel > 0
            generar(nivel)
        endif
    endif
until nivel = 0
```

Problema 6.13 a) Programar un algoritmo de backtracking no recursivo según uno de los esquemas vistos en clase para el problema de dado un conjunto de n números y una serie de $n-1$ operaciones binarias entre ellos, encontrar las posibles expresiones que den un resultado N. Las expresiones no tendrán paréntesis y las operaciones se harán siempre de izquierda a derecha, ejemplo 3*4+2=14. Hay que indicar cómo es el árbol de soluciones, cuál es el criterio de fin, programar el esquema que se utiliza y las funciones generar, solucion, criterio, mas_hermanos y retroceder.

b) Dar una idea para disminuir el número de nodos que se generan.

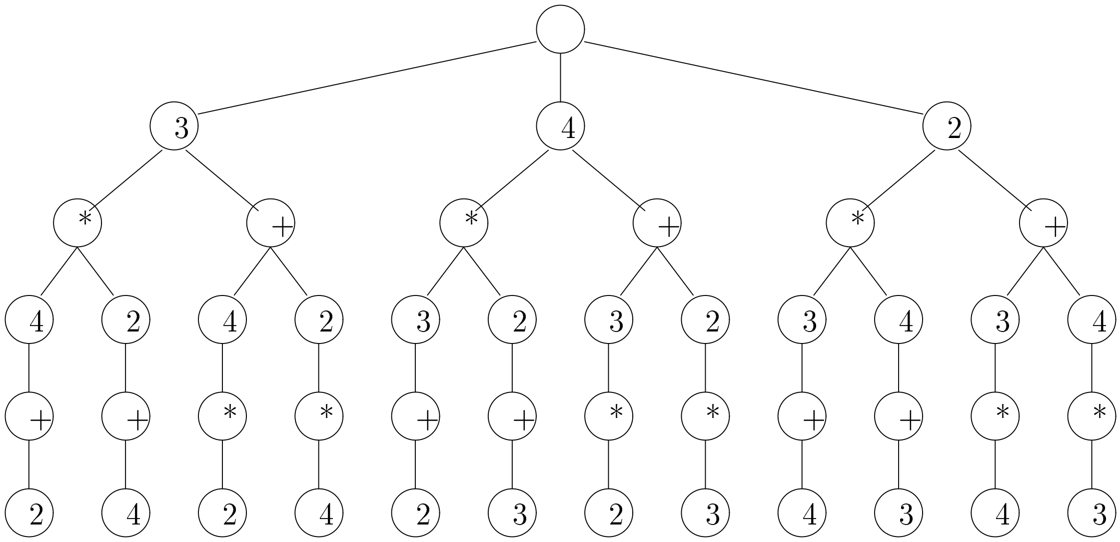
c) Estudiar el número de nodos que se generan.

Solución:

a) Como se trata de encontrar las posibles expresiones que den un resultado hay que recorrer todo el árbol buscando soluciones y cuando se encuentra una tratarla (puede ser escribirla en pantalla, en un fichero, en una lista de soluciones, ...).

Para ver la forma que tendrá el árbol, dado que tenemos n números y $n-1$ operadores, primero habrá que elegir un número (en el ejemplo 3, 4 o 2), en el siguiente nivel un operador (en el ejemplo * o +), en el siguiente otro número de los que todavía no se han elegido (si en el ejemplo se había elegido el 3 se podrá elegir el 4 o el 2), y así sucesivamente. En los niveles impares se elegirán números y en los pares operadores. El orden en que se tomen los números y los operadores importa, por lo que en cada nivel probaremos a tomar todas las posibilidades pero si la elección que hacemos ya se había hecho antes (lo cual nos lo dirá la función criterio) no se continúa por ese nodo.

Con el ejemplo $3, 4, 2$ y $*, +$ el árbol sería:



donde, por motivos de claridad, no aparecen los nodos que representan elecciones repetidas y por los que no se seguirá la búsqueda.

Dado que se buscan todas las configuraciones el esquema del algoritmo será:

```
nivel = 1
generar(s,nivel)
repeat
   if solucion(s,nivel)
      tratar(s)
   endif
   if criterio(s,nivel)
      nivel ++
      generar(s,nivel)
   else
      while (nivel ≠ 0) and (not mas_hermanos(s,nivel)
         retroceder(s,nivel)
      endwhile
      if nivel ≠ 0
         generar(s,nivel)
      endif
   endif
until nivel = 0
```

Tanto si el nodo es solución como si no lo es se comprueba con la función criterio si se puede continuar hacia abajo en el árbol, y si no se puede se comprueba si tiene más hermanos para seguir por ellos o retroceder, poniendo como límite del retroceso que se vuelva al nodo raíz.

Programamos las funciones:

- Un nodo será solución si es terminal y la expresión da una cantidad N:

  solucion(s,$nivel$):

  return ($nivel = 2n - 1$ and $evaluar$(s,nivel) $= N$)

  donde evaluar evalúa la expresión formada. De esta manera el tiempo de ejecución de la función solucion es lineal, y se puede hacer constante si en vez de evaluar la expresión se van evaluando las subexpresiones que se van formando en cada nodo. Esto tiene el problema de que al retroceder o cambiar a un nodo hermano hay que deshacer operaciones, para lo que habría que tener un array de operadores inversos de los que tenemos. No lo haremos de esa manera por simplicidad.

- La función criterio comprueba si se puede seguir hacia abajo en el árbol y si el operador u operando que estamos tomando ha sido tomado ya. Para esto tendremos una array $operandos$ y otro $operadores$ que serán de enteros, indicando su valor el número de veces que se ha tomado un operador u operando. Inicialmente están a 0, y las dimensiones son n para $operandos$ y $n - 1$ para $operadores$.

  criterio(s,$nivel$):

  if impar($nivel$)

  return($nivel \neq 2n - 1$ and $operandos[s[nivel]] = 1$)

  else

  return $operadores[s[nivel]] = 1$

  endif

  Donde suponemos que tenemos una función impar que nos dice si un número es impar o no. La configuración está bien si el operador u operando aparece sólo una vez en la expresión.

- El array s estará inicializado a 0, lo que indica que no se ha tomado ninguna decisión en cada nivel. El valor de $s[i] = j$ indica que en la posición i de la expresión se toma el j-ésimo número u operador, por lo que la función generar consiste en sumar uno a $s[nivel]$, pero además habrá que indicar como tomado el correspondiente operador u operando, y si se genera un nodo hermano de otro generado anteriormente hay que deshacer la decisión anterior:

  generar(s,$nivel$):

  if $s[nivel] \neq 0$

  if impar($nivel$)

  $operandos[s[nivel]] - -$

  else

  $operadores[s[nivel]] - -$

endif
endif
$s[nivel]++$
if impar($nivel$)
$operandos[s[nivel]]++$
else
$operadores[s[nivel]]++$
endif

- En mas_hermanos se comprueba si hemos evaluado hasta el operando n o el operador $n-1$:

  mas_hermanos(s,$nivel$):
  if impar($nivel$)
  return $s[nivel] \neq n$
  else
  return $s[nivel] \neq n-1$
  endif

- Al retroceder se vuelve a poner el valor inicial en s y se sube de nivel, pero indicando que el operador u operando último que se había tomado deja de tomarse:

  retroceder(s,$nivel$):
  if impar($nivel$)
  $operandos[s[nivel]]--$
  else
  $operadores[s[nivel]]--$
  endif
  $s[nivel] = 0$
  $nivel--$

b) En general no se pueden eliminar nodos, pero sí en algunos casos concretos:

- Si la primera operación que se toma es conmutativa, en los tres primeros niveles del árbol se pueden eliminar la mitad de los nodos. Esto se ve en el árbol ejemplo donde se toma $3*4$ pero no hace falta tomar $4*3$, ... Habría que modoficar la función criterio para que trabajara de manera especial en el nivel tres.

- Si todos los operandos son positivos y todos los operadores incrementan el valor (son por ejemplo * y +), cuando la evaluación de la expresión parcial que llevamos pase de N no es necesario seguir. Esto se podría hacer comprobando antes de empezar si el problema es de estas características y si lo es utilizar otra función criterio que incluya la evaluación de las subexpresiones en los niveles impares.

c) Tenemos el nodo raíz, mas n nodos en el primer nivel, $n(n-1)$ en el segundo nivel, $n(n-1)(n-1)$ en el tercer nivel (no consideramos los nodos que se evalúan pero por los que no se sigue), $n(n-1)(n-1)(n-2)$ en el cuarto nivel, ... De este modo el número de nodos es:

$$1+n+n(n-1)+n(n-1)(n-1)+n(n-1)(n-1)(n-2)+n(n-1)(n-1)(n-2)(n-2)+\ldots =$$

$$1+n+n(n-1)+n(n-1)^2(n-1)+n(n-1)^2(n-2)^2(n-2)+\ldots+n(n-1)^2(n-2)^2+\ldots+2^2+1$$

$$\geq n!(n-1)!$$

Problema 6.14 Consideramos n profesores que tienen que dar una cantidad m de horas de clase, estando el número de horas de clase que da cada profesor almacenado en un array $h = (h_1, h_2, \ldots, h_n)$, siendo $h_1 + h_2 + \ldots + h_n = m$. Para atender las preferencias de dichos profesores se permite a cada uno de ellos que asigne un total de m penalizaciones en los m huecos del horario, con lo que se puede obtener una tabla de penalizaciones como la siguiente cuando $m = 4$, $n = 3$ y $h = (2, 1, 1)$:

0	2	2	0
0	0	2	2
1	0	1	2

Se pretende hacer una asignación de horas que minimice la penalización total. Programar un algoritmo por backtracking para resolver este problema. Indicar la forma del árbol de soluciones, del array solución, la condición de fin, programar el esquema utilizado y las funciones "generar", "solucion", "hermanos", "retroceder" y "criterio". Indicar cómo funcionaría el programa con el ejemplo anterior.

Solución:

En cada nivel del árbol se decide a qué profesor se asigna una hora. El árbol tendrá m niveles, más el raíz, y cada nodo tendrá un máximo de n hijos. En el nivel i se decide a quién se asigna la hora i.

La solución será un array s con índices de 1 a m (uno por hora), y valores de 0 a n, indicando valores de 1 a n el profesor al que se asigna la hora, y el valor 0 indica que no se ha asignado. Por tanto s estará inicializado a 0.

La condición de fin será que se vuelva al nodo raíz, pues es un problema de optimización. Se recorrerá todo el árbol comprobando en cada nodo solución si la solución que representa tiene una penalización menor que la solución óptima actual; para esto se utilizará un array SOA inicializado a 0, donde se almacena la solución óptima actual y que tendrá al final de la ejecución la solución, y una variable VOA que contendrá la penalización de la SOA y que estará inicializado a $mn + 1$.

Como es un problema de optimización el esquema es:

$SOA \leftarrow 0$
$VOA = mn + 1$
$s \leftarrow 0$
$v = 0$ (Tendrá la penalización que llevamos hasta ahora)
$nivel = 1$
generar(s,$nivel$,v)
repeat
    if solucion(s,$nivel$)
        if $v < VOA$
            $SOA \leftarrow s$
            $VOA = v$
        endif
    endif
    if criterio(s,$nivel$,v)
        $nivel + +$
        generar(s,$nivel$,v)
    else
        while $nivel \neq 0$ and not hermanos(s,$nivel$)
            retroceder(s,$nivel$,v)
        endwhile
    endif
    if $nivel \neq 0$
        generar(s,$nivel$,v)
    endif
until $nivel = 0$

Un nodo será solución si es terminal y la asignación que se ha hecho no viola la restricción del número de horas que tiene que dar cada profesor. Cuando se asigne una hora se restará uno al número de horas a dar por el profesor, por lo que en un nodo terminal el número de horas que le quedan tiene que ser cero:

solucion(s,$nivel$):
    return $nivel = m$ y $h[nivel] = 0$

Un nodo tendrá más hermanos si no es el correspondiente al último profesor:

hermanos(s,$nivel$):
    return $s[nivel] < n$

Se puede seguir hacia abajo en un nodo si no es terminal, si no se han asignado horas de más al profesor, y si la penalización que llevamos no supera la de la SOA, pues en este caso, y por ser las penalizaciones valores mayores o iguales a cero, no se puede mejorar la solución que tenemos como óptima:

criterio(s,$nivel$,v):
    return $nivel < m$ y $h[nivel] \geq 0$ y $v < VOA$

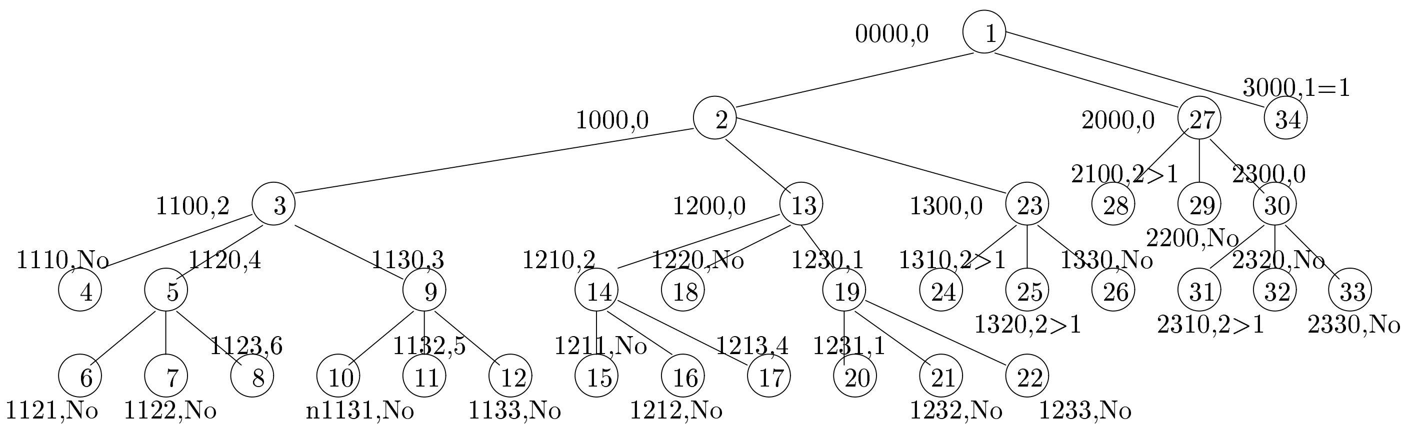
Al generar se suma uno al valor que tengamos en s, para lo que se ha inicializado s a 0; se actualiza v y el número de horas asignadas al profesor. Si se está generando un nodo hermano de otro habrá que quitar la penalización que se había añadido por ese nodo y sumar uno al número de horas disponibles del profesor:

generar(s,$nivel$,v):
    if $s[nivel] \neq 0$
        $v = v - p[s[nivel], nivel]$
        $h[s[nivel]] + +$
    endif
    $s[nivel] + +$
    $v = v + p[s[nivel], nivel]$
    $h[s[nivel]] - -$

Antes de retroceder hay que quitar la penalización que se había acumulado por la última asignación, sumar uno al número de horas disponibles del profesor, y volver a poner s al valor inicial:

retroceder(s,$nivel$,v):
    $v = v - p[s[nivel], nivel]$
    $h[s[nivel]] + +$
    $s[nivel] = 0$
    $nivel - -$

Mostramos a continuación el funcionamiento del algoritmo con el ejemplo. En la figura aparece dentro de cada nodo el número que ocupa en el recorrido, y al lado la solución parcial o total (en los terminales) que representa, y el valor de la variable v. Cuando aparece la palabra `No` se indica que no se cumplen las restricciones en cuanto al número de horas de cada profesor, por lo que la función criterio descartará seguir por ese nodo. Cuando aparece algo como $2 > 1$ o 1=1, se indica que el valor de v es mayor o igual que la penalización de la solución óptima actual, con lo que la función criterio evita que sigamos por ese nodo. Los nodos que son soluciones óptimas actuales son el 8, el 11, el 17 y el 20, que es la solución del problema, y porque tiene valor 1 se eliminan los nodos 24, 25, 28, 31 y 34.



Problema 6.15 Tenemos un conjunto de caracteres con los que se quiere formar secuencias de n caracteres, y cada carácter i debe aparecer n_i veces en la secuencia ($\sum n_i = n$). Además se tiene una tabla de 0 y 1, indicando un 1 en la posición i, j que el carácter j puede aparecer en la cadena después del i, y un cero que no puede aparecer después de i. Hacer un programa por backtracking para encontrar todas las cadenas que cumplen esas condiciones. Habrá que utilizar uno de los esquemas no recursivos vistos, indicando cómo es el árbol de soluciones, cuál es la condición de fin, cómo se representan las soluciones, y hay que programar las funciones "generar", "solucion", "criterio", "mashermanos" y "retroceder".

Solución:

Como se quieren encontrar todas las cadenas que cumplen las restricciones habrá que recorrer todo el árbol y la condición de fin será volver al nodo raíz.

Suponemos que los caracteres los tenemos en un array c:array$[1, \dots, m]$ de caracteres, siendo m el número de caracteres que tenemos, y que el número de veces que cada carácter tiene que aparecer en las cadenas está almacenado en un arrary num:array$[1, \dots, m]$ de enteros. Al introducir un nuevo carácter en la cadena se restará uno al número de veces que puede aparecer, y cuando su valor en num sea cero no podrá incluirse más en la cadena.

Como queremos formar cadenas de n caracteres, la solución se irá almacenando en un array s:array$[1, \dots, n]$ de enteros entre 1 y m, indicando $s[i] = j$ que en la posición i de la cadena se pone el carácter $c[j]$. Por lo tanto, el árbol tendrá n niveles (mas el raíz), indicándose en el nivel i el número del carácter que se pone en la posición i de la cadena, con lo que cada nodo tendrá m descendientes.

La función criterio se encargará de eliminar los nodos desde los que no se puede llegar a cadenas que cumplen las restricciones del problema.

Además, como valor inicial de las soluciones (que nos servirá también para programar de manera adecuada la función generar) pondremos el 0, con lo que los valores que toma s son entre 0 y m, estando inicializado a 0.

Con estas ideas el esquema será:

$s \leftarrow 0$
$nivel = 0$
repeat
    if solucion(s,$nivel$)
        tratarsolucion(s,$nivel$)
    endif   (No parará pues queremos todas las soluciones)
    if criterio(s,$nivel$)
        $nivel++$
        generar(s,$nivel$)
    else
        while $nivel \neq 0$ and not mashermanos(s,$nivel$)
            retroceder(s,$nivel$)
        endwhile
        if $nivel \neq 0$
            generar(s,$nivel$)
        endif
    endif
until $nivel = 0$

Son nodos solución los terminales (se ha completado la cadena de n caracteres) que cumplen las restricciones:

solucion(s,$nivel$):
    return ($nivel = n$ and restricciones(s,$nivel$))

Donde en restricciones se comprueba que el último carácter incluido en la cadena no excede del número de veces que puede aparecer, y que en la tabla aparezca un uno indicando que se puede poner a continuación del carácter anterior. Esto último no es necesario en el primer nivel ni en el nodo raíz:

restricciones(s,$nivel$):
    if $nivel > 1$
        return($num[s[nivel]] \geq 0$ and $tabla[s[nivel-1], s[nivel]] = 1$)
    elsif $nivel = 1$
        return($num[s[nivel]] \geq 0$)
    else
        return true

En tratar solución habrá que hacer lo que se desee con la cadena que se ha encontrado. Podría ser escribirla por pantalla:

tratarsolucion(s,$nivel$):
    for $i = 1$ to n
        escribir($c[s[i]]$)

En criterio se comprueba si se puede seguir hacia abajo en el árbol ($nivel \neq n$) y si no se violan las restricciones:

criterio(s,$nivel$):

return($nivel \neq 0$ and restricciones(s,$nivel$))

En mashermanos se comprueba si es el último hijo de un nodo. Como en generar se irá sumando uno al valor en s, habrá más hermanos mientras no estemos en el último de los posibles caracteres:

mashermanos(s,$nivel$):
return($s[nivel] < m$)

Las funciones generar y retroceder son las que nos sirven para movernos por el árbol, por lo que serán las únicas con las que se modifican las variables que nos describen el nodo en el que estamos.

En generar se actualiza el número de carácter sumando uno (por esto se ha puesto el valor inicial cero), pero si nos movemos de un nodo a otro hermano hay que deshacer la decisión que se había tomado:

generar(s,$nivel$):
if $s[nivel] \neq 0$
$num[s[nivel]] + +$
endif
$s[nivel] + +$
$num[s[nivel]] - -$

Antes de retroceder hay que deshacer la decisión que se tomó al incluir el carácter, y dejar la s a su valor inicial:

retroceder(s,$nivel$):
$num[s[nivel]] + +$
$s[nivel] = 0$
$nivel - -$

Capítulo 7

BRANCH AND BOUND

7.1 Descripción

7.1.1 Ideas generales

La técnica **Branch and Bound** (ramificación y poda) se utiliza en la resolución de problemas de optimización discreta, donde hay que tomar una secuencia de decisiones. Como el Backtracking, esta técnica consiste en hacer un recorrido sistemático en el árbol de soluciones, teniendo en este caso que recorrerse todo el árbol para estar seguros de que hemos encontrado la solución óptima.

Como el recorrido de todo el árbol produciría un tiempo de ejecución prohibitivo en muchos casos, lo que se pretende es utilizar una buena técnica de **ramificación** obteniendo para cada nodo una estimación del beneficio de la solución óptima que se puede encontrar a partir de él, y usar esta estimación para decidir qué zonas del árbol explorar primero, generando en cada paso los hijos del nodo con mayor beneficio estimado (o de menor costo si estamos en un problema de minimización del costo).

La **poda** se realizará calculando en cada nodo dos cotas, una inferior y otra superior, del beneficio que se puede alcanzar a partir de ese nodo. Dependiendo de las características del problema estas cotas se utilizarán de diferente manera. Si estamos seguros de que a partir de un cierto nodo siempre hay solución, cuando $CS(i) \leq CI(j)$ se puede podar el nodo i pues ninguna solución a partir de ese nodo puede superar en beneficio a la solución óptima a partir de j; por tanto, hay que llevar una variable global CI que sea la mayor de las $CI(i)$ de los nodos generados. Si las cotas que se obtienen son cotas de soluciones a partir de ese nodo pero no podemos asegurar que exista ninguna solución cumpliendo las restricciones, el criterio de poda anterior no es válido, y sólo se puede empezar a podar después de haber encontrado una solución factible, podando los nodos en los que $CS(i) \leq B$, siendo B el beneficio de la solución óptima actual.

El principal problema de esta técnica está en la estimación del beneficio y en el cálculo de las cotas, pues no hay métodos generales sino que se suele hacer de manera heurística. Hay que intentar que la estimación del beneficio se aproxime lo más posible al beneficio real que obtendríamos (para guiar mejor la búsqueda), y que las cotas estén lo más próximas posible entre sí (para realizar podas más productivas). Normalmente la estimación del beneficio está entre las dos cotas, y muchas veces los valores están relacionados siendo la estimación del beneficio la cota inferior o superior o una media, que puede ser ponderada, de las dos. Además de que estos valores sean adecuados hay que tener en cuenta que su cálculo conlleva un tiempo de ejecución y que este tiempo de ejecución suele ser mayor para obtener mejores estimaciones y cotas, por lo que se trata de encontrar un equilibrio entre la bondad de los parámetros y el coste de su cálculo. En algunos casos no es posible encontrar cotas y se pueden poner con valores $-\infty$ y ∞, con lo que el método sólo servirá para hacer ramificación, pero no poda.

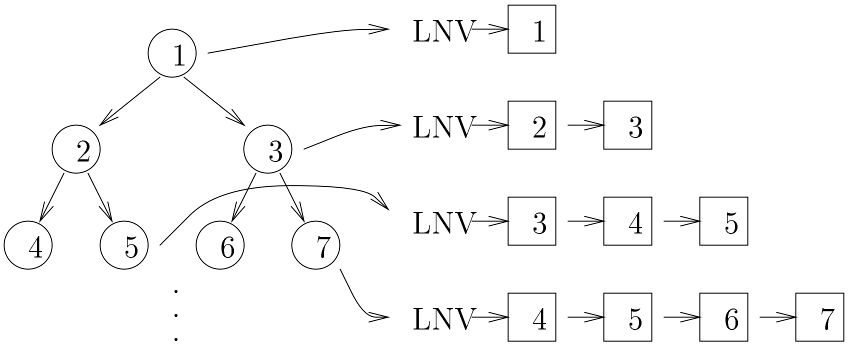
Además, la gestión de la lista de nodos vivos ocasiona un aumento del tiempo de ejecución y de la ocupación de memoria.

7.1.2 Estrategias de ramificación

Para recorrer el árbol se utiliza lo que se llama una **lista de nodos vivos**, donde inicialmente se encuentra el nodo raíz. En esta lista se encuentran los nodos de los que todavía no se han generado los hijos y que no se han podado. La generación de nuevos nodos se realiza tomando un nodo de la lista y generando todos sus hijos (calculando su beneficio estimado, sus cotas inferior y superior, viendo si se puede podar e incluyéndolo en la lista de nodos vivos si no ha sido podado). Hay distintas estrategias de elección de un nodo de la lista de nodos vivos para generar sus hijos:

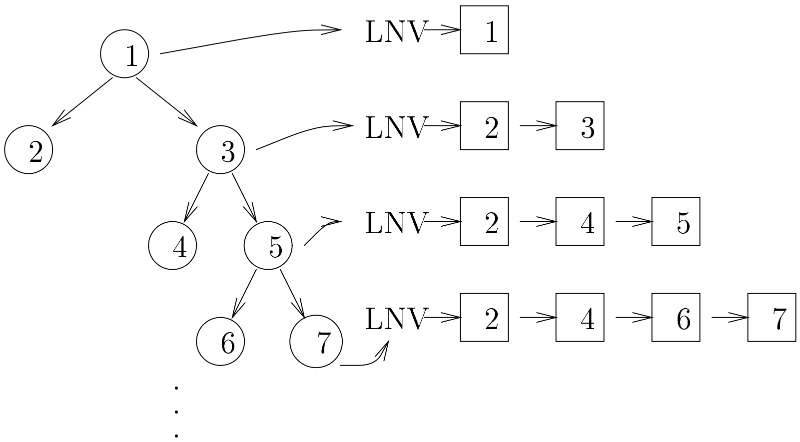
- Estrategia FIFO.

  Cuando no se tiene una estimación del beneficio no sabemos cuál es el nodo más prometedor, por lo que se puede elegir generar los descendientes de los nodos en el mismo orden en que han sido generados (FIFO: first in first out) o en orden inverso (LIFO: last in first out). Por tanto, con la estrategia FIFO la lista de nodos vivos se implementa como una cola. Vemos cómo se recorre el árbol y cómo evoluciona la lista de nodos vivos con un ejemplo pequeño, donde representamos la nueva lista de nodos vivos después de la generación de los hijos de un nuevo nodo:



- Estrategia LIFO.

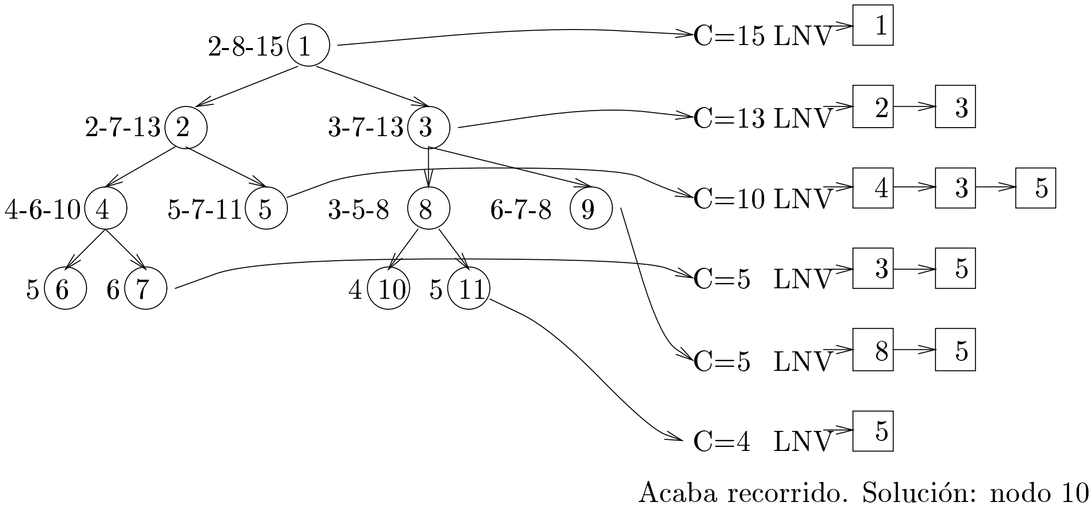
  En este caso la lista de nodos vivos se implementa como una pila:



- Estrategia LC-FIFO.

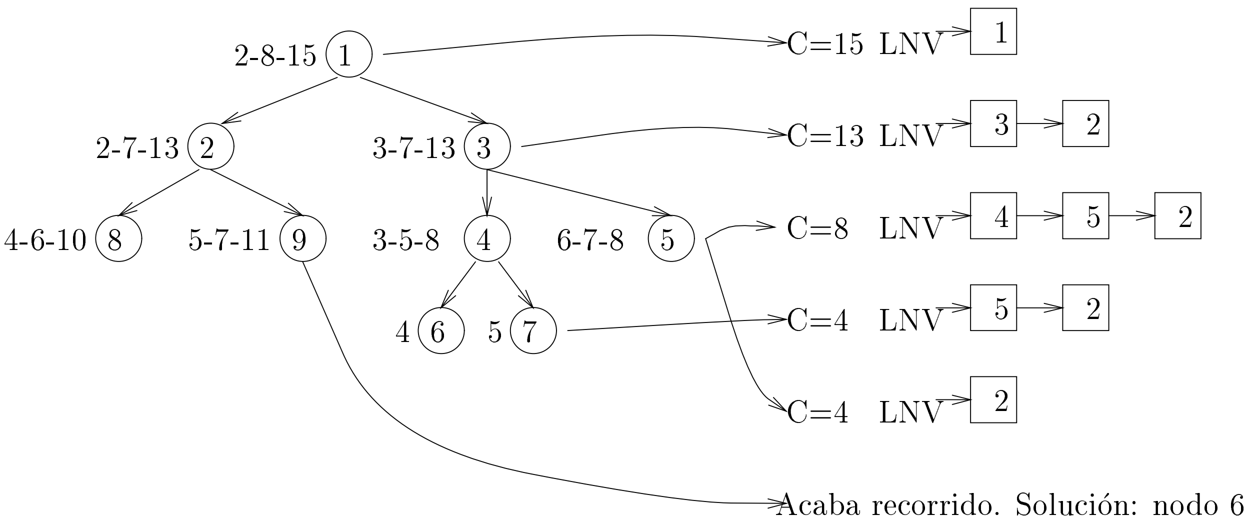
  Cuando se dispone de una estimación del beneficio se elige entre todos los nodos vivos, como nodo para generar sus hijos, el nodo de mayor beneficio (o de menor coste si lo que se quiere es minimizar un coste, de ahí el nombre de LC: least cost), y a igualdad de beneficio se puede seguir una estrategia FIFO o LIFO. Vemos un ejemplo de recorrido con una estrategia LC-FIFO, suponiendo un problema de minimización del coste. En este caso al lado de cada nodo hay tres números, el primero es la cota inferior del coste de una solución óptima a partir de ese nodo, el segundo el coste estimado y el tercero la cota superior. Además de la lista de nodos vivos hay que mantener una variable global C donde almacenamos la menor de las cotas superiores, de manera que si $CI(i) \geq C$ podemos podar el nodo i. Estamos suponiendo que a partir de un nodo siempre hay solución.

En el recorrido del árbol se ve que hay nodos vivos que se pueden podar después de haber obtenido una solución cuyo coste no pueden disminuir (nodos 5 y 9), pero consideramos que no se podan hasta que no se eligen de la lista de nodos vivos para generar sus hijos, comprobando en ese momento que no pueden mejorar la solución óptima actual y, por lo tanto, no generando sus hijos. Se acaba el recorrido del árbol cuando no quedan nodos vivos.



- Estrategia LC-LIFO.

  Vemos el recorrido del árbol anterior con esta estrategia:



7.1.3 Problema de las reinas

El problema de las reinas es un problema típico de backtracking que difícilmente puede adaptarse al esquema del Branch and Bound que además es apropiado para problemas de optimización, mientras que en el problema de las reinas no estamos intentando optimizar ninguna función.

A pesar de todo, podemos abordar el problema de las reinas por Branch and Bound, asociando cotas inferior y superior a cada nodo los valores $-\infty$ y $+\infty$, e intentando maximizar el beneficio, pudiendo indicar el valor $-\infty$ que no hay solución y el $+\infty$ que sí hay solución. Estos valores no servirán para podar el árbol, nada más que tras haber encontrado una solución, en cuyo caso no se siguen buscando porque hemos alcanzado el valor $+\infty$.

En cuanto a la estimación del beneficio, podemos considerar que es más prometedora una configuración cuantas más casillas tengamos no alcanzables desde ninguna de las reinas puestas en el tablero. De esta manera el recorrido del árbol de búsqueda con $n = 4$ será el de la figura 7.1. En esta figura se muestran al lado de cada tablero dos números, el primero indica el orden en que se recorre el árbol, y el segundo es la estimación del beneficio en ese nodo. Se utiliza una estrategia LC-FIFO. Comparando con la figura representada en el capítulo del backtracking para este mismo problema vemos que no reducimos el número de nodos explorados, a pesar de aumentar el tiempo de ejecución en cada nodo al tener que hacer la estimación del beneficio.

Esto no quiere decir que no pueda hacer una estimación del beneficio productiva, ¡sólo que no hemos sabido hacerla! Podemos intentar hacerlo mejor: tendremos en cuenta el número de casillas libres pero multiplicando el número de casillas libres en cada fila por el número de fila en la que están, de manera que es más favorable tener casillas libres en las últimas filas, pues en estas filas es donde más problemas vamos a tener a la hora de intentar poner las reinas. El recorrido del árbol se muestra en la figura 7.2. Vemos que con esta nueva estimación del beneficio seguimos generando la misma cantidad de nodos a pesar de ser un criterio que heurísticamente parece mejor que el anterior. La diferencia debe notarse más con problemas mayores.

Pero puede que esto sea mejorable: podemos tener en cuenta el nivel por el que vamos, pues parece mejor que si estamos cerca del final del árbol, aunque queden pocas casillas libres, sigamos explorando esa zona pues quedan pocas casillas pero pocas reinas por poner. De esta forma la estimación del beneficio puede ser la anterior dividida por el número de reinas que quedan por poner. El recorrido del árbol se muestra en la figura 7.3. Se ve que ahorramos un nodo respecto a los métodos anteriores.

7.2 Secuenciamiento de trabajos

Consideramos un problema de asignación de tareas un poco más general que el estudiado con la técnica de avance rápido. Tenemos n trabajos y un único proce-

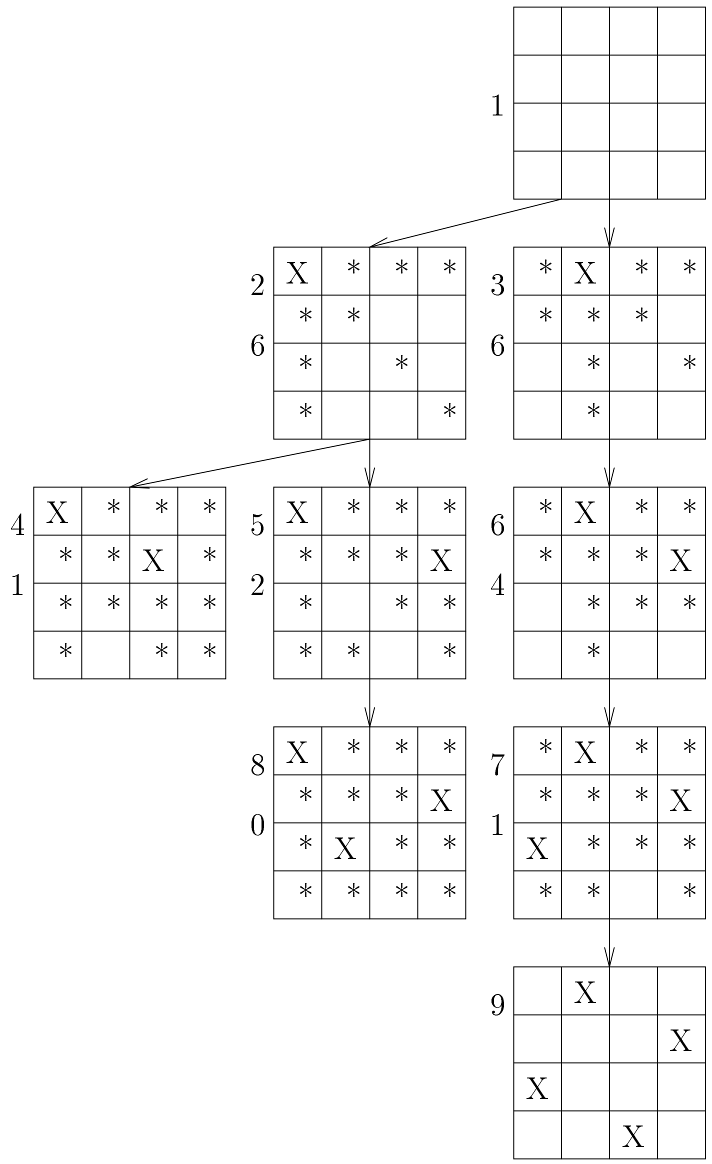


Figura 7.1: Recorrido del árbol en el problema de las 4 reinas cuando se estima el beneficio proporcionalmente al número de casillas no alcanzables

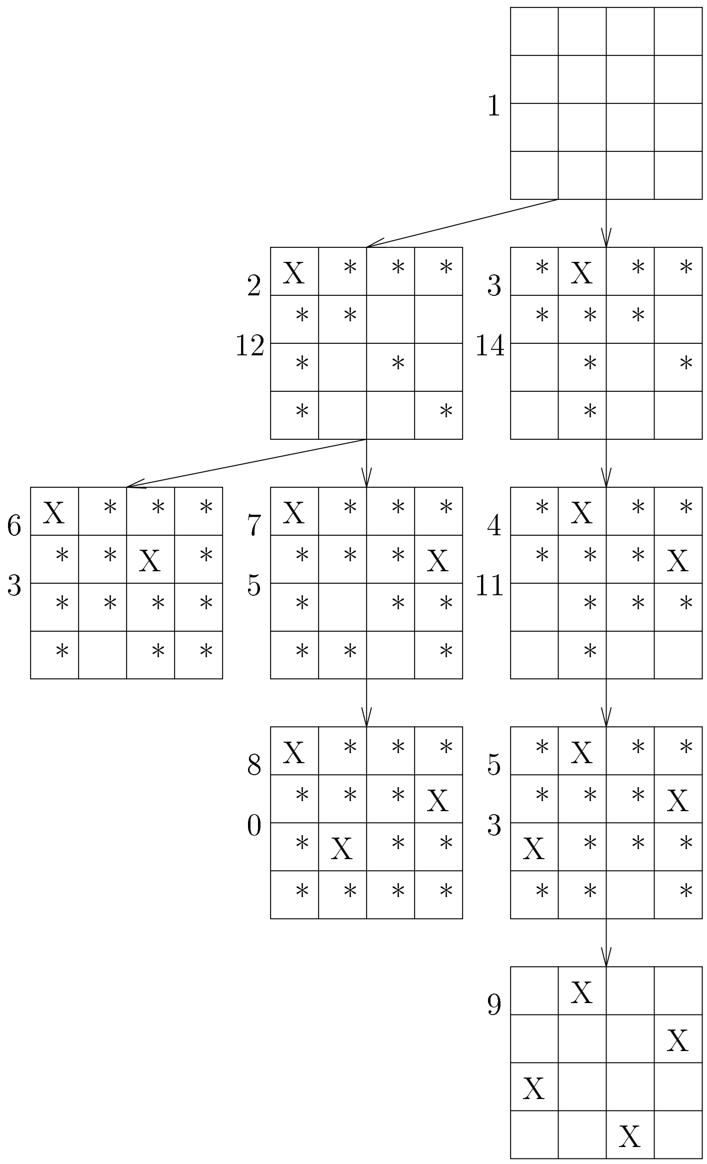


Figura 7.2: Recorrido del árbol en el problema de las 4 reinas cuando se estima el beneficio proporcionalmente al número de casillas no alcanzables teniendo en cuenta la fila en la que están las casillas

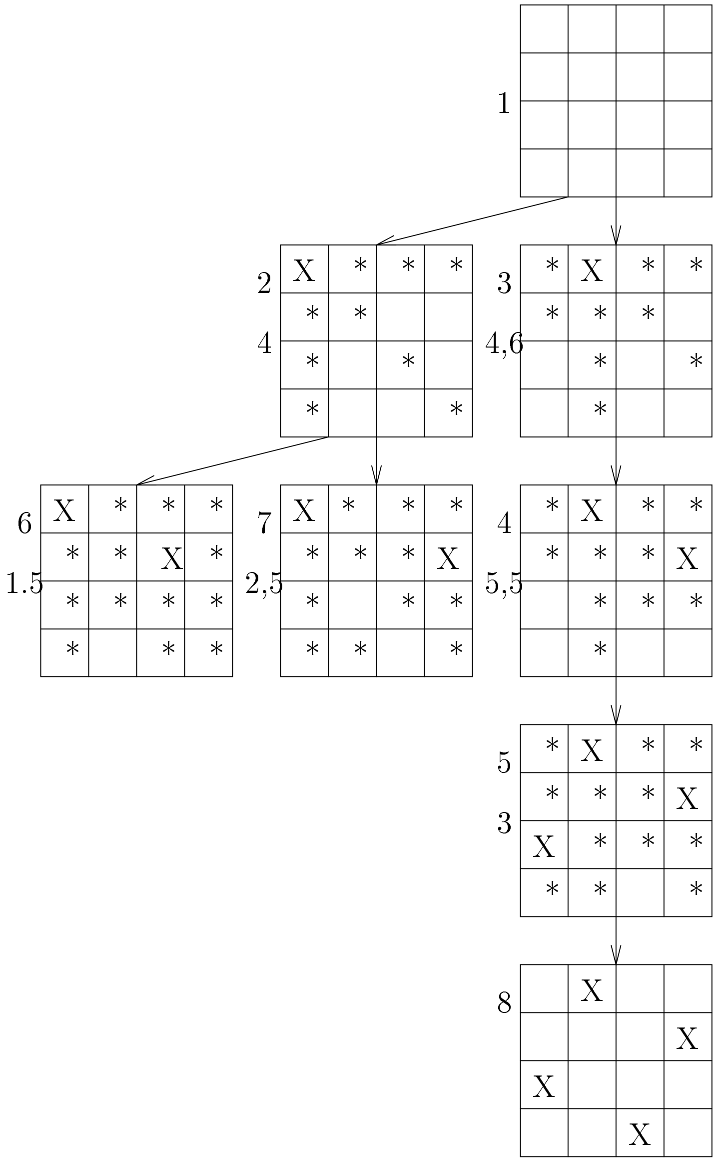


Figura 7.3: Recorrido del árbol en el problema de las 4 reinas cuando se estima el beneficio proporcionalmente al número de casillas no alcanzables teniendo en cuenta la fila en la que están las casillas y el nivel en que se encuentra el nodo

sador, un desplazamiento por cada trabajo $d = (d_1, d_2, \ldots, d_n)$, unas penalizaciones $p = (p_1, p_2, \ldots, p_n)$, y unas duraciones de los trabajos $t = (t_1, t_2, \ldots, t_n)$, y se tiene la penalización p_i si el trabajo i no empieza a ejecutarse dentro de su plazo d_i.

Ejemplo 7.1

Consideramos el problema con $p = (5, 10, 6, 3)$, $d = (1, 3, 2, 1)$ y $t = (1, 2, 1, 1)$. Consideramos que la cota inferior en cada nodo es la penalización que llevamos hasta ese momento, y la cota superior esa penalización más la suma de todas las penalizaciones de los trabajos que no hemos asignado todavía (lo peor que puede pasar es que no se pueda ejecutar ningún trabajo más). En la figura 7.4 se muestra el recorrido del árbol con una estrategia FIFO.

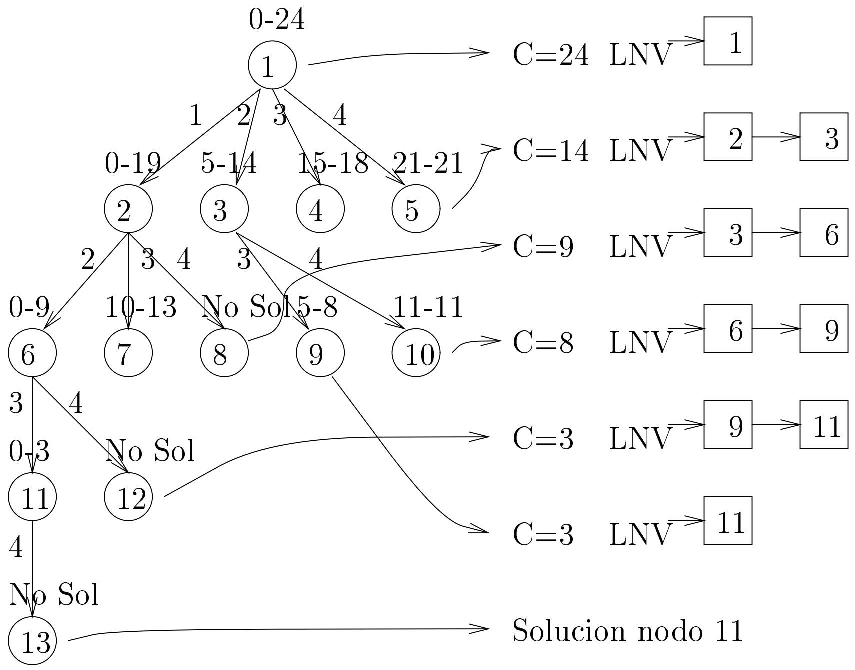


Figura 7.4: Recorrido del árbol de soluciones del ejemplo, con estrategia FIFO

Lo difícil puede ser la comprobación de si forman solución posible, lo que se puede hacer de manera similar a como se hacía en el método de avance rápido, manteniendo los trabajos ordenados por desplazamientos, pero teniendo en este caso en cuenta los tiempos de ejecución de cada trabajo. La solución será ejecutar los trabajos 1, 2 y 3 (nodo 11), pero en el orden 1,3,2.

Si no tenemos otra manera de estimar el coste se puede tomar por ejemplo la media entre los dos costes, de este modo el recorrido con estrategia LC-FIFO es el de la figura

7.5, donde se ve que el utilizar una mejor estrategia de ramificación nos ha reducido un poco el número de nodos a generar.

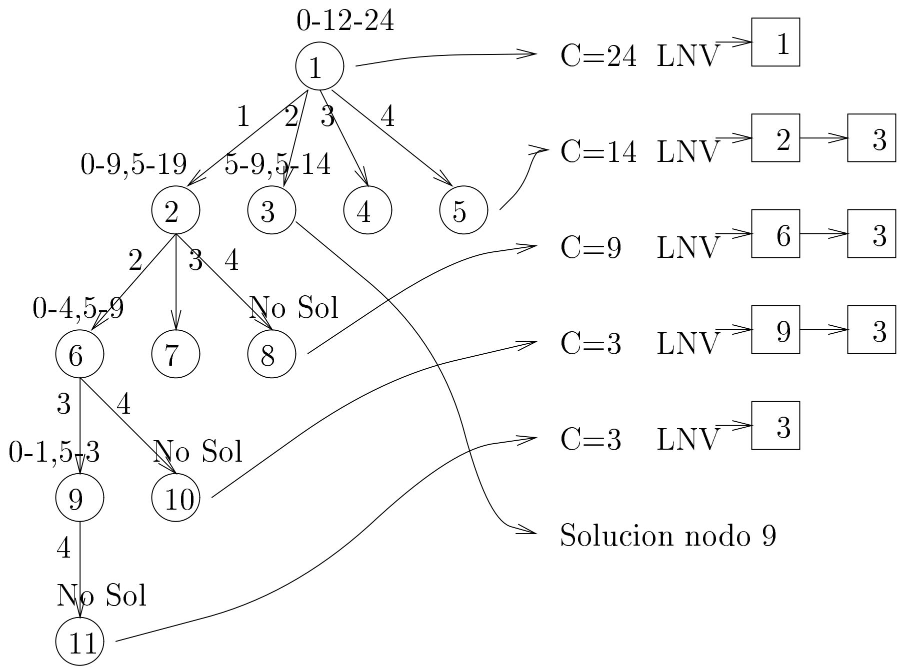


Figura 7.5: Recorrido del árbol de soluciones del ejemplo, con estrategia LC-FIFO

El utilizar otro tipo de árbol también puede reducir el número de nodos a generar. Considerando un árbol binario tendríamos la figura 7.6. Aunque con la representación en árbol binario hay más nodos que con la representación anterior, el número de nodos generados es menor en este caso, debido a que se generan menos hijos de cada nodo y la poda es más productiva.

7.2.1 Estructura para la programación

Para realizar programas para la técnica Branch and Bound hay que tener en cuenta que el árbol que se recorre es un árbol lógico pero que es preferible no implementarlo pues ocuparía mucha memoria y su recorrido sería muy costoso. Se podría implementar utilizando una tabla donde cada entrada tuviera varios campos: *padre*, *siguiente*, CI, *estimacion*, CS y *valor*, por ejemplo; donde cada entrada corresponde a un nodo, indicando *padre* la posición del nodo padre en la tabla, *siguiente* la posición del siguiente en la lista de nodos vivos (implementada en la tabla), CI, *estimacion* y CS la cota inferior, la estimación del coste y la cota superior, y *valor* el trabajo a que corresponde

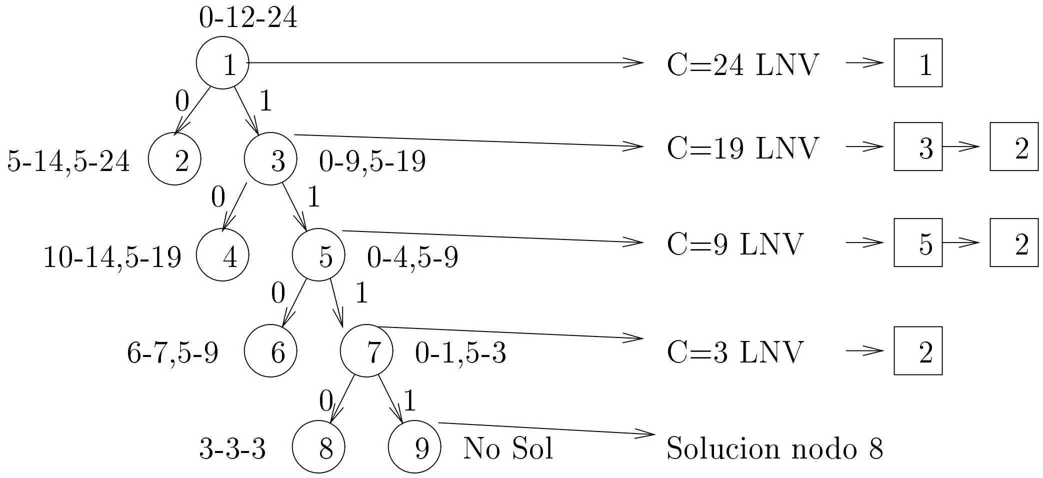


Figura 7.6: Recorrido del árbol de soluciones del ejemplo, con árbol binario y estrategia LC-FIFO

el nodo. Habrá que mantener también una variable C donde almacenamos la menor de las cotas superiores, y LNV que indicará la posición en la tabla donde está el primer nodo de la lista de nodos vivos.

Ejemplo 7.2

Analizamos cómo evolucionan los datos en el recorrido del árbol del ejemplo 7.1 cuando se utiliza la estrategia LC-FIFO y el árbol de soluciones no binario 7.5.

Inicialmente tenemos:

$C = 24$, $LNV = 1$, $\begin{array}{ccccccc} 1: & 0 & 0 & 0 & 12 & 24 & 0 \end{array}$

Tras generar los nodos 2, 3, 4 y 5, sólo añadimos el 2 y 3, pues el 4 y 5 no pueden dar lugar a solución óptima mejor que la del nodo 3:

$$C = 14,\ LNV = 2,\quad \begin{array}{ccccccc} 1: & 0 & 0 & 0 & 12 & 24 & 0 \\ 2: & 1 & 3 & 0 & 9.5 & 19 & 1 \\ 3: & 1 & 0 & 5 & 9.5 & 14 & 2 \end{array}$$

Tras generar los nodos 6, 7 y 8 sólo se incluye el 6 en la tabla, pues con el 7 no se puede obtener mejor solución que con el 6, y el 8 no cumple las restricciones (no se pueden ejecutar los trabajos dentro de sus plazos):

$$C = 9,\ LNV = 4,\quad \begin{array}{ccccccc} 1: & 0 & 0 & 0 & 12 & 24 & 0 \\ 2: & 1 & 0 & 0 & 9.5 & 19 & 1 \\ 3: & 1 & 0 & 5 & 9.5 & 14 & 2 \\ 4: & 2 & 3 & 0 & 4.5 & 9 & 2 \end{array}$$

$C = 3$, $LNV = 5$,

1:	0	0	0	12	24	0
2:	1	0	0	9.5	19	1
3:	1	0	5	9.5	14	2
4:	2	0	0	4.5	9	2
5:	4	3	0	1.5	3	3

$C = 3$, $LNV = 3$,

1:	0	0	0	12	24	0
2:	1	0	0	9.5	19	1
3:	1	0	5	9.5	14	2
4:	2	0	0	4.5	9	2
5:	4	0	0	1.5	3	3

$C = 3$, $LNV = 0$,

1:	0	0	0	12	24	0
2:	1	0	0	9.5	19	1
3:	1	0	5	9.5	14	2
4:	2	0	0	4.5	9	2
5:	4	0	0	1.5	3	3

Y se acaba al ser $LNV = 0$ (lista vacía). El valor óptimo es el de $C = 3$, y para recomponer la solución hay que recorrer el árbol, utilizando el campo *padre* de los nodos, desde el último nodo donde se modificó el valor de C hasta llegar a la raíz: posición 5, $valor = 3$; posición 4, $valor = 2$; posición 2, $valor = 1$; y nodo raíz; con lo que los trabajos son 3, 2 y 1.

En el ejemplo vemos que el tamaño de la tabla es bastante pequeño, pero no podemos predecir cómo de grande va a ser, por lo que en vez de una tabla podríamos utilizar variables dinámicas.

La necesidad de guardar los nodos que han muerto viene del hecho de que los recorremos para recomponer la solución, aunque esto no es necesario si en cada nodo guardamos la solución completa a que corresponde.

7.3 Problema de la mochila

Ya hemos visto en el capítulo anterior cómo el problema de la mochila 0/1 se puede resolver por backtracking y en el ejemplo 6.2 estudiamos la influencia de los distintos factores del backtracking en el número de nodos que se generan y por lo tanto en el tiempo de ejecución. Utilizaremos el mismo ejemplo para comparar el comportamiento de la técnica Branch and Bound con la de Backtracking:

Ejemplo 7.3

Recordamos los valores del problema: $n = 4$, $M = 7$, $p = (2, 3, 4, 5)$ y $w = (1, 2, 3, 4)$.

Consideramos que en cada nodo la cota inferior es el beneficio que se obtendría incluyendo los objetos que se han incluido hasta ese nodo, que la estimación del beneficio es el que se obtendría incluyendo objetos enteros desde ese nodo utilizando la técnica de avance rápido, y la cota superior se obtiene resolviendo el problema no 0/1 a partir de ese nodo y tomando la parte entera de la solución, tal como hacíamos en el ejemplo 6.2.

Consideramos primero el caso con soluciones compuestas por 0 y 1, y que se generan en el orden 0,1. El recorrido (LC-FIFO) del árbol viene dado en la figura 7.7.

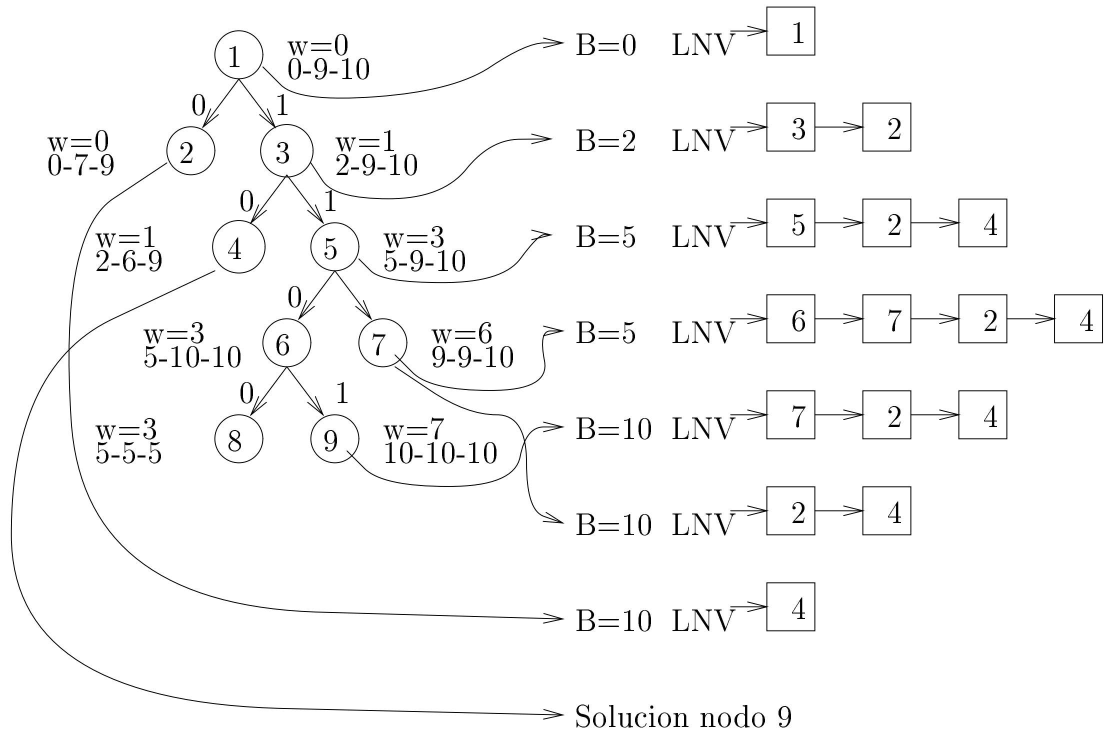


Figura 7.7: Recorrido del árbol de soluciones del ejemplo, con árbol binario

Vemos que se recorren 9 nodos mientras que con el método del backtracking se recorrían 15. Esto no quiere decir que el Branch and Bound recorra siempre menos nodos que el Backtracking. Por ejemplo, si generamos las soluciones primero 1 y después 0 (recorremos el árbol de derecha a izquierda), con Branch and Bound se siguen recorriendo 9 nodos, pero con Backtracking se recorrían 8. Además, si consideramos las soluciones con valores 1, 2, 3 y 4, con el backtracking se recorren 6 nodos y con el Branch and Bound 11, tal como se muestra en la figura 7.8.

¿Qué quiere decir esto? No hay normas generales que nos digan cuándo es preferible

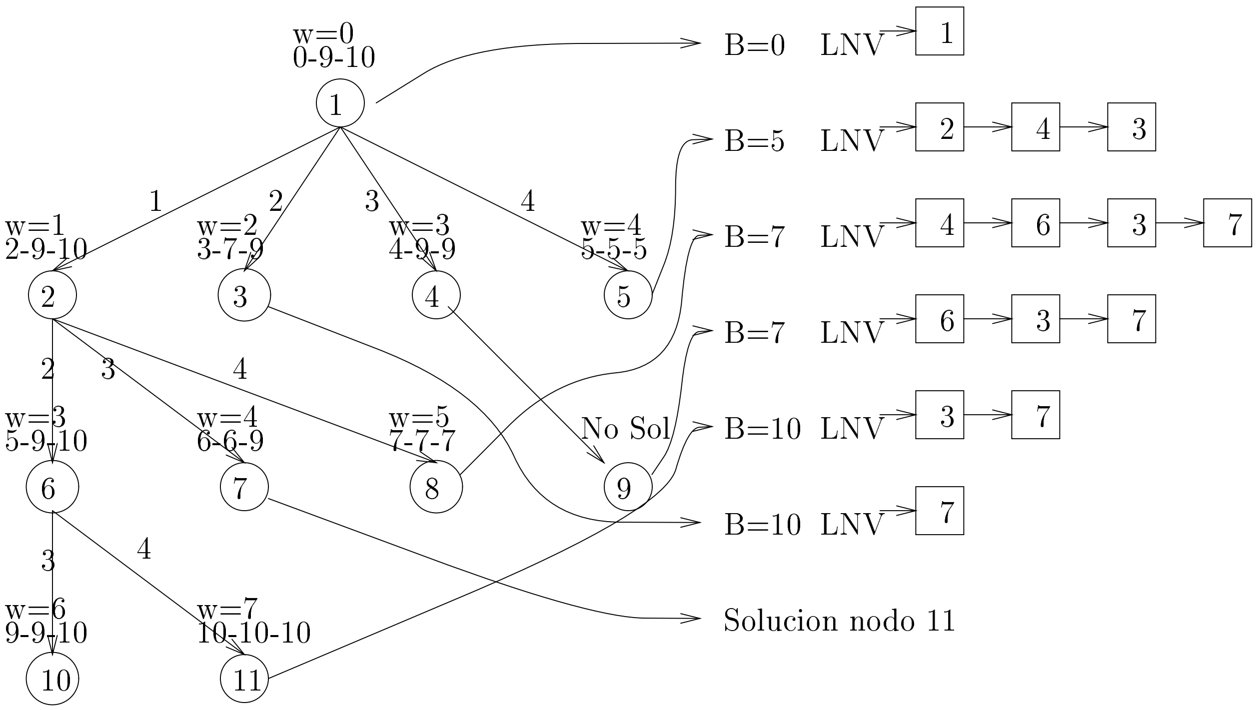


Figura 7.8: Recorrido del árbol de soluciones del ejemplo, con árbol combinatorio

un método u otro, pero cuando el problema es muy grande (lo que no ocurre en los ejemplos) el número de nodos crece exponencialmente, por lo que suele ser preferible utilizar el método que más nodos pode, y por lo tanto es preferible una técnica Branch and Bound, siempre que se puedan encontrar unas cotas y una estimación del beneficio que parezcan apropiadas.

7.4 Árboles de juegos

Consideramos juegos con dos jugadores A y B que mueven alternativamente. Este tipo de juegos se puede representar con un árbol de juegos, donde en un nivel mueve el jugador A y en el siguiente el B. El objetivo de cada jugador es ganar, y los nodos terminales corresponden a posiciones finales donde se sabe qué jugador gana o si quedan empatados. Basándose en la información de los nodos terminales se puede asignar un valor a los nodos que no son terminales, y cada jugador puede decidir el movimiento a hacer intentando maximizar su esperanza de ganar. Resolver el juego de esta manera consiste en hacer un recorrido por el árbol a partir de cada nodo por el que pase el juego (cada instante del juego) antes de hacer el movimiento óptimo. Lo veremos con un ejemplo:

Ejemplo 7.4 *Juego del NIM.*

Tenemos n palos en un montón. Los jugadores A y B mueven alternativamente, consistiendo un movimiento en quitar 1, 2 ó 3 palos del montón, pero no más de los que hay. Pierde el último jugador que mueve.

El estado del juego viene determinado por el jugador que mueve (depende del nivel del árbol en el que se esté) y el número de palos que quedan. Estamos en una configuración terminal cuando no quedan palos en el montón.

Un juego viene representado por una secuencia $C_1, C_2, \ldots, C_m$ que cumple:

- C_1 es la configuración inicial. Por ejemplo, si mueve A y tenemos 4 palos, será $C_1 = (4, A)$.
- Cada C_i con $0 < i < m$ son no terminales.
- C_{i+1} se obtiene de C_i por medio de un movimiento legal de A si i es impar, y de un movimiento legal de B si i es par.
- C_m es una configuración terminal. Por ejemplo, $C_m = (0, X)$ siendo X el jugador ganador.

Un juego es finito cuando no hay secuencias válidas de longitud ∞. Los juegos finitos se pueden representar por árboles, y los juegos infinitos también, pero representando en este caso sólo parte del árbol (la parte que se está explorando).

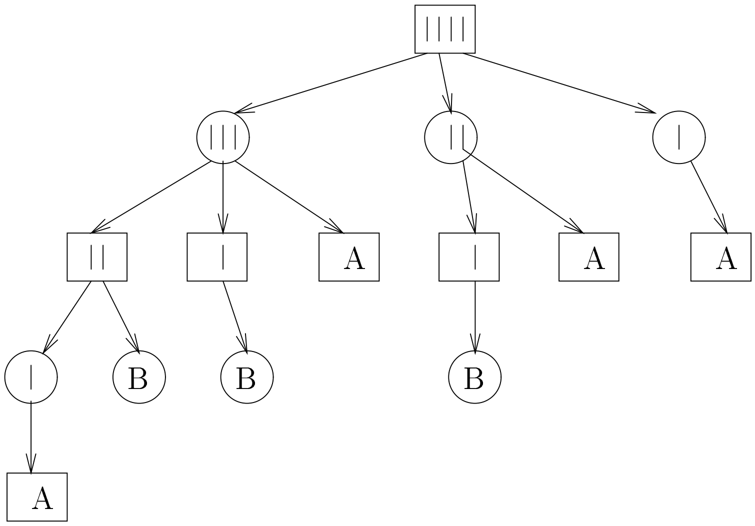


Figura 7.9: Árbol para el juego del NIM

Un árbol para este juego puede ser el de la figura 7.9. Un nodo cuadrado representa una configuración donde le toca mover al jugador A, y uno circular donde le corresponde mover al jugador B. El número de palos dentro de un nodo representa el número de palos en ese momento del juego, y cuando dentro de un nodo terminal hay una A o B representa el jugador que gana.

7.4.1 Procedimiento minimax

Un jugador hará el movimiento que maximice sus posibilidades de ganar independientemente de lo que haga el otro jugador. Necesitaremos unas funciones de evaluación que serán:

$$E(X) = \begin{cases} 1 & si\ en\ X\ es\ ganador\ A \\ -1 & si\ en\ X\ es\ ganador\ B \end{cases}$$

cuando X es terminal, y

$$V(X) = \begin{cases} max\{V(hijos(X))\} & si\ en\ X\ mueve\ A \\ min\{V(hijos(X))\} & si\ en\ X\ mueve\ B \end{cases}$$

Estamos viendo el juego desde el punto de vista del jugador A, con lo que este jugador considera positivo ganar (valor 1) y negativo perder (valor -1), e intenta maximizar, tomando de todos sus posibles hijos la decisión que le lleva a un valor mayor. Sin embargo, B intenta minimizar el beneficio de A.

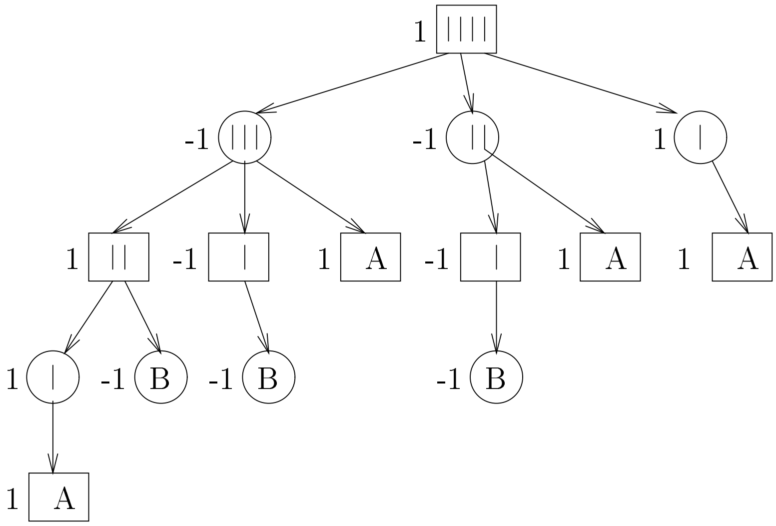


Figura 7.10: Árbol para el juego del NIM. Procedimiento minimax

Ejemplo 7.5 *Juego del NIM. Procedimiento* **minimax**.

En la figura 7.10 consideramos el mismo árbol de la figura 7.9 asociando a cada nodo el valor E o V. Para calcular estos valores, en cada nodo hay que recorrer sus descendientes, por lo que si el programa juega como jugador A tendría inicialmente que recorrer todo el árbol para poder establecer el valor en el nodo raíz. En la figura vemos que el juego lo puede ganar A jugando óptimamente, lo que es quitando 3 palos en el primer movimiento, pero en cualquiera de las otras dos posibilidades gana B si juega bien.

A partir del primer recorrido del árbol ya tendríamos la información de todos los nodos si tuvieramos almacenado el árbol del juego, pero esto no siempre es posible pues estos árboles pueden ser excesivamente grandes si el juego no es trivial. En caso de llegar a un nodo del que no se ha podido almacenar su valor, el programa tendrá que volver a recorrer todos los desdendientes de ese nodo. Esto produce que el tiempo de ejecución sea muy elevado, por lo que es mejor utilizar un método de poda para ahorrarnos recorrer algunos nodos. La idea es similar a la del Branch and Bound y la estudiaremos en la subsección siguiente.

En juegos con muchos nodos puede decidirse no recorrer todo el árbol (pues tiene un coste prohibitivo) sino examinar los niveles más cercanos y usar una función de evaluación de los nodos (de manera heurística) que prediga qué nodos son mejores.

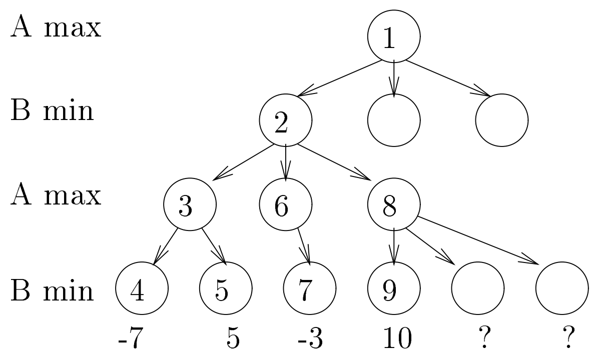


Figura 7.11: Árbol de juego

7.4.2 Método alpha-beta

El procedimiento alpha-beta es una manera de mejorar el método minimax evitando el recorrido de nodos que no van a aportar información adicional, lo que quiere decir que su recorrido y evaluación no va a modificar el valor del máximo o el mínimo que se está calculando. Lo vemos en el ejemplo siguiente.

Ejemplo 7.6 *Procedimiento* **alpha-beta**.

Consideramos un juego con dos jugadores A y B, queriendo A maximinar su ganancia y B maximizar la suya, o lo que es lo mismo minimizar la de A. De este modo, A querrá maximizar una función que B quiere minimizar. Analizaremos la evaluación del nodo raíz en la figura 7.11. Cuando se llega a un nodo terminal se tiene un valor que indica la ganancia de A o B en una cierta cantidad. Algunos nodos terminales tienen asociada una interrogación pues no hace falta recorrerlos para obtener el valor que queremos calcular.

Analizamos el recorrido en una tabla mostrando en cada fila el nodo que se está recorriendo y en cada columna el valor que hemos obtenido en un nodo terminal, el valor α (que indica el valor actual del máximo que estamos calculando) en un nodo donde mueve A, o el valor β (que indica el valor actual en el cálculo de un mínimo) en un nodo donde mueve B.

	1 α	2 β	3 α	4	5	6 α	7	8 α	9
1	$-\infty$								
2	$-\infty$	∞							
3	$-\infty$	∞	$-\infty$						
4	$-\infty$	∞	$-\infty$	-7					
3	$-\infty$	∞	-7	-7					
5	$-\infty$	∞	-7	-7	5				
3	$-\infty$	∞	5	-7	5				
2	$-\infty$	5	5	-7	5				
6	$-\infty$	5	5	-7	5	$-\infty$			
7	$-\infty$	5	5	-7	5	$-\infty$	-3		
6	$-\infty$	5	5	-7	5	-3	-3		
2	$-\infty$	-3	5	-7	5	-3	-3		
8	$-\infty$	-3	5	-7	5	-3	-3	$-\infty$	
9	$-\infty$	-3	5	-7	5	-3	-3	$-\infty$	10
8	$-\infty$	-3	5	-7	5	-3	-3	10	10
2	$-\infty$	-3	5	-7	5	-3	-3	10	10
1	-3	-3	5	-7	5	-3	-3	10	10

La última vez que se llega al nodo 2 no se modifica el valor de β, ya que el valor α del nodo 8 es 10 y por tanto el máximo en ese nodo es ≥ 10, como el mínimo en el nodo 2 se obtiene como el mínimo de los valores en los nodos 3, 6 y 8, y estos valores son 5, -3 y ≥ 10 resulta que el mínimo es -3 sin necesidad de evaluar los dos nodos terminales marcados con una interrogación.

El alpha valor de una posición max (donde se calcula el máximo) está definido como el mínimo valor posible de esa posición. Si el valor de una posición min es $\leq$ que el alpha valor de su padre no hace falta generar los hijos de esa posición.

El valor beta de una posición min es el máximo valor posible para esa posición. Si el valor de una posición max es $\geq$ que el beta de su nodo padre no hace falta generar los hijos de esa posición.

Ejemplo 7.7 *Juego del NIM. Procedimiento* **alpha-beta**.

Volvemos a analizar el juego del NIM, pero esta vez estudiamos el recorrido del árbol utilizando el procedimiento alpha-beta (figura 7.12), y detallamos el recorrido de los nodos y los valores obtenidos de α y β en la tabla siguiente:

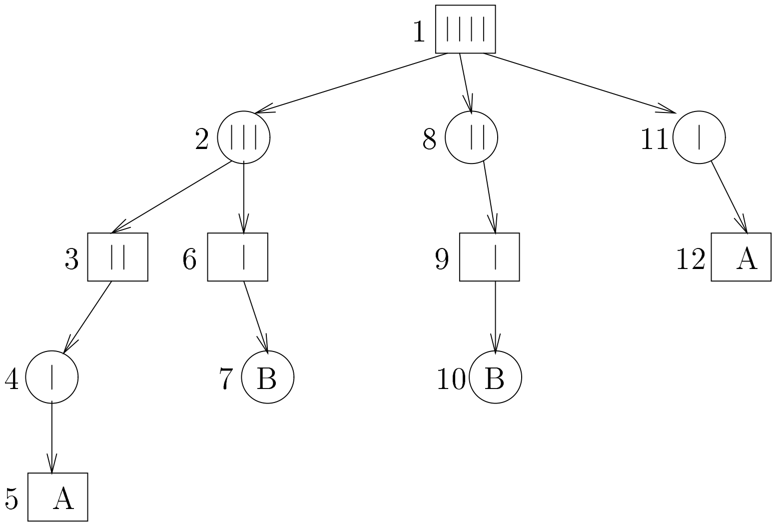


Figura 7.12: Árbol para el juego del NIM. Procedimiento alpha-beta

	1 α	2 β	3 α	4 β	5	6 α	7	8 β	9 α	10	11 β	12
1	$-\infty$											
2	$-\infty$	∞										
3	$-\infty$	∞	$-\infty$									
4	$-\infty$	∞	$-\infty$	∞								
5	$-\infty$	∞	$-\infty$	∞	1							
4	$-\infty$	∞	$-\infty$	1	1							
3	$-\infty$	∞	1	1	1							
No se puede mejorar. No se genera el nodo quitando dos												
2	$-\infty$	1	1	1	1							
6	$-\infty$	1	1	1	1	$-\infty$						
7	$-\infty$	1	1	1	1	$-\infty$	-1					
6	$-\infty$	1	1	1	1	-1	-1					
2	$-\infty$	-1	1	1	1	-1	-1					
No se puede mejorar. No se genera el nodo quitando tres												
1	-1	-1	1	1	1	-1	-1					
8	-1	-1	1	1	1	-1	-1	∞				
9	-1	-1	1	1	1	-1	-1	∞	$-\infty$			
10	-1	-1	1	1	1	-1	-1	∞	$-\infty$	-1		
9	-1	-1	1	1	1	-1	-1	∞	-1	-1		
8	-1	-1	1	1	1	-1	-1	-1	-1	-1		
Su valor es $\leq \alpha$ del padre. No se genera el nodo quitando dos												
1	-1	-1	1	1	1	-1	-1	-1	-1	-1		
11	-1	-1	1	1	1	-1	-1	-1	-1	-1	∞	
12	-1	-1	1	1	1	-1	-1	-1	-1	-1	∞	1
11	-1	-1	1	1	1	-1	-1	-1	-1	-1	1	1
1	1	-1	1	1	1	-1	-1	-1	-1	-1	1	1

Vemos que algunos nodos se han podido podar, tal como pretendíamos. Dos nodos se han podado teniendo en cuenta las características del problema (valores entre -1 y 1) y uno por la utilización del método alpha-beta.

Teniendo en cuenta todo lo anterior, tendremos una función para evaluar el valor en un nodo donde mueve el jugador A y otra para evaluar donde mueve el B. Estas funciones dependen de la configuración del juego en cada nodo. Pondremos un nivel máximo de exploración del árbol, con lo que en los nodos terminales conoceremos el valor exacto y en los nodos no terminales tendremos alguna función que estime el valor de esa posición. Además, en el cálculo del valor de un nodo en el que mueve A hay que conocer el valor beta del padre para ver si se puede podar, y en nodos donde mueve B hay que conocer el alpha del padre. Un esquema de los algoritmos es el que sigue:

Algoritmo 7.1 *Evaluación del valor en un nodo donde mueve A:*

```
function A(x:nodo;l:nivel;β:valor):valor
var
   v, α:valor
   i:integer
begin
   α = −∞
   if terminal(X) or l = 0
      α = valor(X)
   else
      i = 1
      while existehijo(X,i) and α < β
         v=B(hijo(X,i),l − 1,α)
         if α < v
            α = v
         endif
         i = i + 1
      endwhile
   endif
   return α
endA
```

Evaluación del valor en un nodo donde mueve B:

```
function B(x:nodo;l:nivel;α:valor):valor
var
   v, β:valor
   i:integer
begin
   β = ∞
```

```
    if terminal(X) or l = 0
        β = valor(X)
    else
        i = 1
        while existehijo(X,i) and β > α
            v=A(hijo(X,i),l − 1,β)
            if β > v
                β = v
            endif
            i = i + 1
        endwhile
    endif
    return β
endB
```

Donde las funciones tienen los siguientes significados:

- *terminal*, devuelve TRUE si el nodo es terminal.
- *valor*, devuelve el valor de un nodo (el valor exacto si es terminal, y la estimación si no es terminal).
- *existehijo*, devuelve TRUE si el nodo tiene hijo i-ésimo.
- *hijo*, devuelve el hijo i-ésimo del nodo actual.

Habría que hacer una primera llamada con A(1,l,∞) para decidir la esperanza de ganar empezando a mover. Si lo que queremos es decidir qué movimiento hacer sería:

```
mover = 0
i=1
while existehijo(X,i)
    v=B(hijo(X,i),l,α)
    if α < v
        α = v
        mover = i
    endif
    i = i + 1
endwhile
```

de modo que en *mover* tenemos el orden del hijo de X para el que se maximizan las posibilidades de ganar, y a continuación se haría ese movimiento. Después de mover el otro jugador (nuestro programa juega como jugador A) llegaríamos a otro nuevo nodo X y habría que volver a ejecutar el mismo código anterior para decidir el siguiente movimiento.

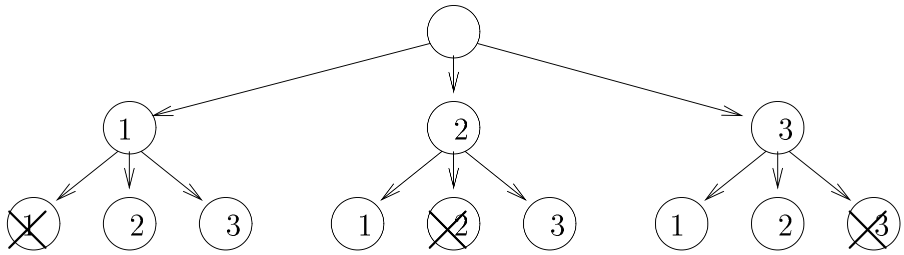
7.5 Problemas

Problema 7.1 Un problema de asignación de trabajos se representa en una tabla T de n filas y n columnas, donde la entrada $T[i,j]$ indica el beneficio que se obtiene de asignar el trabajo i al trabajador j. Los datos en la tabla son números mayores o iguales a cero, indicando un cero que es imposible asignar ese trabajo a ese trabajador. Se trata de resolver el problema de maximizar el beneficio sabiendo que hay que asignar todos los trabajos y que a cada trabajador se le asigna un único trabajo. Resolver el problema por Branch and Bound: indicar cómo sería el árbol de soluciones, cómo se representaría una solución, cómo se calcularían la cota superior, la inferior y la estimación del beneficio, y cómo se harían la ramificación y la poda.

Solución:

La solución la representamos mediante un array s con índices de 1 a n y valores de 1 a n, indicando $s[i] = j$ que al trabajador j se le asigna el trabajo i.

El árbol de búsqueda de las soluciones puede ser un árbol con n niveles teniendo cada nivel n hijos, e indicando el nivel i el trabajo que se está asignando, y el hijo j-ésimo de un nodo a qué trabajador se está asignando. En la figura se representan los dos primeros niveles del árbol con $n = 3$, y los nodos tachados son nodos que se generan (o se evalúan) pero no se generan sus hijos porque no cumplen las condiciones del problema: se asignan dos trabajos a un mismo trabajador:



La cota inferior de un nodo (CI(nodo)) será el beneficio que se lleva con las asignaciones ya realizadas: si vamos por el nivel *nivel* será $\sum_{i=1}^{nivel} T[i, s[i]]$. Para calcular esta cota sin necesidad de hacer la suma en cada nodo podemos llevar una variable CI inicializada a 0 y a la que se suma $T[nivel, s[nivel]]$ cuando se genera el nodo $s[nivel]$ del nivel *nivel*, y a la que se resta $T[nivel, s[nivel]]$ cuando se deshace la asignación.

La cota superior de un nodo (CS(nodo)) puede ser la cota inferir mas la suma de los beneficios máximos de los trabajos aún no asignados: CI(nodo)+$\sum_{i=nivel+1}^{n} max_{j=1,\ldots,n}\{T[i,j]\}$. Los máximos que aparecen en esta expresión se calcularían al principio de la ejecución. Es posible mejorar la CS pero a costa de un trabajo adicional, si tenemos en cuenta que con las asignaciones que se han hecho hasta el nivel *nivel* se hace innecesario tomar el máximo con $j = 1, \ldots, n$, y sólo se puede alcanzar el máximo de los trabajadores a los que no se ha asignado trabajo todavía.

Una estimación obvia del beneficio es la media entre la cota inferior y la superior. Quizás una estimación mejor se puede obtener utilizando un método de avance rápido empezando a asignar desde el nivel $nivel+1$ en adelante al trabajador que nos produzca mayor beneficio.

La ramificación puede ser una LC-FIFO por ejemplo, teniendo en cuenta el tipo de árbol que utilizamos y que no se generan los hijos de nodos que contradigan las condiciones del problema (los nodos tachados en la figura).

La poda se realiza teniendo una variable B que representa el beneficio máximo alcanzable. Cuando en un nodo CS(nodo)$\leq B$ quiere decir que a partir de ese nodo no podemos mejorar la mejor solución que ya llevamos (la de beneficio B) por lo que el nodo se poda. Dado que a partir de un nodo puede no haber solución (y de hecho puede que el problema no tenga solución) B será el máximo beneficio de las soluciones encontradas hasta ese momento, y la poda sólo podrá empezar a hacerse a partir de haber encontrado una solución.

Problema 7.2 En un problema que se resuelve por la técnica Branch and Bound se puede utilizar un método de estimación de la cota inferior y cota superior de cada nodo que supone un coste n por nodo, produciendo una poda de un $t * 50\%$ de los nodos; o se puede utilizar otra técnica de cálculo de las cotas con un coste n^2 por nodo, produciéndose una poda de un $t * 75\%$ de los nodos; donde t es un valor entre 0 y 1 que sirve para indicar con qué velocidad se encuentra una solución óptima (un valor cercano a 1 indica que se encuentra pronto). Si la estimación del beneficio se hace con un coste n en cada nodo el valor de t es $\frac{1}{2}$, y si se hace con un coste n^2 el valor de t es $\frac{1}{4}$. Determinar para los distintos valores de n qué combinación de cálculos de costes y estimación del beneficio es mejor en cuanto a reducción del tiempo de ejecución. (Consideramos que las constantes que afectan a los órdenes de los tiempos que hemos dado son siempre 1).

Solución:

Tenemos dos maneras de calcular las cotas en cada nodo y otras dos de estimar el beneficio, por lo que tenemos un total de cuatro posibilidades:

- caso 1: calcular las cotas con un coste n y estimar el beneficio con un coste n. El coste por nodo será $2n$.
- caso 2: calcular las cotas con coste n y estimar el beneficio con coste n^2. El coste por nodo será $n^2 + n$.
- caso 3: calcular las cotas con coste n^2 y estimar el beneficio con coste n. El coste por nodo será $n^2 + n$.
- caso 4: calcular las cotas con coste n^2 y estimar el beneficio con coste n^2. El coste por nodo será $2n^2$.

Para saber cuál es el coste total en cada uno de los casos hay que considerar el porcentaje de nodos sin podar:

- caso 1: se podan un total de $\frac{1}{2}50 = 25\%$ de los nodos. Quedan sin podar un 75%.
- caso 2: se podan $\frac{1}{4}50 = 12.5\%$ de los nodos. Quedan sin podar un 87.5%.
- caso 3: se podan $\frac{1}{2}75 = 37.5\%$ de los nodos. Quedan sin podar un 62.5%.
- caso 4: se podan $\frac{1}{4}75 = 18.75\%$ de los nodos. Quedan sin podar un 81.25%.

Si N es el número total de nodos del árbol el coste en cada uno de los casos será:

- caso 1: $1.5nN$
- caso 2: $0.875\left(n^2+n\right)N$
- caso 3: $0.625\left(n^2+n\right)N$
- caso 4: $1.625n^2N$

Para decidir cuál de los cuatro casos es mejor no necesitamos considerar el caso 2, pues el caso 3 lo mejora, por lo que compararemos los casos 1, 3 y 4. Dividiendo por n y N lo que nos queda por hacer es comparar las funciones:

- caso 1: 1.5
- caso 3: $0.625(n+1)$
- caso 4: $1.625n$

Las funciones de los casos 4 y 3 se cortan en 0.625, por lo que el caso 3 es mejor que el cuatro a partir de $n = 0.625$. Y las funciones de los casos 1 y 3 se cortan en 1.4, por lo que el caso 1 es preferible a partir de $n = 1.4$.

Resumiendo, el caso 4 es preferible hasta $n = 0.625$, el caso 3 entre $n = 0.625$ y $n = 1.4$, y el caso 1 a partir de $n = 1.4$.

Problema 7.3 Resolver por backtracking el problema de obtener de una serie de números los que suman S con una cantidad mínima de números. Se utilizará un esquema similar al no recursivo visto en clase, con funciones "generar", "mashermanos", "solucion" y "criterio", y se indicará cómo se representan las soluciones y cuál es la condición de fin.

Decir cómo se podrían obtener una cota inferior y otra superior, y cómo se podría estimar el coste para resolverlo con un método Branch and Bound.

Solución:

El problema es de optimización, por lo que habrá que recorrer todo el árbol y la condición de fin será que volvamos al nivel 0 (al nodo raíz).

Llevaremos una solución óptima actual SOA y un valor óptimo actual VOA, donde al final estarán la solución óptima y el valor óptimo del problema. En SOA estarán almacenados los números que forman la solución óptima actual, por lo que inicialmente será el conjunto vacío, y en VOA tendremos el número de números que forman SOA, por lo que incicialmente tendrá el valor $+\infty$, pues estamos en un problema de minimización. Además, llevaremos una variable *suma* donde almacenamos la suma de los números correspondientes a la solución del nodo que vamos examinando, y otra varialbe *num* que contendrá el número de números incluidos en cada momento. Al cambiar de nodo en el árbol lógico habrá que actualizar estas variables.

Hay que decir qué tipo de árbol vamos a utilizar y cómo vamos a representar las soluciones. Consideramos un árbol binario donde en cada nivel se decida si incluir un número o no. De este modo las soluciones se representan con un array de 1 a n de valores de 0 a 1, indicando un 0 en la posición j que no se incluye el j-ésimo número y un 1 que sí se incluye. Además, consideraremos que las soluciones pueden tomar también valor -1, indicando este valor que no se ha tomado todavía ninguna decisión sobre el número. De esta manera, SOA será un array de 1 a n con valores entre -1 a 1, y podemos inicializarlo a 0 ó -1. Si SOA continúa con estos valores al final de la ejecución es que no hay solución.

Un esquema para este problema podría ser:

$SOA \leftarrow -1$
$VOA = +\infty$
$suma = 0$
$nivel = 1$
$s[nivel] = generar(nivel)$
repeat
    if $solucion(nivel)$
        if $mejor(s, VOA)$
            $asignar(s, SOA)$
        endif
    endif
    if $criterio(nivel)$
        $nivel = nivel + 1$
        $s[nivel] = -1$
        $s[nivel] = generar(nivel)$
    elsif $mashermanos(nivel)$
        $s[nivel] = generar(nivel)$
    else
        while not $mashermanos(nivel)$ and $nivel \neq 0$
            $suma = suma - s[nivel]x[nivel]$

$num = num - s[nivel]$
$nivel = nivel - 1$
endwhile
if $nivel \neq 0$
$generar(nivel)$
endif
endif
until $nivel = 0$

Donde las funciones tienen los siguientes significados:

$generar(nivel)$:
$suma = suma + (s[nivel] + 1)x[nivel]$
$num = num + s[nivel] + 1$
return $s[nivel] + 1$

(consideramos que los números están almacenados en un array x)

$solucion(nivel)$:
return $(nivel = n)$ and $(suma = S)$

(sólo pueden ser nodos solución los terminales)

$mejor(s, VOA)$:
return $num < VOA$

$asignar(s, SOA)$:
copia la solución s en SOA
$VOA = num$

$criterio(nivel)$:
return $nivel \neq n$

(podemos seguir hacia abajo en el árbol si no estamos en un nodo terminal. No tiene sentido excluir los nodos cuya suma sobrepase S pues nadie nos ha dicho que los números sean positivos)

$mashermanos(nivel)$:
return $s[nivel] \neq 1$

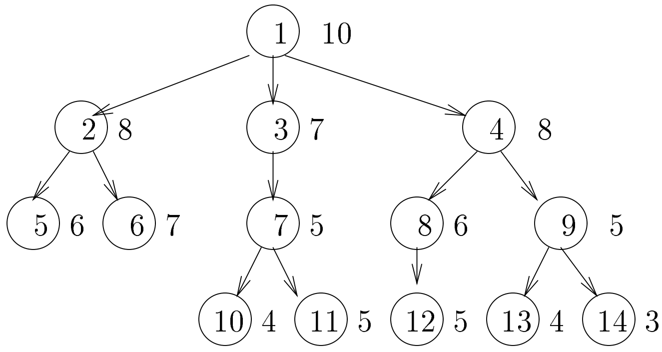
Para resolverlo por Branch and Bound las cotas inferior y superior en cada nodo deben ser cotas del valor que queremos minimizar, por lo que puede ser $CI = num$, y la cota superior puede ser la suma de los números que hemos decidido incluir mas los números sobre los que todavía no hemos decidido nada ($CS = CI + n - nivel$). La estimación del beneficio puede ser la media de las dos cotas (¡está claro que se pueden pensar cotas y estimaciones mejores!). Además, como no estamos seguros de encontrar una solución a partir de un cierto nodo, la poda sólo podría empezar a hacerse después de encontrar una solución...

Problema 7.4 Diseñar un algoritmo por Branch and Bound para resolver el problema de resolver un rompecabezas consistente en un tablero cuadriculado y una serie de piezas formadas por cuadrados, utilizando un número mínimo de piezas.

Problema 7.5 Resolver por Backtracking y Branch and Bound el problema de obtener de un conjunto de números enteros positivos $\{x_1, x_2, \ldots, x_n\}$ todos los subconjuntos que sumen una cantidad dada C.

Problema 7.6 Explicar cómo se podría resolver el problema de la devolución de monedas por Backtracking y por Branch and Bound. En el Branch and Bound, ¿se puede seguir algún criterio para asignar pesos a los nodos del árbol? Considerar el caso en que disponemos de n monedas cuyos valores están almacenados en un array de 1 a n.

Problema 7.7 Dado el árbol de soluciones:



donde el primer número servirá para referirnos al nodo y el segundo en los nodos no terminales representa el máximo valor posible de una solución a partir de ese nodo, y en los nodos terminales el beneficio de la solución, y sabiendo que se pretende maximizar el beneficio. Enumerar, justificándolo, en qué orden se recorren los nodos con los métodos: Backtracking, Branch and Bound FIFO, LIFO, LC-FIFO y LC-LIFO.

Solución:

- En Backtracking se recorre el árbol haciendo en cada nodo el recorrido:

  nodo, backtracking de hijo$_1$ de nodo, backtracking de hijo$_2$ de nodo, ..., backtracking de último hijo de nodo; pero en nuestro caso se pueden eliminar búsquedas que no nos conduzcan a soluciones mejores que las que tenemos. De este modo el recorrido sería:

  1, 2, 5 (tenemos beneficio 6, por lo que no seguiremos por ramas que no mejoren este beneficio)

  6 (beneficio 7)

  3 (no se hace backtracking de sus hijos pues no nos mejoran la solución del nodo 6)

  4 (a partir de aquí puede que mejoremos la solución de beneficio 7, por lo que se generan sus hijos)

  8, 9.

- Branch and Bound FIFO:

  En la técnica Branch and Bound se tiene en cada paso un nodo actual del que se generan todos los hijos, diferenciándose las diversas técnicas en el criterio de elección del siguiente nodo actual entre el conjunto de todos los nodos vivos (son nodos vivos aquellos generados de los que no se han generado todavía los hijos).

  En el caso FIFO la técnica para elegir el siguiente nodo actual es utilizando una cola de nodos vivos. Además, no hay que generar nodos a partir de los cuales no podamos mejorar la solución óptima actual:

nodo generado	cola de nodos vivos
1	1
2, 3, 4	2, 3, 4
5, 6	4

  (5 y 6 no entran en la cola de nodos vivos pues no son nodos terminales. La solución óptima actual es la 6 con beneficio 7. El 3 desaparece de la cola de nodos vivos pues a partir de él no se puede mejorar el beneficio del nodo 6).

  8, 9

  (aquí se acaba pues a partir de 8 y 9, que serían los nodos vivos, no se puede generar ninguna solución mejor que la del nodo 6).

- Branch and Bound LIFO:

  En el caso LIFO la técnica para elegir el siguiente nodo actual es utilizar una pila de nodos vivos. Además, no hay que generar nodos a partir de los cuales no podamos mejorar la solución óptima actual:

nodo generado	pila de nodos vivos
1	1
2, 3, 4	4, 3, 2
8, 9	9, 8, 3, 2
13, 14	8, 3, 2

  (la solución óptima hasta el momento es la del nodo 13, por lo que no recorreremos ramas que no nos puedan mejorar esa solución).

  12 3, 2

  (solución óptima la del nodo 12, beneficio 5).

  7 2

  (a partir del nodo 7 el máximo beneficio que podemos alcanzar es 5, que no mejora la solución de 12).

  5, 6

- Branch and Bound LC-FIFO:

  En el caso LC se tienen en cuenta los costos (en este caso los beneficios) para elegir el nodo a explorar en cada paso, se tendrá una lista de nodos vivos que se mantendrá ordenada de mayor a menor beneficio, y a igualdad de beneficio se utiliza el criterio FIFO.

nodo generado	lista de nodos vivos
1	1
2, 3, 4	2, 4, 3
5, 6	4

  (solución óptima actual la del nodo 7. Se elimina el nodo 3 de la lista pues no puede mejorar el beneficio del nodo 6).

  8, 9

- Branch and Bound LC-LIFO:

  Como LC-FIFO pero en caso de empate se utiliza una técnica LIFO.

nodo generado	lista de nodos vivos
1	1
2, 3, 4	4, 2, 3
8, 9	2, 3, 8, 9

  (el mayor beneficio posible a partir de un nodo de la lista es a partir del nodo 2, por lo que se toma como nodo activo).

  5, 6

  (ninguno de los tres nodos vivos mejora la solución del nodo 6, por lo que se acaba aquí).

Problema 7.8 Dados n programas de longitudes l_1, l_2, ..., l_n y una cinta de longitud L para almacenarlos desde el principio de la cinta un programa a continuación del otro. Diseñar un algoritmo por la técnica Branch and Bound para maximizar el número de programas almacenados. Especificar cómo se calcularían las cotas inferior y superior y la estimación del beneficio en cada nodo y cuáles serían los criterios de poda del árbol de soluciones. Dar el árbol de soluciones para el caso: $l_1 = 4$, $l_2 = 6$, $l_3 = 3$, $l_4 = 4$ y $L = 10$, indicando en qué orden se generan los nodos y los valores de las cotas y de la estimación del beneficio en cada nodo. (Para la ramificación usar el criterio LC-FIFO).

Solución:

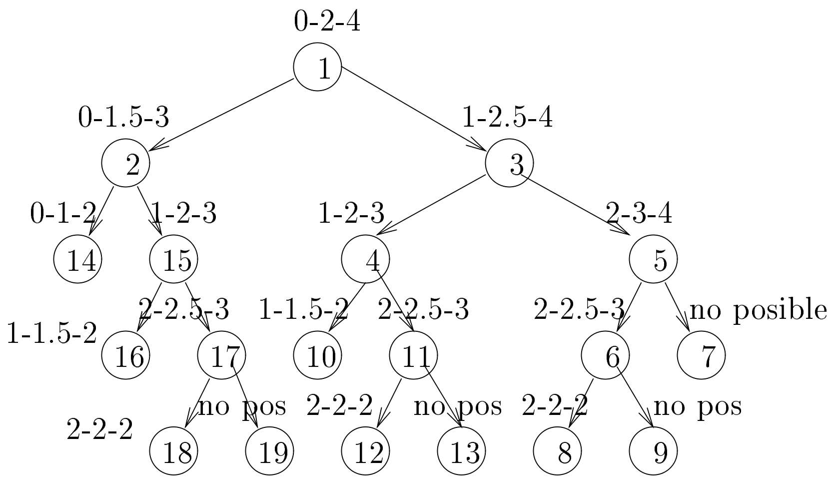
Las soluciones vendrán dadas por una sucesión de 0 y 1 que se almacenarán en un array *solucion* con índices de 1 a n, siendo $solucion[i] = 0$ si no se almacena el programa i-ésimo, y $solucion[i] = 1$ si sí se almacena.

El árbol de soluciones será binario y en cada nivel i (estando el nodo raíz en el nivel 0) se habrán tomado decisiones sobre los programas 1, 2, ..., i.

Como queremos maximizar el beneficio siendo éste el número de programas que se almacenan, la cota inferior de un nodo será el número de unos que hay en la solución parcial (que representa el número de programas que se ha decidido almacenar en la cinta); la cota superior será la inferior mas el número de programas sobre los que todavía no se ha decidido nada; y el beneficio estimado será la media entre estas dos cotas.

Para la poda del árbol se utiliza una variable global BG que en cada momento contendrá la mayor de las cotas inferiores de los nodos evaluados. Cuando en un nodo i sea $CS(i) \leq BG$ no es necesario generar sus descendientes. Además, hay que evaluar sólo nodos que nos lleven a soluciones posibles, por lo que cuando las sumas de las longitudes se los programas con $solucion[i] = 1$ sea mayor que L tampoco hay que generar a partir de ese nodo.

Para el ejemplo dado el árbol quedaría:



donde dentro de cada nodo aparece el orden en que se genera, arriba aparecen la cota inferior, el beneficio estimado y la cota superior, y los nodos a la izquierda corresponden a $solucion[i] = 0$ y los de la derecha a $solucion[i] = 1$.

Problema 7.9 Supuesto que ya tenemos decidido cuáles son los n programas que podemos almacenar en la cinta se trata de distribuirlos en ésta de manera que el tiempo medio de acceso desde el principio de la cinta sea mínimo (si tenemos los programas en el orden l_1, l_2, ..., l_n, el tiempo de acceso a l_1 es 0, a l_2 es proporcional a l_1, a l_3 es proporcional a $l_1 + l_2$, etc...). La información sobre los programas la tenemos en una lista donde los nodos tienen la forma:

nodo=record
  identificador: nombre del programa
  orden: entero (indica el orden en que se almacena en la cinta)

longitud: entero
programa: texto del programa
siguiente: puntero a nodo
endrecord

Programar un algoritmo con complejidad temporal del orden de $n \log n$ que minimice el tiempo de acceso, demostrar que encuentra una solución óptima y calcular su tiempo de ejecución. El programa tiene como finalidad llenar con los valores apropiados el campo orden de los nodos de la lista que no se deberá modificar por ningún otro concepto.

Solución:

El tiempo medio de acceso suponiendo que hay la misma probabilidad de acceder a cada uno de los programas es:

$$\frac{(n-1)l_1 + (n-2)l_2 + \ldots + l_{n-1}}{n}$$

y para minimizarlo basta minimizar el numerador.

Parece lógico que una distribución óptima se obtiene cuando $l_1 \leq l_2 \leq \ldots \leq l_n$. Para demostrarlo, suponemos una distribución con $l_i > l_j$ siendo $i < j$, y demostraremos que no es óptima pues intercambiando l_i y l_j de sitio se obtiene otra distribución mejor.

El término i-ésimo del sumatorio es $(n-i)l_i$ y el término j-ésimo es $(n-j)l_j$, y en la distribución obtenida poniendo l_i en el lugar j y l_j en el lugar i estos términos son $(n-i)l_j$ y $(n-j)l_i$, siendo los demás términos del sumatorio iguales para las dos distribuciones, pero $(n-i)l_i+(n-j)l_j > (n-i)l_j+(n-j)l_i$, pues $-il_i-jl_j > -il_j-jl_i$, ya que $(j-i)l_i > (j-i)l_j$, al ser $l_i > l_j$; por lo que la distribución inicial no es óptima.

Por lo tanto, el problema consiste en ordenar los programas en orden creciente de longitud, y esto hay que hacerlo modificando el campo orden de cada nodo. La ordenación se puede hacer con un coste $n \log n$ por ejemplo con el método mergesort, pero el obtener las longitudes y los nodos y poner los órdenes en el campo orden no nos debe superar el tiempo $n \log n$. Obtener las longitudes y almacenarlas en un array nos supone un tiempo n pues sólo hay que recorrer la lista una vez, pero para actualizar los valores del campo orden con un orden n necesitamos poder acceder a cada nodo sin recorrer la lista, por lo que, junto con la longitud, almacenaremos un puntero al nodo correpsondiente.

Los datos los almacenamos en un array:
a:array[1..n] of $datos$
$datos$:record
$long$:entero
$lugar$:pointer to nodo
endrecord

El programa sería:
leer

ordenar(1,n)
actualizar
donde en "leer" inicializamos a:
$p = lista$ (siendo $lista$ el puntero al principio de la lista)
$i = 1$
while $p \neq nil$
$a[i].long = p- > longitud$
$a[i].lugar = p$
$p = p- > siguiente$
endwhile
en "actualizar" modificaríamos el campo $orden$:
for $i = 1$ to n
$a[i].lugar- > orden = i$
endfor
y en "ordenar" se haría la ordenación de a por el método mergesort:
procedure ordenar(p, q:enteros)
if $p < q$
$m = (p + q)\ div\ 2$
ordenar(p, m)
ordenar($m + 1, q$)
mezclar(p, m, q)
endif
siendo "mezclar":
$i = p$
$j = m + 1$
$k = p$
while $i \leq m$ and $j \leq q$
if $a[i].long \leq a[j].long$
$b[k] = a[i]$ (se copia el registro entero siendo b del mismo tipo que a)
$i = i + 1$
else
$b[k] = a[j]$
$j = j + 1$
endif
$k = k + 1$
endwhile
if $i \leq m$
for $l = i$ to m
$b[k] = a[l]$
$k = k + 1$
endfor

```
else
    for l = j to q
        b[k] = a[l]
        k = k + 1
    endfor
endif
for l = p to q
    a[l] = b[l]
endfor
```

El orden de "leer" es n pues se pasa n veces por el cuerpo del while que es constante. El de "actualizar" también es n pues es un for de 1 a n con cuerpo constante.

Problema 7.10 Resolver por Branch and Bound el problema de la devolución de monedas suponiendo que tenemos en un array de índices de 1 a n los valores de las monedas de que disponemos, y que disponemos de un número ilimitado de monedas de cada uno de los n tipos. Se pide explicar cómo sería el árbol de soluciones, cómo se representarían las soluciones, cómo se calcularían la cota superior, la inferior y la estimación del beneficio en cada nodo y los procedimientos de poda y ramificación.

Solución:

El árbol tendría profundidad n, indicándose en el nivel i-ésimo la cantidad de monedas del tipo i que se dan, y en cada nivel el número de hijos de un nodo será $\lfloor \frac{C}{a[i]} \rfloor + 1$, siendo C la cantidad que queda por dar y $a[i]$ el valor de la moneda i-ésima si vamos por el nivel i.

Una solución se representará en un array de 1 a n de enteros indicando en el lugar i-ésimo la cantidad de veces que se da la moneda i-ésima.

La CI puede ser la cantidad de monedas que se han dado hasta ese momento y la $CS = CI + \frac{C}{min_{i>nivel} a[i]}$, que es la cantidad de monedas que se llevan dadas mas las que habría que dar suponiendo que las que quedan por dar hay que darlas todas de las de menor valor de las que quedan (por eso hacemos el mínimo con $i > nivel$).

La estimación del coste podría ser la media de CI y CS.

El criterio de ramificación si usamos un LC-FIFO, como queremos minimizar el coste sería por la de menor beneficio estimado y a igualdad de beneficio por el primero generado. Esto influye sólo en la gestión de la lista de nodos vivos pues la ramificación se hace siempre tomando el primero de la lista.

La poda se hará cuando $CI(i) \geq min_{soluciones\ generadas} Coste$, por lo que se necesita una variable CG donde se almacene este mínimo. Hay que tener en cuenta que sólo se puede podar cuando se ha llegado a una solución, pues puede ocurrir que a partir de un nodo no tengamos solución y, por tanto, CS en cada nodo es una cota superior de una posible solución a partir de ese nodo.

Problema 7.11 Dado un grafo multietapa con n nodos, uno de ellos (que llamaremos origen) en el primer nivel, otro (que llamaremos destino) en el último nivel, y l niveles intermedios con $\frac{n-2}{l}$ nodos en cada uno de los niveles intermedios.

Resolver por avance rápido el problema de encontrar el camino de longitud mínima del origen al destino. Estudiar el tiempo de ejecución del algoritmo.

Indicar al menos dos maneras en las que el avance rápido se puede usar en este problema para reducir el número de nodos generados cuando se resuelve el problema por backtracking o branch and bound. Poner ejemplos que ilustren los métodos indicados.

Solución:

Los nodos estarán numerados de 1 a n, siendo el nodo origen el 1 y el destino el n. Consideramos los pesos de las aristas almacenados en una matriz de adyacencia (d), donde un ∞ indicará que no existe la arista.

En un método de avance rápido se dan una serie de pasos que parecen óptimos en cada momento. En cada momento se tomará de entre todas las aristas que salen del nodo en el que estamos la arista de menor coste. El número de pasos será $l+1$, partiendo del nodo 1, y se utilizará un array $s : array[0,..,l+1]\ de\ 1,..n$ para indicar a qué nodo se llega desde el nodo actual. $s[0] = 1$ para indicar que se parte del nodo 1. En los l primeros pasos hay que obtener la mínima de $\frac{n-2}{l}$ aristas, y en el último paso sólo hay una posible arista, por lo que el número de pasos se reducirá a l, y $s[l+1] = n$. Además, llevaremos una variable L donde se almacenará la longitud del camino que se va formando, por lo que estará inicializada a 0 y al final tendrá el valor del camino obtenido, que puede no ser el óptimo y que incluso puede que no encontremos camino, con lo que L tendrá el valor ∞. El esquema del algoritmo sería:

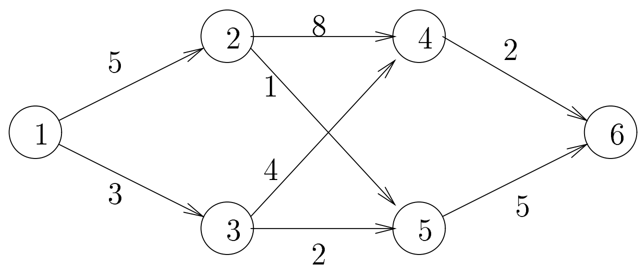
```
L = 0
s[0] = 1
s[l + 1] = n
for i = 1 to l
    min = d[s[i − 1], 2 + (n − 2)(i − 1)/l]
    s[i] = 2 + (n − 2)(i − 1)/l
    for j = 3 + (n − 2)(i − 1)/l to 2 + (n − 2)i/l
        if d[s[i − 1], j] < min
            min = d[s[i − 1], j]
            s[i] = j
        endif
    endfor
    L = L + d[s[i − 1], s[i]]
endfor
L = L + d[s[l], n]
```

El primer bucle se utiliza para realizar los l pasos, y el segundo para obtener la arista de longitud mínima desde el nodo en el que estamos. No es necesario evaluar todos los nodos sino sólo los del siguiente nivel. Si no hay ninguna arista desde el nodo actual se tomará como destino el primer nodo del siguiente nivel, y L tomará el valor ∞. Estamos considerando que tiene sentido sumar valores infinito. Se podría acabar en cuanto *min* saliera del bucle interior con valor ∞, pues en este caso no encontramos el camino que estamos buscando. Esto implicaría cambiar el for externo por un while. Al acabar el for externo se actualiza L pues, como hemos dicho, sólo hay una posible arista desde el nodo actual al destino.

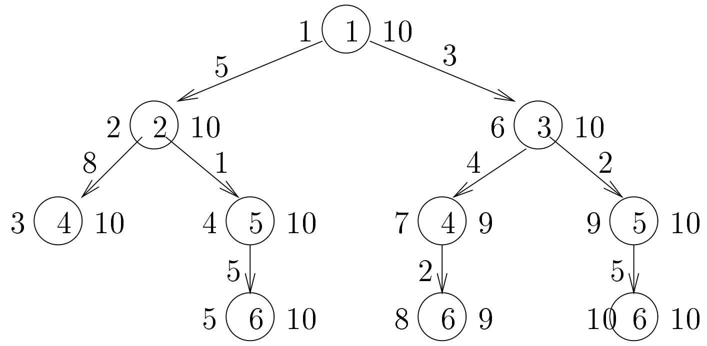
En un esquema general de avance rápido se tiene una función *selección* que en nuestro caso corresponde al bucle interno. La función de comprobación de si es válida la selección que hemos hecho, no aparece en nuestro esquema pues la decisión que tomamos la consideramos siempre válida, y que si decidimos evitar el seguir generando caminos inexistentes consistiría en comprobar si el valor de min es ∞. La función de *añadir* la decisión a la solución es la actualización de s.

Para estudiar el tiempo de ejecución, dado que el algoritmo consta de l pasos, en cada uno de los cuales se obtiene un mínimo de $\frac{n-2}{l}$ datos, el tiempo será del orden de $l\frac{n-2}{l} \in \theta(n)$.

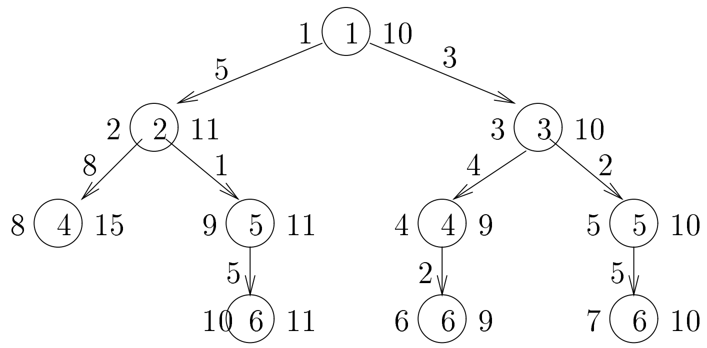
En un backtracking se puede usar el método de avance rápido para obtener una solución, y la función criterio determinaría si en un nodo la longitud del camino hasta el nodo es mayor que la longitud del camino encontrado por avance rápido, en cuyo caso no se cumple el criterio y se evitaría generar algunos nodos. Como ejemplo vemos el siguiente grafo:



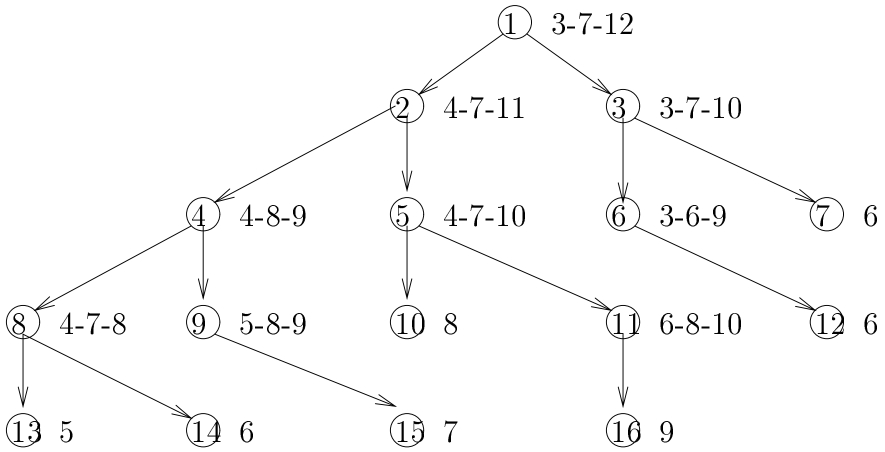
cuyo recorrido por backtracking se muestra en el árbol siguiente, donde dentro de cada nodo se pone el número de nodo a que corresponde en el grafo, en cada arista del árbol se pone al peso de la arista correspondiente en el grafo, y al lado de cada nodo aparecen dos números, indicando el que está a la izquierda el orden en que se recorren los nodos del árbol, y el que está a la derecha la longitud del camino encontrado por avance rápido. En el nodo uno la longitud del camino es 10, y en el nodo dos no se actualiza la longitud del camino pues aplicando avance rápido a partir de ese nodo se obtiene un camino de longitud 11, que no mejora la longitud asociada al nodo uno. Se comprueba que se poda un único nodo.



En un Branch and Bound se puede utilizar el método de avance rápido para estimar en cada nodo el coste de un camino a partir de ese nodo; además, este valor sirve como cota superior, y como cota inferior se puede utilizar la longitud del camino recorrido hasta llegar al nodo. El avance rápido serviría, por tanto, para guiar la búsqueda, y la poda se haría como se ha visto en el backtracking con la función *criterio*. En la gráfica siguiente se muestra como funcionaría el método con el grafo ejemplo. Los número significan lo mismo que en el árbol del backtracking, menos el valor a la derecha de un nodo, que es la estimación del coste (y la cota superior). También en este caso se poda un nodo.



Problema 7.12 Dado el árbol de búsqueda:

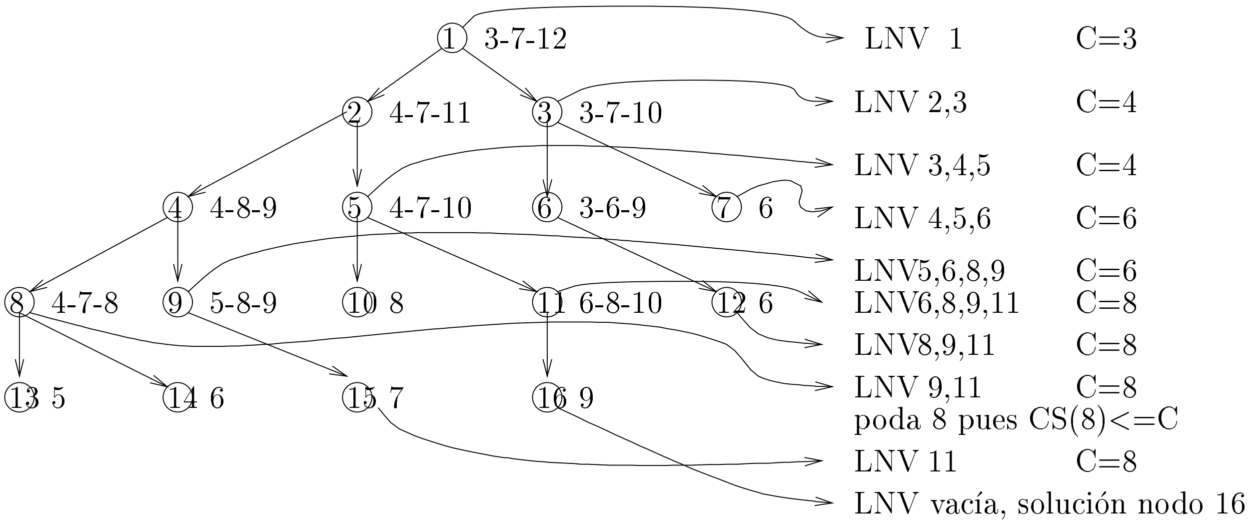


donde al lado de cada nodo aparecen la cota inferior, la estimación del beneficio, y la cota superior estimadas para el nodo; explicar cómo se recorrería el árbol utilizando una técnica Branch and Bound con los métodos FIFO, LIFO, LC-FIFO y LC-LIFO. Habrá que indicar el orden en que se recorren los nodos y el estado de la Lista de Nodos Vivos a lo largo de la ejecución.

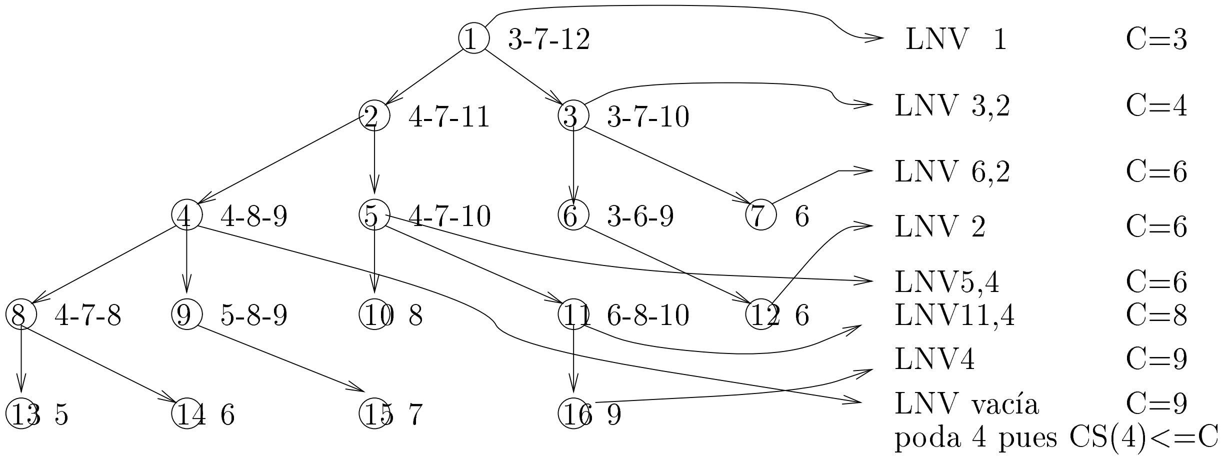
Solución:

Mostraremos la evolución indicando el orden en que se rerorren los nodos y cómo queda la lista de nodos vivos después de cada generación de hijos de un nodo. También indicaremos el valor de la variable global C donde almacenamos la mayor de las cotas inferiores (pues queremos maximizar el beneficio). Consideraremos que a partir de un nodo siempre hay solución, con lo que las podas se pueden hacer desde el principio.

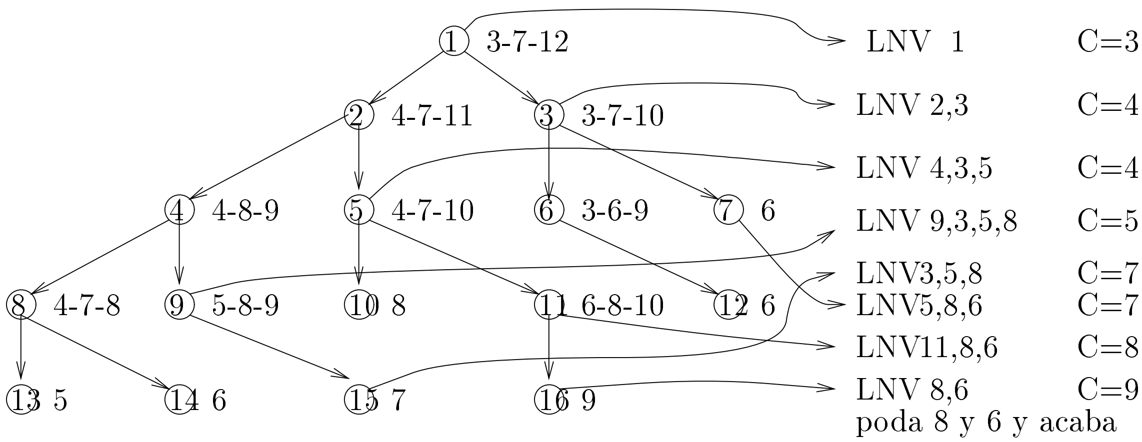
FIFO:



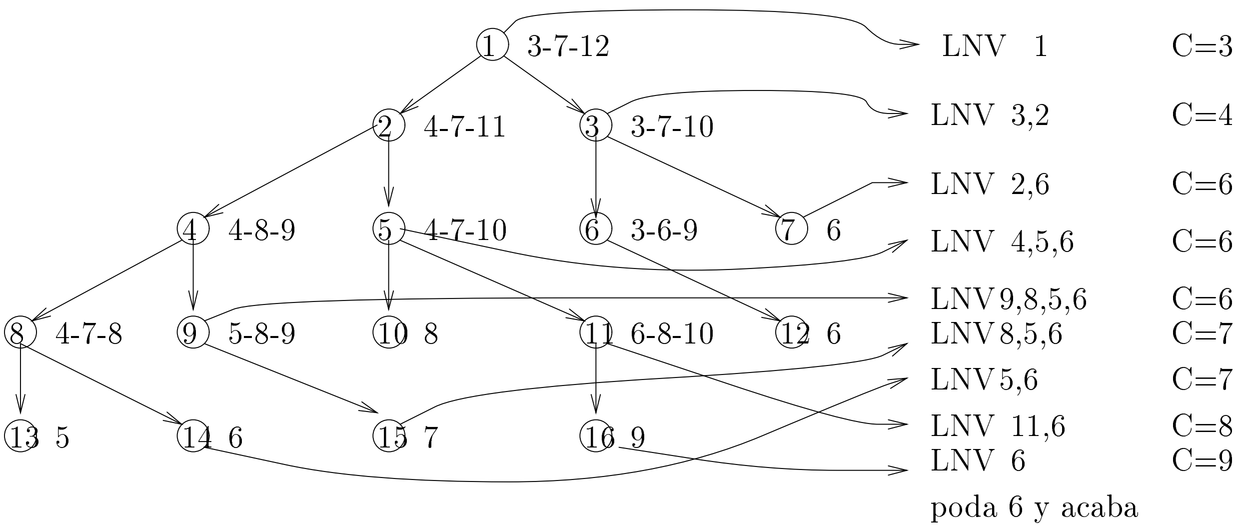
LIFO:



LC-FIFO:



LC-LIFO:



Problema 7.13 Programar la resolución del problema del paseo del caballo por backtracking. Dar el esquema que habría que utilizar, indicar cuál es la condición de fin, y cómo son las funciones "generar", "mas_hijos", "criterio" y "solucion".

Indicar si se podría resolver el problema por Branch and Bound: indicando cómo sería el árbol de soluciones, cómo se representaría una solución, cómo se calcularían la cota superior, la inferior y la estimación del beneficio en cada nodo, y cómo se haría la ramificación y la poda.

Solución:

Como estamos buscando una única solución se acabará cuando se encuentre la primera, pero como no podemos estar seguros de si hay solución también se acabará cuando se recorra todo el árbol volviendo al nodo raíz, en cuyo caso el problema no tiene solución. Según esto el esquema del algoritmo podría ser:

$nivel = 1$
generar($nivel$)
repetir
  si solucion($nivel$)
    fin=verdad
  en otro caso si criterio($nivel$)
    $nivel = nivel + 1$
    generar($nivel$)
  en otro caso si mas_hermanos($nivel$)
    generar($nivel$)
  en otro caso
    mientras (no mas_hermanos($nivel$)) y ($nivel \neq 0$)
      $nivel = nivel - 1$
    finmientras

si $nivel \neq 1$
generar($nivel$)
finsi
finsi
hasta fin o $nivel = 0$

El árbol tendrá n^2 niveles (y la raíz) si consideramos el tablero cuadrado de dimensión $n \times n$, aunque también podría no ser cuadrado el tablero, en cuyo caso el número de niveles es nm, con n el número de filas y m el de columnas. Por simplicidad vamos a considerar el tablero cuadrado.

Cada nodo tendrá 8 hijos, correspondiendo a los 8 posibles movimientos del caballo. Algunos de los movimientos sacarían al caballo del tablero o lo llevarían a una casilla ya visitada, con lo que la función criterio se ocupará de eliminar estos nodos. Usaremos un array $tablero[1..n, 1..n]$ con valores 0 o 1, indicando un cero que no se ha visitado esa posición y un 1 que sí se ha visitado. Inicialmente todas las posiciones estarán a cero.

El array solucion será $s\,[1..n^2]$, donde cada valor $s[i]$ será una terna indicando el primer elemento el movimiento realizado por el caballo (un valor entre 1 y 8 indica un movimiento y un 0 que no se ha realizado movimiento todavía) y los elementos segundo y tercero de la terna indican la posición a que va a parar el caballo en el tablero. A los tres elementos les llamaremos $s[i].m$, $s[i].x$ y $s[i].y$. Si inicialmente el caballo está en la posición (x, y) la generación de $s[1]$ consistirá en hacer $s[1] = (0, x, y)$, por lo que el generar externo al repetir en el esquema debe sustituirse por esa asignación. Además, el resto de los elementos de s se inicializarán a (0,0,0) fuera del repetir. También la variable fin debe inicializarse a falso.

Un nodo cumplirá el criterio cuando la posición que representa no esté fuera del tablero ni haya sido ya visitada:

criterio($nivel$):
devolver $(0 < s[nivel].x < n + 1$ y $0 < s[nivel].y < n + 1$ y
$tablero[s[nivel].x, s[nivel].y] = 0)$

Un nodo será solución cuando se hayan recorrido las n^2 posiciones del tablero habiendo llegado a un nodo que cumple el criterio:

solucion($nivel$):
devolver ($nivel = n^2$ y criterio($nivel$))

Un nodo tendrá más hermanos si todavía no se han ensayado los ocho posibles movimientos que se podían hacer desde el nodo padre:

mas_hermanos($nivel$):
devolver $s[nivel].m < 8$

La única función que es un poco más complicada es la función generar. Se generará el número de movimiento que se va a hacer sumando uno al valor que tuviera, por lo que debe estar inicializado a cero; a partir de ese número de movimiento se generarán las posiciones del tablero a las que se llega haciendo ese movimiento a partir de la posición del nodo padre:

generar($nivel$):
$s[nivel].m = s[nivel].m + 1$
case $s[nivel].m$
1: $s[nivel].x = s[nivel-1].x + 2$
$s[nivel].y = s[nivel-1].y - 1$
2: $s[nivel].x = s[nivel-1].x + 1$
$s[nivel].y = s[nivel-1].y - 2$
3: $s[nivel].x = s[nivel-1].x - 1$
$s[nivel].y = s[nivel-1].y - 2$
4: $s[nivel].x = s[nivel-1].x - 2$
$s[nivel].y = s[nivel-1].y - 1$
5: $s[nivel].x = s[nivel-1].x - 2$
$s[nivel].y = s[nivel-1].y + 1$
6: $s[nivel].x = s[nivel-1].x - 1$
$s[nivel].y = s[nivel-1].y + 2$
7: $s[nivel].x = s[nivel-1].x + 1$
$s[nivel].y = s[nivel-1].y + 2$
8: $s[nivel].x = s[nivel-1].x + 2$
$s[nivel].y = s[nivel-1].y + 1$
fincase
si criterio($nivel$)
$tablero[s[nivel].x, s[nivel].y] = 1$
finsi

(los movimientos se pueden numerar de esta manera o de cualquier otra). Se pone a cero la posición del tablero que se estaba ocupando antes y a uno la que se pasa a ocupar tras el movimiento, pero comprobando que la posicion está dentro del tablero.

Por último, hay que tener en cuenta que antes de retroceder en el árbol hay que poner $s[nivel].m = 0$ para que al llegar de nuevo a ese nivel pero desde otro nodo se vuelvan a intentar los ocho movimientos. Además hay que liberar la posición del tablero que se estaba ocupando. El cuerpo del while quedará:

$s[nivel].m = 0$
$tablero[s[nivel].x, s[nivel].y] = 0$
$nivel = nivel - 1$

Para resolverlo por Branch and Bound hay que tener en cuenta que se pueden eliminar nodos si no cumplen el mismo criterio que se exige en el backtracking, y que en este caso se generan todos los hijos de un nodo incluyendo en la lista de nodos vivos los que cumplen el criterio.

Como no es un problema de optimización sino de encontrar una única solución, el esquema del Branch and Bound no tiene mucho sentido y las cotas inferiores y superiores de un nodo no solución pueden ser $CI = -\infty$ y $CS = +\infty$ indicando estos valores números suficientemente pequeños y grandes, y en un nodo solución $CI = CS = +\infty$, con lo que no se podarán nodos hasta llegar a uno solución, en cuyo caso se podarán todos pues no hace falta seguir buscando una solución.

Más sentido tiene la ramificación en este ejemplo, pues nos debe guiar la búsqueda de manera que se examinen primero los nodos que parezcan más prometedores. Para esto podemos usar el esquema de Warsdorff y considerar que son más prometedores nodos a los que se puede llegar desde menos posiciones no recorridas todavía. Si lo consideramos como un problema de maximización la estimación del beneficio sería $EB = 8 - p$, siendo p el número de posiciones no recorridas a las que se puede llegar desde ese nodo.

También se podría tener en cuenta que puede ser más prometedor seguir por nodos de niveles más profundos pues en este caso puede que estemos más cerca de la solución. De este modo la estimación del beneficio podría ser $EB = 8 - p + nivel$, u otra fórmula donde la varialbe $nivel$ aumente la estimación del beneficio.

Problema 7.14 Dados n números naturales $S = \{x_1, x_2, ..., x_n\}$ y un número natural N. Queremos encontrar el subconjunto de S ($T \subset S$, $T = \{y_1, y_2, \ldots, y_m\}$) que minimice $|N - y_1 - y_2 - \ldots - y_m|$.

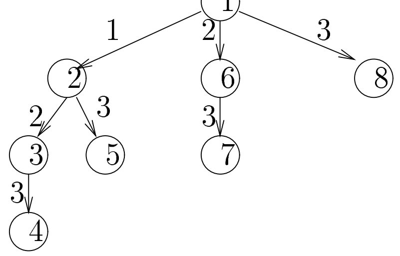
a) Resolver el problema por backtracking utilizando un esquema similar a los vistos en clase, programando las funciones "criterio", "mashijos" (o "mashermanos"), "generar" y "solucion".

b) Indicar cómo se podría resolver por Branch and Bound, indicando cómo se calcularían la cota inferior y superior y la estimación del beneficio en cada nodo, cómo se haría la ramificación y la poda.

Solución:

a) El problema es de optimización, por lo que habrá que recorrer el árbol lógico de soluciones quedándonos con la óptima en cada momento, y se acabará cuando se vuelva al nodo raíz, o cuando la función objetivo sea cero ($N = y_1 + y_2 + \ldots + y_m$).

La solución se representará con un array $s[0..n]$ con valores entre 0 y n, indicando $s[i] = j$ que se toma x_j, y un valor $s[i] = 0$ que no se toma ningún número, y si $s[i] = 0$ (con $i > 0$) también será $s[j] = 0$ si $j > i$. Con esta representación de las soluciones el árbol tiene la forma:



Se necesitará un array auxiliar $so[0..n]$ para almacenar la solución óptima actual. Este array estará inicializado a 0.

También usaremos una variable auxiliar vo que contendrá el valor de la solución óptima actual, y que estará inicializado a ∞ ya que estamos en un problema de minimización.

En cada nodo habrá que calcular un valor $|N - x_{s[1]} + x_{s[2]} + \ldots + x_{s[nivel]}|$ si el nodo está en el nivel indicado por la variable $nivel$. Este valor se comparará con vo para determinar si la solución representada por el nodo es mejor que la óptima actual. Este valor se puede calcular en cada nodo o utilizar otra variable auxiliar (va) que contenga $N - x_{s[1]} + x_{s[2]} + \ldots + x_{s[nivel]}$ y que habrá que actualizar al pasar de un nodo a otro. No se puede incluir el valor absoluto en esta variable pues no se podría actualizar sumando o restando los valores.

Las funciones pueden tener la siguiente forma:

- La función criterio devolverá true cuando a partir de un nodo se pueda encontrar una solución que mejore la actual. Al ser los números todoa naturales, cuando $va < 0$ el añadir números implicará disminuir su valor y aumentarlo en valor absoluto, con lo que cualquier solución a partir de ese nodo será peor que la del nodo donde estamos. También hay que comprobar si estamos en un nodo terminal. Cuando $vo = 0$ tampoco hará falta seguir, pero esto se comprobará cada vez que se modifique so. Tendremos:

  criterio($nivel$):

  return($va > 0$ and $s[nivel] < n$)

- Todos los nodos salvo los terminales (y el raíz) tienen más hermanos, con lo que la función
  mas_hermanos será:

  mas_hermanos($nivel$):

  return($s[nivel] < n$)

- La función solucion siempre vale true pues cualquier nodo representa una posible solución, aunque habrá que comparar con la solución óptima actual para comprobar si la mejora:

  solucion($nivel$):

  return true

- La función generar, en la generación del primer hijo de un nodo debe poner en s el valor del nodo padre mas 1 (por eso necesitamos la posición 0 del array s inicializada a 0), y si no es el primer hijo debe sumar uno al valor actual. Además, como se cambia de nodo en el recorrido del árbol lógico, habrá que actualizar la varialbe v:

```
generar(nivel):
    if s[nivel] = 0
        aux = s[nivel - 1] + 1
    else
        va = va - x_{s[nivel]}
        aux = s[nivel] + 1
    endif
    va = va + x_{aux}
    s[nivel] = aux
```

Con las variables auxiliares y las funciones descritas el esquema del algoritmo podría ser:

```
Backtracking(n):
    nivel = 0
    s ← 0
    so ← 0
    vo = ∞
    va = N
    fin=false
    repeat
        if solucion(nivel)     /*Podría no ponerse pues es true*/
            if |va| < vo
                so ← s
                vo = |va|
                if vo = 0
                    fin=true
                endif
            endif
        endif
        if criterio(nivel)
            nivel = nivel + 1
            generar(nivel)
        elsif mas_hermanos(nivel)
            generar(nivel)
        else
            while ((nivel > 0) and (not mas_hermanos(nivel)))
                va = va - x_{s[nivel]}
                s[nivel] = 0
                nivel = nivel - 1
```

endwhile
if ($nivel = 0$)
fin=true
else
generar($nivel$)
endif
endif
until fin

b) Para resolverlo por Branch and Bound, dado que estamos minimizando un coste, la estrategia de ramificación puede ser LC-FIFO o LC-LIFO, y como no podemos predecir qué suma va a estar más cercana a N tampoco podemos saber si una estrategia es preferible a la otra.

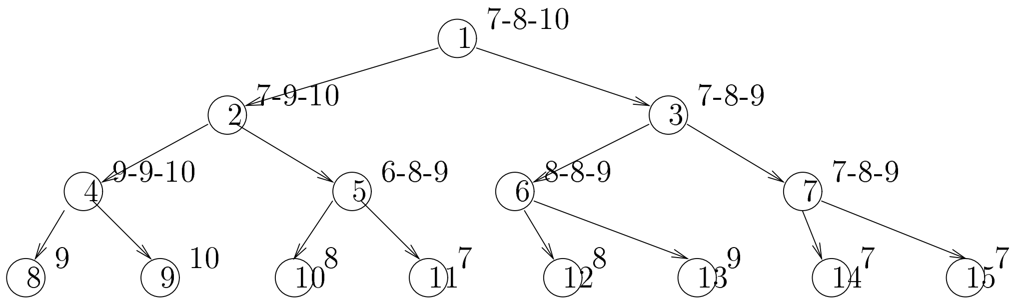
En un nodo, lo peor que puede pasar es que no tomemos ningún número más, y por tanto una cota superior del coste será $CS(nodo) = |N - \sum_{i=1}^{nivel} x_{s[i]}|$.

Para la cota inferior podemos tomar la cota superior cuando el valor en ese nodo (la variable va del apartado anterior) sea negativo, o el valor 0 en otro caso, pues no sabríamos a partir de ese nodo hasta qué valor llegaríamos, a no ser recorriendo sus hijos.

La estimación del coste puede ser la media de CI y CS, con lo que en este caso sería $EC = \frac{CS+CI}{2}$. El coste también se puede estimar por avance rápido tomando números a partir del que estamos mientras el añadir un número nos disminuya el valor de la función objetivo.

Se podará un nodo cuando $CI(nodo) \leq min_{i \in Nodosgenerados}\{CS(i)\}$ ya que todos los nodos representan una solución.

Problema 7.15 Se quiere resolver por Branch and Bound un problema representado por el árbol:



donde al lado de cada nodo hay tres números que indican la cota inferior, la estimación del beneficio y la cota superior, y al lado de los nodos terminales hay un número que representa el valor de la solución asociada a ese nodo. Si el problema es de maximizar el beneficio, indicar cómo se recorrería el árbol y cómo quedaría la lista de nodos vivos y la variable global utilizada para podar en cada paso del recorrido utilizando una técnica LC-FIFO (en nuestro caso no es menor coste sino máximo beneficio) en los casos:

a) Que se sabe que apartir de cada nodo hay solución, por lo que las cotas inferiores y superiores son cotas de una solución óptima a partir del nodo.

b) Que no se sabe si a partir de un nodo hay solución o no, por lo que las cotas lo son de una solución óptima si es que existe solución a partir del nodo.

Solución:

a) Se podará un nodo cuando su $CS(nodo) \leq max\{CI(nodos\ generados)\}$ pues sabemos que a partir de cualquier nodo hay solución. La lista de nodos vivos y el valor de la variable C donde guardamos el máximo de las cotas inferiores queda en cada paso:

- LNV: 1. C=7.
- Generación del nodo 2. C=7.
- Generación del nodo 3.

  LNV: 2,3. C=7.

  El 2 está antes que el 3 en la lista porque su estimación del beneficio es mayor.
- Generación del nodo 4. C=9.
- Generación del nodo 5, que se poda porque $CS(5) = C$.

  LNV: 4,3. C=9.
- Generación del nodo 8 que no se pone en la lista pues es terminal.
- Generación del nodo 9. No se pone en la lista porque es terminal. Se actualiza C.

  LNV: 3. C=10.
- Se toma el nodo 3 de la LNV, y se poda porque $CS(3) < C$.
- El algoritmo acaba con valor 10 y nodo solución el 9 que es el último con que se actualizó C.

b) En este caso las cotas lo son de posibles soluciones, pero sólo se sabe si hay solución a partir de un nodo si se ha llegado a sus nodos terminales. No se podrá podar nodos por el mismo criterio anterior, sino que se podará un nodo cuando su $CS(nodo) \leq max\{valor(nodos\ solución\ generados)\}$. Para que no se pode antes se inicializará C a 0, y se actualizará sólo con los nodos solución. La lista de nodos vivos y el valor de la variable C donde guardamos el máximo de los valores de los nodos solución queda en cada paso:

- LNV: 1. C=0.
- Generación del nodo 2. C=0.
- Generación del nodo 3.

  LNV: 2,3. C=0.

  El 2 está antes que el 3 en la lista porque su estimación del beneficio es mayor.
- Generación del nodo 4. C=0.
- Generación del nodo 5. A diferencia del apartado a), en este caso no se puede podar, y se pone después del nodo 3 pues a igualdad de estimación del beneficio utilizamos el criterio FIFO.

  LNV: 4,3,5. C=0.
- Generación del nodo 8 que no se pone en la lista pues es terminal. C queda con valor 9.
- Generación del nodo 9. No se pone en la lista porque es terminal.

  LNV: 3,5. C=10.
- Se toma el nodo 3 de la LNV, y se poda porque $CS(3) < C$.
- Se toma el nodo 5 de la LNV, y se poda porque $CS(5) < C$.
- El algoritmo acaba con valor 10 y nodo solución el 9 que es el último con que se actualizó C.

Problema 7.16 Se quiere resolver el problema de, dado un damero rectangular de dimensiones $n \times m$ y una serie de p piezas rectangulares de dimensiones $n_1 \times m_1, \ldots, n_p \times m_p$, obtener una asignación de las piezas al tablero con la mayor cantidad de piezas posibles que cumpla: no se salga ninguna porción de una pieza fuera del tablero y no se solapen las piezas.
a) Resolver el problema por Backtracking según un esquema no recursivo de los vistos en clase. Habrá que decir las estructuras de datos que se utilizan, cómo se representan las soluciones, cuál sería el árbol de soluciones que se recorre, cuál es la condición de fin, programar las distintas funciones que aparecen en el esquema, y dar el esquema tal como quedaría para este problema.
b) Indicar alguna estrategia para resolver el problema por Branch and Bound.

Solución:
a) Suponemos que tenemos el tipo de datos pares que es un registro con dos campos x e y con valores enteros.

Las dimensiones de las p piezas estarán almacenadas en un array $piezas$:array[1..p] de pares.

La solución que se va examinando se almacenará en un array s:array[1..p] de pares, donde $s[i]$ indica la posición de la pieza i en el tablero, tomándose siempre la posición tomando como referencia el cuadrado superior izquierda de la pieza. Una solución vendrá dada por las posiciones de todas las piezas en el tablero, y se indicará que la pieza i no se introduce con el valor cero en $s[i].x$. Un valor -1 en esa posición indicará que no se ha tomado todavía ninguna decisión sobre esa pieza, por lo que inicialmente s tendrá inicializados todos los campos x a -1. Tendremos otro array so donde se almacena la solución óptima hasta ese momento. Estará inicializado como s.

La ocupación del tablero vendrá dada por un array t:array[1..n,1..m] de 0..1. Un cero indicará que ninguna pieza está ocupando esa posición y un 1 que sí está ocupada. Todas las posiciones estarán inicializadas a cero.

Tendremos dos variables que indicarán el número de piezas situadas en el tablero: np indica el número de piezas en el tablero que se está examinando y npo en el tablero óptimo hasta ese momento.

El árbol tendrá p niveles, uno por cada pieza, y en cada nivel i se decidirá en qué posición del tablero situar la pieza. Como hay nm posibles posiciones y la posibilidad de no incluir la pieza en la solución, cada nodo no terminal tendrá $nm + 1$ hijos. Se empezará intentando poner una pieza en la posición (1,1), después en la (1,2) y así sucesivamente hasta haber probado en todas las columnas de la primera fila, después se pasará a la posición (2,1), etc. La última posibilidad que se contemplará será no incluir la pieza.

La función solucion comprobará si estamos en el último nivel y si se cumplen las restricciones de que no solapen piezas y de que esté entera dentro del tablero, esto se hará con un procedimiento cabe. En caso de caber se pondrá la pieza en el tablero con un procedimiento poner.

La función criterio comprobará que el nodo no sea terminal y que quepa la pieza, con el procedimiento cabe, en cuyo caso la pondrá con poner.

Habrá más hermanos mientras $s[nivel].x \neq 0$.

Al retroceder se debe liberar las posiciones ocupadas en t si se ha puesto la pieza, y se debe actualizar np y volver a poner $s[nivel]$ a su valor inicial.

En generar se generará el siguiente nodo, pero no se pondrá la pieza, lo que se hará con el procedimiento poner, ya sea el nodo solución o no. Para controlar esto se tendrá un array $puesta$:array[1..p]de 0..1, indicando un 0 que la pieza no se ha puesto y un 1 que sí. La función generar genera el nodo pero no pone la pieza, y es en la función poner donde se pone la pieza.

Con todas estas observaciones el esquema podría ser:

$t \leftarrow 0$
$s \leftarrow (-1, -1)$
$so \leftarrow (0, 0)$

```
np = 0
npo = 0
puesta ← 0
nivel = 1
generar
repetir
    si solucion
        si np > npo
            so ← s
            npo = no
        finsi
    finsi
    si criterio
        generar
    else
        mientras no hermanos y nivel ≠ 0
            retroceder
        finmientras
        si nivel ≠ 0
            generar
        finsi
    finsi
hasta nivel = 0
```

La condición de fin es que se recorra todo el árbol pues es un problema de optimización. No hemos considerado parámetros en las funciones y procedimientos por simplificar la escritura.

Las funciones serán:

```
solucion:
    valor=terminal y cabe
    si valor
        poner
    finsi
    devolver valor
```

Donde:

```
terminal:
    devolver (nivel = p)
cabe:
    vale = true
    i = 0
    j = 0
    mientras vale y i ≠ piezas[nivel].x
```

si $(s[nivel].x + i > n)$ o $(s[nivel].y + j > m)$
$vale$=false (*se sale del tablero*)
finsi
si $vale$ (*si no se sale comprueba si solapa*)
si $t[s[nivel].x + i, s[nivel].y + j] = 1$
$vale$=false
finsi
(*pasa a la siguiente posición de la pieza*)
$j + +$
si $j = piezas[nivel].y$
$j = 0$
$i + +$
finsi
finsi
finmientras
devolver $vale$

y:

poner:

(*pone las piezas que previamente ha comprobado que no se salen del tablero ni solapan con otra ya en el tablero*)

desde $i = s[nivel].x$ hasta $s[nivel].x + piezas[nivel].x - 1$
desde $j = s[nivel].y$ hasta $s[nivel].y + piezas[nivel].y - 1$
$t[i, j] = 1$
findesde
findesde
$np + +$
$puesta[nivel] = 1$

La función generar genera el siguiente nodo pero indicando que no se ha puesto la pieza en el tablero, pues esto lo hace la función poner:

generar:

$puesta[nivel] = 0$
si $s[nivel.x = -1$ (*no se ha generado todavía ningún hijo*)
$s[nivel].x = 1$
$s[nivel].y = 1$
en otro caso si $s[nivel].x = n$ y $s[nivel].y = m$ (*se han comprobado todas la posiciones en el tablero*)
$s[nivel].x = 0$ (*se indica que no se pone la pieza*)
en otro caso
si $s[nivel].y \neq m$ (*si no es la última columna*)
$s[nivel].y = s[nivel].y + 1$
en otro caso

$s[nivel].y = 1$
$s[nivel].x = s[nivel].x + 1$
finsi
finsi

Hay más hermanos si no estamos en el último nodo que indica que no se pone la pieza:

hermanos:
devolver $s[nivel].x \neq 0$

Y al retroceder se quita la pieza del tablero si está puesta, en cuyo caso además se quita uno al número de piezas de la solución, y se deja $s[nivel].x$ con valor -1 porque cuando volvamos a bajar a ese nivel será desde otro nodo y habrá que volver a generar nodos desde el principio:

retroceder:
si $puesta[nivel] = 1$
quitar
finsi
$s[nivel].x = -1$
$nivel - -$

Donde:

quitar:
desde $i = s[nivel].x$ hasta $s[nivel].x + piezas[nivel].x - 1$
desde $j = s[nivel].y$ hasta $s[nivel].y + piezas[nivel].y - 1$
$t[i, j] = 0$
findesde
findesde
$np - -$

b) Para resolver el problema por Branch and Bound, como se trata de un problema de maximización del número de piezas que se pueden poner en el tablero, habrá que dar una cota inferior y otra superior de ese número en cada nodo, supuesto que utilizamos un árbol como en el caso del backtracking.

La cota inferior puede ser el número de piezas situadas hasta ese momento, que será el valor *np* del apartado a).

La cota superior puede ser la cota inferior más un número máximo de piezas que se pueden poner de las que quedan. Este número podría ser el número de espacios libres en el tablero partido por el número de cuadros de la pieza más pequeña de entre las que no se han puesto todavía. Para implementar esto de manera eficiente habría que llevar una variable que indicara el número de casillas libres en el tablero, de manera que no se tenga que calcular este número en cada nodo sino que se actualice la variable al poner o quitar una pieza.

Otra forma más sencilla de obtener una cota superior es la cota inferior más el número de piezas que quedan sin colocar. De esta forma la diferencia entre la cota inferior y superior es mayor que con la otra forma de calcular la cota superior, por lo que en este caso seguramente se podarán menos nodos.

Una estimación del beneficio puede ser la media entre la cota inferior y superior, la cota inferior o la cota superior, y estas estimaciones son válidas para cualquiera de las dos maneras de obtener las cotas que hemos dicho.

Una estimación mejor puede ser la cota inferior más una media de las piezas que se pueden incluir, que se puede obtener como el cociente entre el número de casillas libres partido por la media de casillas de las piezas aún sin estudiar. De cualquier manera, como en la estimación tenemos en cuenta el número de casillas libres pero no su posición en el tablero, puede que la media que calculemos de este modo esté muy alejada de la realidad y que lo más realista sea tomar como estimación la cota superior.

Independientemente de cómo se calcule la estimación del beneficio la estrategia de remificación será por el nodo de mayor beneficio estimado y a igualdad se puede tomar una estrategia FIFO o LIFO.

Se podará un nodo cuando su cota superior sea menor que la mayor de las cotas inferiores de los nodos ya generados. Se pueden podar nodos desde el principio pues por debajo de un nodo siempre hay una solución, al menos la que contiene las piezas ya incluidas y ninguna más.

Problema 7.17 a) Consideramos el problema de la mochila modificado consistente en dado un damero de dimensiones $M_1 \times M_2$ y n piezas de dimensiones $i_1 \times j_1$, $i_2 \times j_2$, ..., $i_n \times j_n$, cada una de ellas con beneficio b_1, b_2, ..., b_n, maximizar el beneficio de poner las piezas en el tablero teniendo en cuenta que las piezas se pueden separar en cuadrículas para ponerlas en el tablero y que el beneficio de poner una cuadrícula de una pieza de dimensión $i \times j$ con beneficio b es $\frac{b}{ij}$. Resolver el problema con avance rápido y explicar cómo funcionaría para el caso de un tablero 3×4 y piezas 2×3, 1×4, 3×1 y 2×2, con beneficios 6, 3, 4 y 2, respectivamente.

b) Explicar cómo se podría utilizar el avance rápido anterior en la resolución por Branch and Bound del mismo problema pero sin poder dividirse en cuadrados las piezas.

Solución:

a) El problema es el de la mochila no 0/1, pues las piezas se pueden dividir en cuadrículas para incluirlas en el damero. Se ordenan de mayor a menor beneficio partido por número de cuadrículas y se van metiendo en el tablero mientras el número de cuadrículas de la pieza sea menor o igual que el de casillas libres. Cuando una pieza se mete en el tablero puede que tengamos que dividirla en cuadrículas para que quepa; esto lo hará la función incluir, que no programaremos. Cuando una pieza no cabe entera se incluirán tantas cuadrículas de dicha pieza como queden libres en el tablero. El esquema podría ser:

avance_rapido:
  ordenar piezas de mayor a menor $\frac{b}{ij}$
  $libres = M_1 M_2$

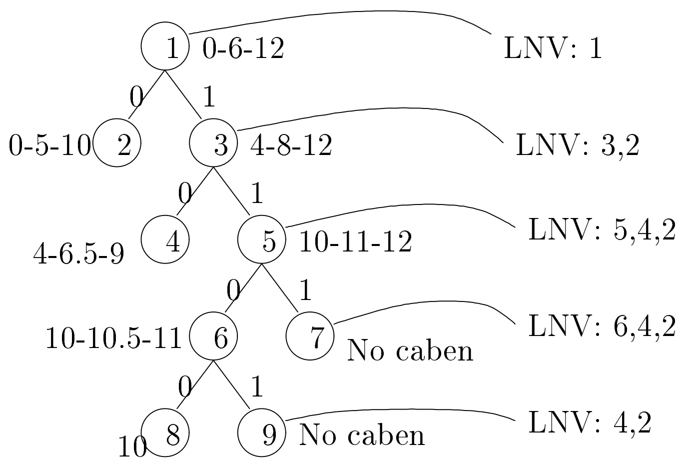
$benef = 0$
$pieza = 1$
while $pieza \leq n$ and $i_{pieza}j_{pieza} \leq libres$
    incluir($pieza$,$i_{pieza}j_{pieza}$)
    $libres = libres - i_{pieza}j_{pieza}$
    $benef = benef + b_{pieza}$
    $pieza++$
endwhile
if $pieza \leq n$
    incluir($pieza$,$libres$)
    $benef = benef + \frac{libres}{i_{pieza}j_{pieza}} b_{pieza}$
endif

Donde *pieza* indica el número de pieza por el que vamos (ya ordenadas). E incluir pone en el tablero una determinada cantidad de cuadrículas de una pieza; no nos importa cómo.

Con el ejemplo que nos dan las piezas quedan ordenadas por beneficios: $4, 6, 3, 2$, y por dimensiones: $3 \times 1, 2 \times 3, 1 \times 4, 2 \times 2$. Inicialmente $libres = 12$; se incluye la primera pieza y $libres = 9$; se incluye la segunda y $libres = 3$; la tercera no cabe entera, por lo que se incluyen tres de sus cuadrículas. El beneficio final es $4+6+\frac{3}{4}3 = 12.25$. La asignación puede ser la que se muestra en la figura, pero depende de cómo se programe incluir:

1	1	1	3
2	2	2	3
2	2	2	3

b) Si las piezas no se pueden dividir tenemos un problema 0/1, por lo que se puede utilizar el método anterior para obtener en cada nodo una cota superior del beneficio alcanzable, al igual que pasa con el problema de la mochila. La cota inferior será en cada nodo el beneficio de las piezas incluidas hasta ese momento, la cota superior el beneficio que se obtiene aplicando avance rápido a partir del nodo (si los datos son enteros podemos tomar la parte entera), y como beneficio estimado podemos tomar la media. Con el ejemplo y considerando las piezas ya ordenadas el recorrido será:



Una vez que se genera el nodo 8 se podan los nodos 4 y 2 pues no se puede mejorar la solución obtenida, y se acaba la ejecución al quedar la Lista de Nodos Vivos vacía.

Problema 7.18 Indicar cómo se puede utilizar el método de avance rápido para resolver el problema de la mochila modificado (visto en los problemas del tema de avance rápido) para resolver el mismo problema por Branch and Bound. Indicar cómo es el árbol de soluciones, cómo es el array solución, cómo se gestiona la lista de nodos vivos, cómo se calculan la cota inferior y superior y la estimación del beneficio en cada nodo, y cómo se realiza la poda. Además, indicar cómo funcionaría con los datos $M = 7$, $b = (4, 5, 3, 6)$ y $p = (2, 3, 1, 3)$, indicando en cada paso de la ejecución cómo queda el recorrido del árbol de soluciones, la lista de nodos vivos y la variable que se utiliza para realizar las podas.

Solución:

El árbol constará de n niveles más el del nodo raíz. En el nivel i-ésimo se decidirá qué porción meter del objeto i. Como cada objeto puede meterse entero, a mitad, o no meterse; cada nodo tendrá tres hijos. Luego el árbol es un árbol ternario de nivel n. Los objetos se ordenarán como en el ejercicio anterior, y los nodos hijos se generarán en el orden 1, 0.5, 0, pues parece más prometedor meter un objeto en la mochila que no meterlo.

El array solución será, por tanto, un array con n índices con valores 1, 0.5 o 0; indicando $s[i] = j$ que del objeto i se mete la porción j en la mochila.

La estrategia de ramificación será LC-FIFO, pero teniendo en cuenta que queremos maximizar el beneficio, por lo que la lista de nodos vivos se mantiene ordenada de mayor a menor beneficio estimado, y a igualdad de beneficio se toma el nodo que primero se generó, ya que se generan antes los nodos de los objetos más prometedores e introduciendo de mayor a menor cantidad de objeto.

La Cota Inferior de un nodo se calculará sumando al beneficio obtenido hasta ese nivel ($\sum_{i=1}^{nivel} b[i] * s[i]$) el resultado de aplicar el avance rápido del ejercicio anterior al problema con capacidad de la mochila la que queda sin ocupar ($M - \sum_{i=1}^{nivel} p[i] * s[i]$) y con los objetos del $nivel + 1$ hasta el n.

La Cota Superior se obtiene sumando al beneficio obtenido hasta ese nivel el resultado de resolver por avance rápido el problema no 0/1 con capacidad la que queda por completar y los objetos del $nivel+1$ hasta el n. Si los beneficios y pesos son números enteros podemos tomar para la cota superior el número entero o fracción de dos anterior al valor que obtengamos.

El beneficio estimado puede ser la media de las dos cotas o cualquiera de ellas, pero tomaremos la Cota Inferior porque se ha obtenido resolviendo por avance rápido el mismo problema que queremos resolver ahora por Branch and Bound.

Para realizar la poda se usará una variable B donde se almacenará el máximo de las cotas inferiores de los nodos generados. Como a partir de cada nodo hay solución, pues al menos tenemos la que se obtiene resolviendo por el avance rápido del ejercicio tres a partir de ese nodo, la poda se puede hacer desde el principio cuando $CS(nodo) < B$. No se hace cuando $CS(nodo) \leq B$ si el B no corresponde a un nodo terminal, pues esto podría hacer que un nodo se podara a sí mismo, como se verá en el ejemplo.

Indicaremos cómo se realiza la ejecución con el ejemplo considerando que los objetos están ordenados de mayor a menor $\frac{b}{p}$: $b=(3,4,6,5)$, $p=(1,2,3,3)$. En una tabla mostramos el orden en que se generan los nodos, la solución parcial (con valores -1 para indicar que no se ha tomado decisión), la Cota Inferior (que se usa también como estimación del beneficio), la Cota Superior, el valor de la variable usada para podar, el estado de la lista de nodos vivos, y algunas anotaciones para indicar si se poda, si la configuración no cumple las restricciones, ...

Nodo	s	*CI*	*CS*	*B*	*LNV*	
1	$-1,-1,-1,-1$	13	14.5	13	1	
2	$1,-1,-1,-1$	13	14.5	13	2	
3	$0.5,-1,-1,-1$	14	14	14	$3,2$	
4	$0,-1,-1,-1$		13	14	$3,2$	*poda*
5	$0.5,1,-1,-1$	14	14	14	$5,2$	
6	$0.5,0.5,-1,-1$		13.5	14	$5,2$	*poda*
7	$0.5,0,-1,-1$		12.5	14	$5,2$	*poda*
8	$0.5,1,1,-1$	14	14	14	$8,2$	
9	$0.5,1,0.5,-1$		13.5	14	$8,2$	*poda*
10	$0.5,1,0,-1$		10.5	14	$8,2$	*poda*
11	$0.5,1,1,1$			14	2	*no caben*
12	$0.5,1,1,0.5$	14	14	14	2	*solucion*
13	$0.5,1,1,0$	11.5	11.5	14	2	*no mejora*
14	$1,1,-1,-1$	13	14.5	14	14	
15	$1,0.5,-1,-1$		14	14	14	*poda*
16	$1,0,-1,-1$		14	14	14	*poda*
17	$1,1,1,-1$	13	14.5	14	17	
18	$1,1,0.5,-1$		14	14	17	*poda*
19	$1,1,0,-1$		12	14	17	*poda*
20	$1,1,1,1$			14		*no caben*
21	$1,1,1,0.5$			14		*no caben*
22	$1,1,1,0$	13	13	14		*no mejora*

Problema 7.19 Considerarmos el problema (visto en los problemas de avance rápido) de dada una tabla de números obtener la sucesión de números de suma máxima de entre las sucesiones formadas entrando por una casilla de la primera fila y pasando de esta casilla a otra de la siguiente fila adyacente a ella en vertical o diagonal, hasta llegar a la última fila. Se pide:
a) Indicar cómo se puede resolver por Backtracking sin utilizar función criterio salvo para indicar que se ha llegado al último nivel del árbol. Utilizar un esquema no recursivo de los vistos en clase y programar las diferentes funciones. Indicar cómo se representan las soluciones, cómo es el árbol de soluciones y cuál es la condición de fin. Calcular el número de nodos que se recorrerían con la tabla ejemplo.
b) Programar una función criterio más elaborada que la del apartado a). Explicar el funcionamiento del programa con la tabla ejemplo, contar el número de nodos que se recorren y compararlo con los calculados en el apartado a).
c) Indicar cómo se podría resolver el problema por Branch and Bound y cómo se puede utilizar el método de avance rápido en este esquema. Habrá que indicar el criterio de recorrido del árbol de soluciones, cómo se calculan las cotas y la estimación del beneficio

en cada nodo y cuál es el criterio de poda. Explicar el funcionamiento del algoritmo con la tabla ejemplo, contar el número de nodos que se recorren y compararlo con los calculados en los apartados a) y b).

Solución:

a) Tenemos un problema de optimización, por lo que el esquema será:

$s \leftarrow 0$ /*Array donde se almacena la solución parcial*/
$valor = 0$ /*Variable donde se guarda la suma actual*/
$SOA \leftarrow 0$ /*Array donde se almacena la solución óptima actual*/
$VOA = 0$ /*Variable donde se guarda la suma óptima actual*/
$nivel = 1$
generar(s,$nivel$)
repeat
    if solucion(s,$nivel$)
        if $valor < VOA$
            $SOA \leftarrow s$
            $VOA = valor$
        endif
    endif
    if criterio(s,$nivel$)
        $nivel + +$
        generar(s,$nivel$)
    else
        while $nivel \neq 0$ and not mashermanos(s,$nivel$)
            retroceder(s,$nivel$)
        endwhile
        if $nivel \neq 0$
            generar(s,$nivel$)
        endif
    endif
until $nivel = 0$

Al ser un problema de optimización la condición de fin es que se recorra todo el árbol y se regrese al nodo raíz ($nivel = 0$), y se lleva una solución óptima actual (SOA) y un valor óptimo actual (VOA) que se actualizan cuando se encuentra un nodo solución que mejora la solución óptima actual.

El árbol tendrá en el nivel 0 el nodo raíz, y además tendrá tantos niveles como número de filas, pues en el nivel i se decide qué columna tomar de la fila i. Dado que de la primera fila se pueden elegir todas las columnas, en el primer nivel habrá m nodos. Como de cada nodo que representa una columna no extrema se puede acceder a 3 columnas en la fila siguiente, y de cada columna extrema a 2 columnas en la fila siguiente, cada nodo en los niveles del 1 al $n-1$ tendrá 2 o 3 hijos.

Como nos dicen que la función criterio debe comprobar sólo que no se ha llegado al último nivel, será:

criterio(s,$nivel$):
  return $nivel \neq n$

La función generar es la que nos recorre el árbol, teniendo éste la forma que hemos explicado:

```
generar(s,nivel):
  if nivel = 1 /*Primera fila*/
    if s[1] = 0 /*Primer hijo*/
      s[1] = 1
      valor = valor + t[1,1] /*t es la tabla*/
    else /*Pasa de un nodo a su hermano*/
      valor = valor − t[1,s[1]]
      s[1]++
      valor = valor + t[1,s[1]]
    endif
  else
    if s[nivel] = 0 /*Primer hijo*/
      if s[nivel − 1] = 1 */Primera columna*/
        s[nivel] = s[nivel − 1]
      else
        s[nivel] = s[nivel − 1] − 1
      endif
      valor = valor + t[nivel,s[nivel]]
    else
      valor = valor − t[nivel,s[nivel]]
      s[nivel]++
      valor = valor + t[nivel,s[nivel]]
    endif
  endif
```

Los nodos solución son los terminales:

solucion(s,$nivel$):
  return $nivel = n$

El número de hermanos depende de si estamos en la primera fila y si estamos en una de las columnas extremas:

```
mashermanos(s,nivel):
  if nivel = 1 /*Primera fila*/
    return s[1] ≠ m
  else
    if s[nivel − 1] = m
      return s[nivel] ≠ m
    else
```

return $s[nivel] \neq s[nivel-1]+1$
endif
endif
Antes de retroceder de nivel hay que dejar s y $valor$ a sus valores anteriores:
retroceder(s,$nivel$):
$valor = valor - t[nivel, s[nivel]]$
$s[nivel] = 0$
$nivel - -$

Podemos llamar $e(i)$ y $c(i)$ al número de nodos que representan columnas extremas y no extremas, respectivamente, en el nivel i. Se tiene además que $e(i+1) = e(i)+c(i)$, y $c(i+1) = e(i) + 2c(i)$. Teniendo esto en cuenta contamos el número de nodos por nivel.

En el nivel 0 tenemos sólo el nodo raíz.

En el nivel 1 tenemos 4 nodos, 2 extremos y 2 no extremos ($e(1)$=2, $c(1) = 2$).

En el nivel 2 tenemos $e(2) = 4$ y $c(2) = 6$, con lo que el número de nodos es 10.

En el nivel 3 $e(3) = 10$ y $c(3) = 16$, y el número de nodos es 26.

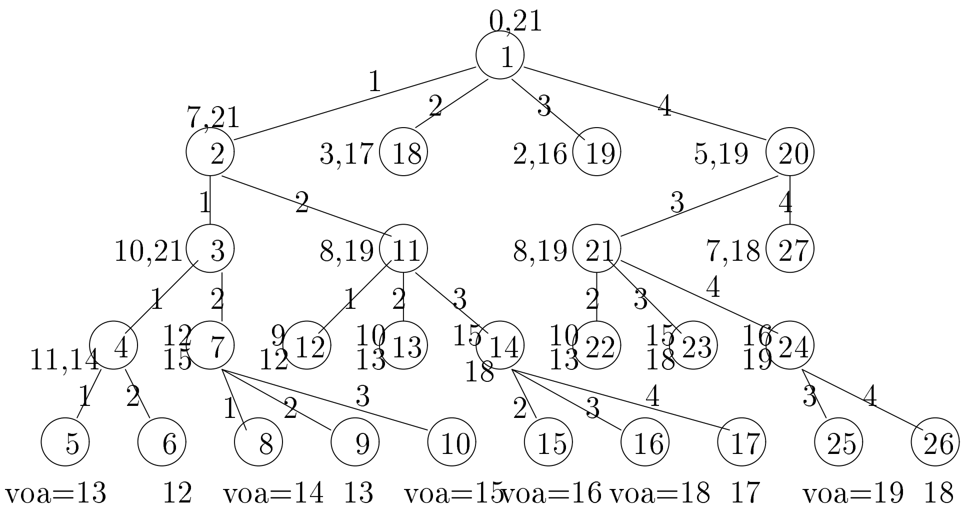
Y en el nivel 4 $e(4) = 26$ y $c(4) = 42$, con lo que el número de nodos es 68.

Así, el número total de nodos en el ejemplo es de 109. Este número se puede obtener también representando el árbol o representándolo hasta el nivel tres y contando los nodos del último nivel. Además sólo habría que representar la mitad del árbol pues es simétrico.

b) Se pueden obtener los valores máximos de cada fila y con estos valores almacenar, en un array $maxasumar$, los valores máximos que se pueden sumar a partir de la fila donde estemos. En el ejemplo sería $maxasumar = (21, 14, 11, 3)$. En la función criterio, además de comprobar que no es un nodo terminal, se comprobará si sumándole al valor que llevamos lo máximo que se puede sumar a partir de esa fila se puede mejorar la solución óptima actual:

criterio(s,$nivel$):
return $nivel \neq n$ and $valor + maxasumar[nivel] > VOA$

Para explicar el funcionamiento con el programa ejemplo y contar el número de nodos utilizamos una gráfica:

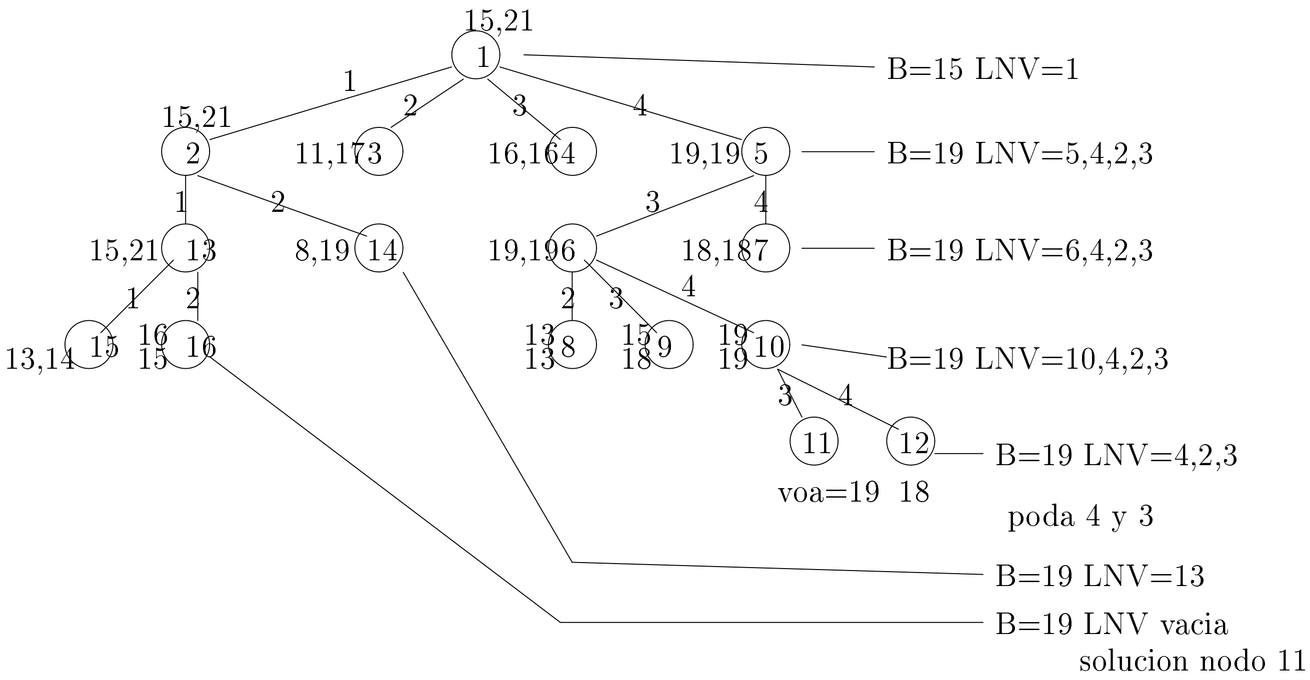


En el interior de cada nodo aparece un número que representa el orden en que se recorren. En cada arista un número representa la columna a que corresponde. Al lado de cada nodo no terminal aparecen dos números, el menor es el valor del nodo y el mayor el máximo que se puede obtener sumando a ese valor *maxasumarnivel*. Al lado de cada nodo terminal aparece su valor, y cuando se actualiza la solución óptima actual se indica que ese valor se asigna a la variable VOA. El nodo con el que se optiene la solución óptima es el 19, pues en él se actualiza por última vez VOA. El número de nodos es 27, con lo que la reducción conseguida respecto al backtracking primitivo es de un 75 por ciento.

c) En cada nodo la cota inferior podría ser el valor asociado al nodo, pero como nos preguntan cómo se puede aplicar el avance rápido a este caso, podemos tomar como cota inferior el valor mas el resultado de aplicar avance rápido a partir de la posición en la que estamos. Este valor nos puede servir también como estimación del beneficio, ya que por medio del avance rápido intentamos aproximarnos a una solución óptima. La cota superior puede ser $valor + maxasumar[nivel]$.

El recorrido puede ser un LC-fifo, siendo en este caso un máximo beneficio en vez de un mínimo costo. Un nodo se podará cuando su cota superior sea menor que la máxima cota inferior de los nodos generados, y cuando ese valor provenga de una solución se puede podar aunque sean iguales, pues en este caso lo máximo que se puede hacer a partir del nodo es igualar una solución que ya tenemos.

Mostramos el funcionamiento con la tabla ejemplo por medio de una gráfica:



Junto a cada nodo se muestran la cota inferior, que es también la estimación del beneficio, y la cota superior. El número de nodos recorridos es 16, lo que representa una reducción de un 40 por ciento respecto a la segunda versión del backtracking.

Problema 7.20 Se tiene una tabla cuadrada de números enteros positivos, donde la entrada i, j indica la preferencia de un trabajador i por realizar un trabajo j. Se trata de hacer una asignación de los trabajos a los trabajadores (un único trabajo a cada trabajador) lo más igualitaria posible, para lo que se pretende minimizar la desviación de la asignación de los trabajos:

$$\sqrt{\sum_{i=1}^{n}(media - t[i,s[i]])^2}$$

donde $s[i]$ representa el trabajo asignado al trabajador i, t es la tabla con las preferencias, y $media$ es la media de las preferencias de las asignaciones hechas ($media = \frac{\sum_{i=1}^{n} t[i,s[i]]}{n}$).
a) Resolver el problema por avance rápido. Se intenta minimizar la desviación, por lo que no será válida una solución que no tenga esto en cuenta.
b) Indicar cómo se puede utilizar el método de avance rápido anterior para resolver el problema por Branch and Bound. Indicar cómo se calcularían las cotas inferior y superior y la estimación del beneficio en cada nodo, y cómo se haría la poda.
Solución:

a) La solución se almacenará en un array s:array$[1, \dots n]$ of $1, \dots, n$; donde $s[i] = j$ significa que al trabajador i se le asigna el trabajo j. Como se debe asignar un único trabajo a cada trabajador será $s[i] \neq s[j]$ si $i \neq j$, y para asegurar esto se llevará un array *asignado*:array$[1, \dots, n]$ of boolean donde true indica que el trabajo correspondiente se ha asignado, por lo que estará inicializado a false.

Un método de avance rápido consiste en una serie de pasos, en este caso en cada paso se decide qué trabajo asignar a un trabajador, por lo que habrá n pasos que se darán en un for. El trabajo que se asigne debe ser de los no asignados todavía (consultando *asignado*), y dentro de estos el que parezca que nos va a dar una asignación más equilibrada. Una posibilidad es hacer un preprocesamiento consistente en calcular la media de todas las preferencias y en cada paso asignar el trabajo no asignado que nos dé la preferencia más cercana a la media:

```
m = 0
for i = 1 to n
    for j = 1 to n
        m = m + t[i, j]
    endfor
endfor
m = m / n^2
for i = 1 to n
    min = ∞
    for j = 1 to n
        if not asignado[j] and |m − t[i, j]| < min
            k = j
            min = |m − t[i, j]|
        endif
    endfor
    asignado[k] = true
    s[i] = k
endif
```

De esta manera se trata de obtener una asignación equilibrada, pero la desviación que se está intentando minimizar es con respecto a la media de todos los valores de la tabla. Para intentar minimizar con respecto a la media de los valores asignados se podría ir calculando las medias parciales y las desviaciones de los valores asignados respecto a esas medias parciales.

b) Ya que la media que hemos calculado en el preprocesamiento no es la media de las preferencias de la asignación final no podemos asegurar que el valor de la desviación que llevamos hasta un cierto nodo sea una cota inferior, por lo que no tenemos cota inferior y tomaremos 0.

El valor que encontramos a partir de un nodo por avance rápido no sirve como cota superior, pues no corresponde al valor de una solución a partir del nodo, pero puede servir como estimación del coste en ese nodo.

Para la cota superior se puede calcular la desviación de la solución encontrada por avance rápido (con respecto a la media de las asignaciones realizadas: $media = \frac{t[1,s[1]]+...+t[n,s[n]]}{n}$). Como estamos minimizando una solución óptima a partir de un nodo debe tener desviación menor o igual que la de la solución encontrada con avance rápido.

Como las cotas superiores que encontramos son valores de soluciones se puede asegurar que a partir de un nodo hay solución y se podría podar desde el principio, pero al ser las cotas inferiores 0 no habrá ninguna manera de podar, salvo que encontráramos una solución cuya desviación fuera cero, en cuyo caso no hace falta seguir recorriendo el árbol.

Bibliografía

[1] María Tereda Abad Soriano. Introducción a los esquemas algorítmicos. Technical Report LSI-97-6-T, Facultad de Informática. Universidad Politécnica de Cataluña, 1997.

[2] Aho, Hoptcroft y Ullman. *The design and analisys of computer algorithms.* Adison-Wesley, 1974.

[3] Aho, Hoptcroft y Ullman. *Estructuras de datos y algoritmos.* Adison-Wesley, 1988.

[4] Baase. *Computer algorithms. Introduction to design and analysis.* Addison-Wesley, 1983.

[5] Brassard y Bratley. *Algorítmica. Concepción y análisis.* Masson, 1990.

[6] Brassard y Bratley. *Fundamentals of algorithmics.* Prentice-Hall, 1996.

[7] Richard P. Brent. Algorithms. Transparencias, nivel 2-5. `http://www.comlab.ox.ac.uk/oucl/users/richard.brent/algs1998.html`

[8] Thomas H. Cormen, Charles E. Leiserson y Ronald L. Rivest. *Introduction to Algorithms.* The MIT Press, 1990.

[9] Ginés García Mateos. Apuntes Algoritmos y Estructuras de Datos II. Facultad de Informática. Universidad de Murcia. 1999. En Power Point. `http://www.um.es/informatica/algyest.htm`

[10] Horowitz y Shani. *Fundamentals of Computer Algorithms.* Pitman, 1978.

[11] Khatib y Shoaff. Algorithms. Transparencias. `http://cs.fit.edu/courses/cse5081`

[12] D. E. Knuth. *El arte de programar ordenadores. Vol 1: algoritmos fundamentales.* Reverté, 1985.

[13] D. E. Knuth. *El arte de programar ordenadores. Vol 3: clasificación y búsqueda.* Reverté, 1987.

[14] Mehlhorn. *Data structures and algorithms. Vol 1: Sorting and Searching.* Springer-Verlag, 1984.

[15] Mehlhorn. *Data structures and algorithms. Vol 2: Graph algorithms and NP-Completeness.* Springer-Verlag, 1984.

[16] Troya Linero. *Análisis y diseño de algoritmos.* VI Escuela de Verano de Informática. AEIA, 1984.

[17] Niklaus Wirth. *Algoritmos+Estructuras de datos=Programas.* Ediciones del Castillo, 1980.

[18] Niklaus Wirth. *Algoritmos y estructuras de datos.* Prentice-Hall, 1987.

[19] Curso algoritmos. Universidad Virtual. Méjico. `http://research.cem.itesm.mx/jesus/cursos/analisis3/presentacion.html`

APUNTES Y PROBLEMAS DE

ALGORÍTMICA

Domingo Giménez Cánovas

Departamento de Informática y Sistemas
Universidad de Murcia

Julio, 2001