

DOMINGO GIMÉNEZ CÁNOVAS
JOAQUÍN CERVERA LÓPEZ
GINÉS GARCÍA MATEOS
NORBERTO MARÍN PÉREZ

**ALGORITMOS Y
ESTRUCTURAS DE DATOS
VOLUMEN II
ALGORITMOS**

AD DE
TIA
ADES



ICE



Índice general

6. Análisis de algoritmos	1
6.1. Consumo de recursos	3
6.1.1. Factores que influyen en el consumo de recursos	3
6.1.2. Tiempos de ejecución	5
6.1.3. Ocupación de memoria	6
6.1.4. Estudio de tiempo de ejecución y ocupación de memoria	6
6.2. Análisis de algoritmos	8
6.2.1. Análisis a priori y a posteriori	8
6.2.2. El proceso de conteo de instrucciones	8
6.2.3. Estudio experimental	13
Ejercicios resueltos	14
Ejercicios propuestos	18
Cuestiones de autoevaluación	19
Referencias bibliográficas	19
7. Notaciones asintóticas	21
7.1. Notaciones asintóticas	23
7.1.1. La notación O	24
7.1.2. La notación Ω	25
7.1.3. La notación Θ	25
7.1.4. La notación o (o-pequeña)	26
7.2. Propiedades de las notaciones asintóticas	27
7.3. Complejidades más frecuentes	29
7.4. Extensiones de las notaciones asintóticas	30
7.4.1. Notaciones con varios parámetros	30
7.4.2. Notaciones condicionales	31
Ejercicios resueltos	32
Ejercicios propuestos	41
Cuestiones de autoevaluación	43
Referencias bibliográficas	44
8. Ecuaciones de recurrencia	45
8.1. Ecuaciones de recurrencia	47
8.2. Expansión de la recurrencia	48
8.3. Método de la ecuación característica	49

8.3.1. Ecuaciones lineales homogéneas	49
8.3.2. Ecuaciones lineales no homogéneas	51
8.3.3. Cambio de variable	52
8.3.4. Transformación de la imagen	52
8.4. Técnica de la inducción constructiva	53
8.5. Fórmulas maestras	54
Ejercicios resueltos	56
Ejercicios propuestos	57
Cuestiones de autoevaluación	58
Referencias bibliográficas	58
9. Divide y vencerás	59
9.1. Método general	61
9.1.1. Esquema general	61
9.1.2. Esquema recursivo	61
9.2. Análisis de tiempos	62
9.2.1. Caso de dos subproblemas	62
9.2.2. Caso de a subproblemas	63
9.3. Búsqueda del máximo y mínimo	64
9.3.1. Método directo	64
9.3.2. Con divide y vencerás	65
9.3.3. Comparación	67
9.4. Ordenación por mezcla	67
9.4.1. Algoritmo divide y vencerás	67
9.4.2. Estudio	69
9.5. Ordenación rápida	70
9.5.1. Algoritmo divide y vencerás	70
9.5.2. Estudio	71
9.5.3. El problema de selección	72
9.6. Multiplicación de enteros largos	73
9.6.1. Multiplicación de Karatsuba y Ofman	74
9.7. Multiplicación de matrices	76
9.7.1. Multiplicación de Strassen	77
Ejercicios resueltos	78
Ejercicios propuestos	93
Cuestiones de autoevaluación	94
Referencias bibliográficas	95
10. Algoritmos voraces	97
10.1. Método general	99
10.1.1. Esquema general	99
10.1.2. Distintas posibilidades en la solución de problemas	100
10.1.3. Análisis del tiempo de ejecución	100
10.2. Ejemplos de avance rápido con técnicas heurísticas	101
10.2.1. Problema del cambio de monedas	101

10.2.2. Paseo del caballo	103
10.2.3. Camino mínimo en grafos multietapa	103
10.3. Problema de la mochila no 0/1	105
10.3.1. Planteamiento	105
10.3.2. Esquema de solución	106
10.3.3. Solución óptima	107
10.4. Secuenciamiento de trabajos a plazos	108
10.4.1. Planteamiento	108
10.4.2. Solución óptima	109
10.4.3. Algoritmo	110
10.5. Heurísticas voraces	112
10.5.1. Problema del viajante	112
10.5.2. Coloración de grafos	114
Ejercicios resueltos	115
Ejercicios propuestos	117
Cuestiones de autoevaluación	118
Referencias bibliográficas	119
11. Programación dinámica	121
11.1. Método general	123
11.1.1. Ideas generales	123
11.1.2. Programación dinámica en problemas que no son de optimización .	125
11.1.3. Análisis de tiempo de ejecución y ocupación de memoria	126
11.2. Problema del grafo multietapa	127
11.3. Problema del cambio de monedas	128
11.3.1. Planteamiento de la solución por programación dinámica	128
11.3.2. Algoritmo	129
11.4. Problema de la mochila 0/1	130
11.4.1. Planteamiento de solución por programación dinámica	130
11.4.2. Algoritmo	131
11.5. Multiplicación encadenada de matrices	132
11.5.1. Planteamiento	132
11.5.2. Solución por programación dinámica	133
Ejercicios resueltos	134
Ejercicios propuestos	142
Cuestiones de autoevaluación	144
Referencias bibliográficas	145
12. Backtracking	147
12.1. Método general	149
12.1.1. Ideas generales	149
12.1.2. Distintos tipos de árboles de soluciones	150
12.1.3. Esquemas de backtracking	152
12.1.4. Análisis de algoritmos de backtracking	157
12.2. Problema de las reinas	157

12.2.1. Planteamiento	157
12.2.2. Soluciones por backtracking	158
12.2.3. Estudio	162
12.3. Problema de la mochila	163
12.3.1. Planteamiento	163
12.3.2. Solución con árbol binario	164
12.3.3. Mejora de la función criterio	165
12.3.4. Modificación del orden de recorrido	166
12.3.5. Solución con árbol combinatorio	167
Ejercicios resueltos	168
Ejercicios propuestos	182
Cuestiones de autoevaluación	183
Referencias bibliográficas	183
13. Ramificación y poda	185
13.1. Método general	187
13.1.1. Ideas generales	187
13.1.2. Estrategias de poda	188
13.1.3. Estrategias de ramificación	188
13.1.4. Esquema general	191
13.1.5. Análisis de algoritmos de ramificación y poda	191
13.2. Problema de la mochila 0/1	192
13.2.1. Árboles de solución	192
13.2.2. Estimación del beneficio y las cotas	192
13.2.3. Estrategias de ramificación y poda	193
13.2.4. Algoritmo	194
13.3. Secuenciamiento de trabajos con plazos	196
13.3.1. Definición del problema	196
13.3.2. Diseño de una solución	197
13.4. Problema de las reinas	198
Ejercicios resueltos	204
Ejercicios propuestos	219
Cuestiones de autoevaluación	220
Referencias bibliográficas	222
14. Árboles de juegos	223
14.1. Árboles de juegos	225
14.2. Estrategia minimax	227
14.2.1. Algoritmo	228
14.3. Poda alfa-beta	229
14.3.1. Algoritmo	230
Ejercicios resueltos	231
Ejercicios propuestos	235
Cuestiones de autoevaluación	237
Referencias bibliográficas	238

15. Complejidad algorítmica	239
15.1. Complejidad de problemas	241
15.1.1. Concepto de cota inferior de un problema	241
15.1.2. Métodos para la obtención de la cota inferior de un problema	242
15.2. Equivalencia de problemas	245
15.2.1. Reducción entre problemas matriciales	246
15.3. Clasificación de problemas	247
15.3.1. Problemas P y NP	248
15.3.2. Soluciones heurísticas y aproximadas	248
Ejercicios resueltos	249
Ejercicios propuestos	250
Cuestiones de autoevaluación	251
Referencias bibliográficas	251
D. Práctica 2: Análisis de algoritmos	255
D.1. Enunciado	255
D.2. Cómo resolver la práctica	257
E. Práctica 3: Esquemas algorítmicos	277
E.1. Enunciado	277
E.2. Programación dinámica	278
E.2.1. Resolución	278
E.2.2. Estudio teórico y experimental	278
E.3. Avance rápido	280
E.3.1. Resolución	280
E.3.2. Estudio teórico y experimental	281
E.4. Backtracking	282
E.4.1. Resolución	282
E.4.2. Estudio teórico y experimental	283
E.5. Ramificación y poda	284
E.5.1. Resolución	284
E.5.2. Estudio teórico y experimental	286

Capítulo 6

Análisis de algoritmos

La ejecución de un algoritmo consume recursos de tiempo, memoria, etc. Nos interesa estimar los recursos consumidos, para poder determinar si un algoritmo es preferible a otro que resuelve el mismo problema, comprobar si un programa realizado a partir de un algoritmo funciona correctamente, o ver si un algoritmo es viable antes de implementarlo. En este capítulo analizaremos el proceso básico para medir el consumo de tiempo de ejecución y de ocupación de memoria.

Objetivos del capítulo:

- Identificar los recursos consumidos por un algoritmo y los distintos factores que afectan a este consumo. Entender que para cada aplicación puede variar lo que se considera un recurso crítico.
- Distinguir los distintos tiempos de ejecución que se utilizan en el estudio de algoritmos: caso más favorable, más desfavorable y promedio.
- Distinguir las distintas medidas de la ocupación de memoria: caso más favorable, más desfavorable y promedio.
- Aprender las técnicas básicas del estudio teórico de algoritmos: análisis del tiempo por conteo de instrucciones y estudio de la ocupación de memoria. Comprender la importancia de este estudio.
- Saber determinar qué parámetros constituyen la dimensión de un problema.
- Aprender a realizar estudios experimentales, comprender su importancia y la necesidad de contrastar los resultados experimentales con los teóricos.

Contenido del capítulo:

6.1. Consumo de recursos	3
6.1.1. Factores que influyen en el consumo de recursos	3
6.1.2. Tiempos de ejecución	5
6.1.3. Ocupación de memoria	6
6.1.4. Estudio de tiempo de ejecución y ocupación de memoria	6
6.2. Análisis de algoritmos	8
6.2.1. Análisis a priori y a posteriori	8
6.2.2. El proceso de conteo de instrucciones	8
6.2.3. Estudio experimental	13
Ejercicios resueltos	14
Ejercicios propuestos	18
Cuestiones de autoevaluación	19
Referencias bibliográficas	19

6.1. Consumo de recursos

La ejecución de un algoritmo **consume recursos** de tiempo de ejecución y de ocupación de memoria. Se llama eficiencia de un algoritmo a la relación entre los recursos que consume y los resultados que obtiene. En este sentido, se pretende desarrollar algoritmos eficientes, y no únicamente algoritmos que consuman pocos recursos. Por ejemplo, un algoritmo para resolver un problema de optimización puede ser muy rápido pero no encontrar la solución óptima, mientras que otro puede consumir mucho tiempo de ejecución y obtener la solución óptima. Cuál de los dos algoritmos es preferible puede depender de que se necesite la solución óptima o de que sea suficiente con obtener una cercana a la óptima pero se necesite la respuesta en un tiempo reducido.

Además del consumo de recursos hay otros criterios a tener en cuenta a la hora de desarrollar algoritmos: facilidad de programación, robustez, posibilidad de reutilización, facilidad de mantenimiento, etc.

De los recursos consumidos por un algoritmo el tiempo de ejecución y la ocupación de memoria se pueden estimar con las técnicas que se analizan en este capítulo y los dos siguientes. Lo que se intentará es que los programas se ejecuten en poco tiempo y usen poca memoria. Las técnicas de análisis de ocupación de memoria y de tiempo de ejecución son similares, aunque normalmente es más complejo el estudio del tiempo de ejecución, por lo que nos centraremos en este estudio con preferencia sobre el de ocupación de memoria.

6.1.1. Factores que influyen en el consumo de recursos

El valor exacto del tiempo de ejecución y de la ocupación de memoria dependerá de una serie de factores externos al algoritmo, y también del tamaño y la forma de los datos de entrada.

Factores externos que influyen en el tiempo de ejecución son:

- El programa con que se implemente el algoritmo y, por tanto, la experiencia del programador.
- La máquina en que se ejecuta.
- El compilador utilizado o las opciones de compilación.
- Los tipos de datos utilizados.
- Los usuarios que están trabajando en el sistema.

Debido a la dependencia de esta serie de factores, no se puede predecir exactamente el tiempo de ejecución, y se debe independizar su estudio de los factores externos. Muchas veces lo que interesa es calcular la forma en que crece la función tiempo (o la ocupación de memoria).

El estudio se hace en función de la entrada. Puede que el consumo de recursos dependa sólo del tamaño de la entrada o que además dependa del contenido de la entrada. Si depende del tamaño, el tiempo se representa como $t(n)$, con n el tamaño de la entrada. Lo que se considera tamaño de la entrada varía para distintos problemas:

- En el cálculo del factorial de un número el tamaño es el valor del número.
- En la ordenación de números el tamaño es el número de datos a ordenar.
- El tiempo puede depender de varios parámetros. Por ejemplo, en la multiplicación de matrices rectangulares de dimensiones $n \times m$ y $m \times r$, el tiempo depende de los parámetros n , m y r .

El tiempo también puede depender del contenido de la entrada. En este caso tendremos $t(n, \sigma)$, donde n es el tamaño de la entrada y σ representa la entrada.

Por ejemplo, en la ordenación de números el tiempo también depende de la posición relativa de los números en la entrada, y no sólo de la cantidad de datos, pues normalmente el tiempo será menor si los datos están ordenados crecientemente que si están inversamente ordenados.

Cuando el tiempo de ejecución depende del contenido de la entrada, se pueden estudiar los tiempos en el caso más favorable y más desfavorable:

- **El tiempo en el caso más favorable** ($t_m(n)$), que es el tiempo de ejecución con la entrada que nos produce el menor tiempo. Por ejemplo, en una ordenación puede ser que la entrada más favorable sea una en que los datos estén ordenados.
- **El tiempo en el caso más desfavorable** ($t_M(n)$) es el tiempo de ejecución con la entrada que produce el mayor tiempo. En una ordenación puede ser la entrada con los datos inversamente ordenados.

Puede interesar también obtener el **tiempo promedio**, que dará una idea del tiempo que podemos esperar en una ejecución particular. El tiempo promedio ($t_p(n)$) se obtiene promediando los tiempos de cada una de las posibles entradas por la probabilidad de que aparezca la entrada:

$$t_p(n) = \sum_{\sigma \in S} t(\sigma) p(\sigma) \quad (6.1)$$

donde S es el conjunto de todas las posibles entradas. Según la fórmula habría que calcular los tiempos de todas las entradas para un tamaño dado, pero esto no es siempre factible. Por ejemplo, en el caso de ordenar n naturales, si el rango de los naturales es $0, \dots, 32.000$, tendríamos 32.001^n posibles entradas distintas, pero el tiempo de ejecución no dependerá de los valores concretos de los números, sino de su posición relativa. Por tanto, se pueden considerar entradas iguales aquellas en las que las posiciones relativas son las mismas, con lo que tenemos un total de $n!$ entradas distintas. Aun así, el número de entradas distintas puede ser muy grande y en algunos casos se calcula el tiempo promedio contando las operaciones y multiplicando cada operación por la probabilidad de que se ejecute.

Las mismas consideraciones hechas para el tiempo de ejecución son válidas para la ocupación de memoria.

6.1.2. Tiempos de ejecución

El **tiempo de ejecución** se mide en unidades de tiempo desde que empieza la ejecución del programa hasta que se obtienen los resultados, y se denota como $t(n)$, con t una función $t : N \rightarrow R^+$, y tamaño de la entrada n .

Dado un algoritmo no estamos interesados en calcular un valor exacto del tiempo de ejecución, sino que modelizamos este tiempo. El estudio se suele hacer por **conteo de instrucciones**. Una posibilidad es contar el número de veces que se ejecutaría cada instrucción en el algoritmo y obtener el número total de instrucciones.

En ese caso no se considera que puede haber instrucciones de distinto tipo y que tienen distinto coste. Para solventar este problema se pueden contar las instrucciones de distinto tipo por separado, y el número total de instrucciones se obtendría multiplicando cada tipo de instrucciones por una constante distinta. Por ejemplo, en un método de ordenación por comparación se pueden contar, por un lado, las comparaciones de elementos del tipo que se están ordenando, por otro las asignaciones de elementos de ese tipo, por otro las asignaciones de índices, y por otro las comparaciones de índices.

En algunos tipos de problemas se suele contar sólo algunos tipos de operaciones, porque son las más costosas en esos problemas. Esto pasa con los algoritmos numéricos que trabajan con números en coma flotante. En este caso se suele contar el número de operaciones en coma flotante (flops), sin tener en cuenta el trabajo con los índices.

Si no se puede estimar el tiempo, se puede intentar acotarlo o calcular la forma en que crece. Los tiempos más favorable y más desfavorable nos sirven como cotas del tiempo de ejecución:

- El tiempo en el caso más favorable sirve como **cota inferior** del tiempo de ejecución. Hay veces en que tampoco se puede obtener el valor de $t_m(n)$, pero sí unas cotas para él. Una cota inferior de $t_m(n)$ nos sirve como cota inferior de $t(n)$, pero la cota superior de $t_m(n)$ no nos sirve para acotar $t(n)$.
- El tiempo en el caso más desfavorable sirve como **cota superior**. $t_M(n)$ es una cota superior de $t(n)$. Si no podemos obtener $t_M(n)$ pero sí unas cotas, una cota superior de $t_M(n)$ será también cota superior de $t(n)$, pero una cota inferior de $t_M(n)$ no nos sirve para acotar $t(n)$.

Para cualquiera de los tiempos tratados puede ocurrir que, por las características del algoritmo, no seamos capaces de determinar su valor. En estos casos se puede intentar a su vez encontrar una cota inferior y otra superior.

Errores comunes

Hay una serie de fallos usuales de los que es conveniente huir:

- El caso más favorable no es tener el menor tamaño posible en la entrada sino, dado un tamaño de entrada fijo, cuál es el contenido de los datos que necesita un menor tiempo de ejecución, aun para tamaños grandes.
- El tiempo promedio no es la media de los tiempos más favorable y más desfavorable, aunque siempre estará entre ambos.

- No siempre los casos más favorables y más desfavorables son los que aparentan. Por ejemplo, en una ordenación rápida (quicksort) el caso más favorable puede no ser el de tener los datos ordenados sino todo lo contrario: ese caso puede ser el más desfavorable.

6.1.3. Ocupación de memoria

El estudio de la ocupación de memoria es un aspecto fundamental en el diseño de una estructura de datos, como vimos en los capítulos anteriores. Pero, de la misma forma que también resultaba importante considerar el tiempo de las operaciones que manejan la estructura, una parte fundamental del análisis de algoritmos es el estudio de la memoria requerida por las estructuras que utiliza. La **ocupación de memoria** de un algoritmo 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 su ejecución.

Todo lo dicho sobre los tiempos de ejecución, casos más favorable y más desfavorable y promedio, y la necesidad en algunos casos de calcular cotas, sirve también para la ocupación de memoria. Tendremos, por tanto, las siguientes medidas de ocupación de memoria:

- Memoria en el caso más favorable ($m_m(n)$), que es la memoria que se necesita para ejecutar el programa con la entrada que genera menos necesidades.
- Memoria en el caso más desfavorable ($m_M(n)$), que es la que se necesita para ejecutar el programa con la entrada que genera más necesidades.
- Memoria promedio ($m_p(n)$), que se obtiene promediando la memoria necesitada para cada entrada por la probabilidad de que aparezca.

Hay una diferencia importante entre el estudio del tiempo y la memoria. En el tiempo de ejecución, las cotas –y por lo tanto los casos más favorable, más desfavorable y el tiempo promedio– nos dan una idea de cuál puede ser el tiempo de una ejecución particular. Si el tiempo de ejecución es muy grande, el estudio debe estimar lo que puede tardar en resolverse el problema (puede que siglos) pero no se concluye que no se pueda resolver. En el caso de la ocupación de memoria, la medida más interesante es $m_M(n)$, pues indica la memoria que necesitamos para que se pueda resolver cualquier problema de tamaño n . De este modo, si en un sistema se dispone de una cierta cantidad de memoria y $m(n)$ es mayor que esa cantidad, el problema no se puede resolver en ese sistema.

6.1.4. Estudio de tiempo de ejecución y ocupación de memoria

Aunque, como ya hemos dicho, las técnicas para estudiar el tiempo de ejecución y la ocupación de memoria son las mismas, suele ser más fácil el estudio de la ocupación de memoria por varios motivos:

- La ocupación no suele depender del contenido de la entrada, mientras que el tiempo sí. Si pensamos en un método de ordenación de datos que están en un array, el

tiempo varía con la posición relativa de los datos, pero la ocupación de memoria no, pues sólo se necesita espacio para el array y las variables auxiliares e índices que se utilizan en el programa. Tendremos valores distintos para $t_m(n)$, $t_M(n)$ y $t_p(n)$, pero será $m_m(n) = m_M(n) = m_p(n)$.

- El coste del tiempo de ejecución suele ser mayor que el de la ocupación de memoria. Por ejemplo, para calcular el tiempo se puede necesitar expandir una recursión si hacemos llamadas recursivas, mientras que para la ocupación de memoria no será necesario resolver una ecuación recursiva si no se reserva nueva memoria en las sucesivas llamadas.

Ilustramos estas diferencias en el estudio del tiempo y la memoria con el siguiente ejemplo.

Ejemplo 6.1 Estudiamos la función factorial:

```

operación fact(n:entero):entero
    si n ≠ 1 entonces
        devolver fact(n-1)*n
    sino
        devolver 1
    finsi
```

Analizamos el tiempo contando las instrucciones que se ejecutan, y asignando valores constantes a algunas instrucciones, y la memoria contando en número de variables y parámetros para los que se necesita espacio.

El tiempo se puede obtener a partir de la recurrencia $t(n) = t(n - 1) + a$, con a una constante que corresponde a la comprobación del **si**, la resta y la multiplicación. Expandiendo la recursión tenemos $t(n) = t(n - 2) + 2a = \dots = t(1) + (n - 1)a$, con lo que la forma del tiempo de ejecución es lineal.

La ocupación de memoria se puede estudiar con una recurrencia igual, $m(n) = m(n - 1) + b$, con b la memoria necesaria para almacenar el parámetro y el valor que devuelve **fact**¹. En el momento de mayor ocupación necesitamos $2n$ posiciones, por lo que la ocupación de memoria es $m(n) = bn$.

El tiempo y la memoria se estudian de manera similar y tienen la misma forma lineal, pero no suele ser así. Consideraremos una versión iterativa del factorial:

```

operación fact(n:entero):entero
    f:=1
    para i:=2,...,n hacer
        f:=f*i
    devolver f
```

El tiempo de ejecución se puede calcular planteando también en este caso una ecuación de recurrencia. Es $t(n) = a + t_{para}(n)$, con a el coste de las instrucciones externas al bucle y $t_{para}(n)$ el tiempo de ejecución del bucle con valores de 2 a n . Se puede plantear la recurrencia $t_{para}(n) = t_{para}(n - 1) + b$, con b el coste de un paso por el bucle. De este modo, el tiempo de ejecución vuelve a ser lineal, pero la ocupación de memoria es constante pues sólo se necesita espacio para el parámetro n y las dos variables f e i .

¹Se ocupa en realidad más espacio por necesitar índices de regreso de la llamada

6.2. Análisis de algoritmos

6.2.1. Análisis a priori y a posteriori

El análisis de algoritmos tiene sentido en dos casos:

- Se puede hacer un **estudio del algoritmo a priori**, antes de realizar el programa.
Este estudio tiene sentido por varios motivos:
 - Como cualquier otro estudio previo para evitar la pérdida de tiempo ante la máquina, al identificar puntos del algoritmo que es mejor programar de una cierta manera.
 - Para determinar si un algoritmo es bueno y merece la pena hacer un programa para ese algoritmo o pensar otro mejor.
 - Para poder determinar entre dos algoritmos o programas dados cuál es preferible.
 - Para determinar el tamaño de los problemas que podemos resolver dependiendo de la memoria de que dispongamos.
- También se puede hacer un **estudio a posteriori** después de haber hecho el programa. Este estudio tiene sentido en varios casos:
 - Para comparar dos programas de manera que se puede decidir cuál es preferible utilizar en la solución del problema para una cierta entrada o tamaño de la entrada.
 - Para encontrar la causa del mal funcionamiento de un programa o identificar puntos que pueden ser mejorados.

El estudio de un algoritmo se puede hacer de manera teórica, estimando el tiempo de ejecución y la ocupación de memoria, o experimentalmente, llevando a cabo experimentos una vez desarrollado un programa que implemente el algoritmo. El estudio teórico puede ser a priori o a posteriori, mientras que el experimental sólo se puede hacer a posteriori. En algunos casos, en el proceso de implementación de un algoritmo se pueden mezclar los dos estudios, ya que se podría desarrollar una parte del algoritmo, haciendo su estudio teórico y experimental y, una vez que esa parte trabaja satisfactoriamente, pasar a la implementación de otra parte.

6.2.2. El proceso de conteo de instrucciones

La forma más básica de estudio del tiempo consiste en **contar las instrucciones** que se ejecutan a lo largo de la ejecución. Como ya hemos dicho, se suele hacer por conteo de instrucciones, contándolas todas o sólo las de un cierto tipo, según la clase de problema que se resuelva. Los pasos a seguir para este tipo de estudio serían:

- Decidir las instrucciones que se quiere contar.

- Se puede decidir contar las instrucciones pero asignándole a instrucciones de distinto tipo costes distintos, y calcular el tiempo en función de esos valores. De esta manera al hacer la implementación y en un sistema concreto se puede estimar los valores de las constantes y sustituirlos en la fórmula.
- Si tenemos llamadas a funciones, primero se cuenta el número de instrucciones de las funciones y en cada llamada se sustituye ese número.
- En el caso de bucles se puede expresar con un sumatorio, con índices que indican los sucesivos pasos por el bucle. Si no sabemos con qué valores se pasa (lo que puede ocurrir en un bucle **mientras** o **repetir**) se puede obtener unos valores mínimo y máximo, que nos servirán para obtener cotas inferiores y superiores del número de instrucciones.
- Si tenemos una bifurcación, habrá que contar el número de veces que se pasa por cada rama, y si no es posible se puede utilizar el número de instrucciones de la rama más costosa para obtener una cota superior, y el de la menos costosa para la cota inferior.

Una alternativa distinta es la estimación del tiempo contando una serie de operaciones básicas, como puede ser el número de nodos que se recorre en un algoritmo sobre grafos o árboles, o en algoritmos que trabajan sobre matrices el número de datos de la matriz a los que se accede. Esto es lo que vimos en el capítulo 5 como estudio del trabajo total realizado.

Ejemplo 6.2 Como ejemplo de comparación de algoritmos podemos estudiar la ordenación de datos de un cierto tipo almacenados en un array, según los dos algoritmos siguientes. Identificamos cada instrucción que consideramos elemental con un par de números para poder referirnos a ella en el estudio:

operación ordenar1(*x:array[1..n]* de tipo)

```

1.1   fin:=falso
1.2   mientras NO fin hacer
1.3       fin:=verdadero
1.4       para i:=1,...,n-1 hacer
1.5           si x[i]>x[i+1] entonces
1.6               intercambiar
1.7               fin:=falso
                   finsi
               finpara
           finmientras

```

operación ordenar2(*x:array[1..n]* de tipo)

```

2.1   para i:=1,...,n-1 hacer
2.2       para j:=i+1,...,n hacer
2.3           si x[i]>x[j] entonces
2.4               intercambiar
                   finsi

```

finpara

finpara

Estudiamos el número de veces que cada uno de los algoritmos pasa por una sentencia de las que hemos considerado elemental en el caso más desfavorable. El caso más favorable se estudiará de manera similar.

En el estudio de algoritmos se pretende obtener la función tiempo para cualquier tamaño n (función $t(n)$), pero en algunos casos es preferible hacer estudios para casos particulares y pequeños que nos permitan sacar conclusiones del funcionamiento para casos mayores.

Consideramos $n = 4$ y 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
				1.4			salida del bucle	
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		salida del bucle	
1.2	1.3		1.4	1.5	1.6	1.7	quedará	1 2 3 4
				1.4	1.5		salida del bucle	
				1.4	1.5		salida del bucle	
1.2	1.3		1.4	1.5			salida del bucle	
			1.4	1.5			salida del bucle	
				1.4			salida del bucle	
					1.2		salida del bucle	

Analizando este caso podemos obtener el número de instrucciones en el caso general, que será de la forma $t(n) = 2 + \sum_{p=1}^n t_{mientras}(n, p)$. El 2 corresponde a la instrucción 1.1 y a la ejecución de la 1.2 para salir del bucle. Hay n pasos por el bucle, en cada paso se lleva un elemento a su posición, y en el último se comprueba que están ordenados al no tener que hacer ningún cambio. p indica el paso por el que vamos. $t_{mientras}(n, p)$ representa el coste de cada paso por el bucle, y es $t_{mientras}(n, p) = 3 + 2(n-1) + 2(n-p)$, donde el 3 corresponde a las instrucciones 1.4 y a las instrucciones 1.2, 1.3 y la 1.4 para salir, $2(n-1)$ corresponde a las instrucciones 1.4 y 1.5, y $2(n-p)$ a las 1.6 y 1.7. Sustituyendo queda $t(n) = 2 + \sum_{p=1}^n (4n - 2p + 1) = 3n^2 + 2$.

Ya que instrucciones distintas pueden tener distinto coste, podemos hacer el estudio asignando constantes distintas: a la 1.1, 1.3 y 1.7 les asignamos coste a , a la 1.2 b , a la 1.4 c , a la 1.5 d , y a la 1.6 coste e . El tiempo sería $t(n) = a + b + \sum_{p=1}^n (a + b + c + (c + d)(n-1) + (e+a)(n-p)) = n^2 (\frac{a}{2} + c + d + \frac{e}{2}) + n (\frac{a}{2} - \frac{e}{2}) + a + b$.

Para ordenar2 se tiene:

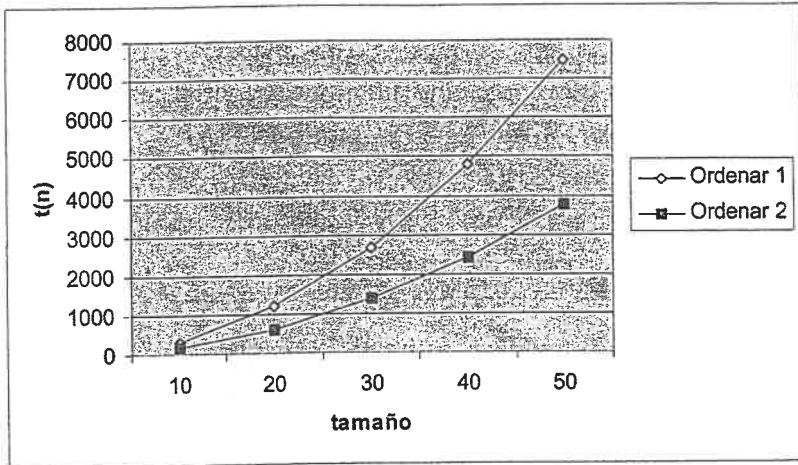


Figura 6.1: Tiempos de ejecución en el caso más desfavorable de los dos algoritmos del ejemplo 6.2.

```

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
      2.2           salida del bucle
2.1 2.2 2.3 2.4 quedará 1 3 4 2
      2.2 2.3 2.4 quedará 1 2 4 3
      2.2           salida del bucle
2.1 2.2 2.3 2.4 quedará 1 2 3 4
      2.2           salida del bucle
      2.1           salida del bucle
  
```

En este caso es más fácil estudiar el tiempo, pues los dos bucles se pueden poner como sumatorios y se ejecutan todas las instrucciones cada vez que se pasa. Si contamos las instrucciones tenemos $t(n) = 1 + \sum_{i=1}^{n-1} \left(2 + \sum_{j=i+1}^n 3 \right)$, donde el 1 corresponde a la instrucción 2.1 cuando se ejecuta para comprobar que no hay que entrar al bucle, el 2 a la 2.1 y la 2.2 para salir del bucle, y el 3 a la 2.2, 2.3 y 2.4. El tiempo queda $t(n) = \frac{3}{2}n^2 + \frac{n}{2} - 1$.

Si consideramos cada instrucción con el valor constante antes asignado, las instrucciones 2.1 y 2.2 tienen coste c , la 2.3 coste d , y la 2.4 coste e . Con estos valores el coste queda $t(n) = n^2 \frac{c+d+e}{2} + n \frac{3c-d-e}{2} - c$.

Comprobamos que es preferible el segundo método, y que será aproximadamente el doble de rápido que el primero en el caso más desfavorable. Esto se ve también representando los dos tiempos de ejecución en la figura 6.1.

Está claro que no todas las instrucciones tienen el mismo coste. Las instrucciones de trabajo con booleanos pueden ser menos costosas que las de trabajo con índices, y la instrucción intercambiar conllevaría tres asignaciones. Además, dependiendo del tipo

de datos que se esté ordenando y de cómo estén implementados, puede que hacer una asignación o una comparación no tengan un tiempo constante (por ejemplo, si trabajamos con listas de caracteres), en cuyo caso el estudio que hemos realizado con constantes sería válido, aunque sustituyendo las constantes correspondientes por el coste de las operaciones asociadas.

Conteo en el caso promedio

Como ya hemos mencionado, el estudio del tiempo promedio por la fórmula 6.1 puede ser muy costoso, y en algunos casos es preferible calcularlo por conteo de instrucciones. Se trata de multiplicar cada instrucción por la probabilidad de que se ejecute. Se puede sumar el número promedio de veces que se ejecuta cada instrucción. Tendríamos $\sum_{i=1}^k t_p(n, i)$, donde k es el número de instrucciones del programa y $t_p(n, i)$ es el número promedio de veces que se ejecuta la instrucción i para tamaño del problema n . A su vez, $t_p(n, i) = \sum_{j=1}^{\infty} p(i, j)j$, con $p(i, j)$ la probabilidad de que la instrucción i se ejecute j veces. Ilustramos este método con el siguiente ejemplo.

Ejemplo 6.3 Consideremos el algoritmo de búsqueda secuencial con centinela:

```
operación búsqueda(a:array[1..n] de tipo):entero
    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.

Para calcular el tiempo en el caso promedio hay que hacer una asignación de probabilidades razonable. Podemos considerar que x está en el array con probabilidad p (por lo tanto no está con probabilidad $1 - p$) y que si está en el array tiene la misma probabilidad de estar en cualquiera de los n lugares del array, con lo que tendremos una probabilidad $\frac{p}{n}$ de que esté en el lugar i , con i entre 1 y n . Estas suposiciones son razonables si el rango de los datos es grande y es poco probable que haya elementos repetidos en el array.

Se puede estudiar el tiempo promedio estudiando las distintas entradas posibles y calculando el tiempo de cada entrada, o por conteo de instrucciones. Veremos los dos casos.

El conjunto de posibles entradas lo dividimos en entradas que nos produzcan distintos tiempos. Está claro que de la entrada lo único que nos interesa es la primera aparición del valor que estamos buscando, y puede ser en la posición i del array, con $i = 1, \dots, n$, o que no esté en el array. Si la primera aparición es en el lugar i , el número de operaciones es $2i + 2$, y si no está en el array es $4 + 2n$. Con estos valores:

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

Podemos hacerlo también por conteo de instrucciones. Las dos primeras instrucciones se ejecutan una vez, y las instrucciones del interior y de control del bucle se ejecutan i veces, con $i = 1, \dots, n$, con probabilidad $\frac{p}{n}$, y $n+1$ veces con probabilidad $1-p$ (cuando el dato no está en el array). Por tanto:

$$t_p(n) = 2 + 2 \left(1 \frac{p}{n} + 2 \frac{p}{n} + \dots + n \frac{p}{n} + (n+1)(1-p) \right) = 2n + 4 - (n+1)p$$

que coincide con el valor obtenido con el método anterior.

6.2.3. Estudio experimental

Una vez implementado un algoritmo, se puede hacer –y es conveniente– un estudio experimental, y comparar los resultados de este estudio con los del estudio teórico para poder identificar posibles errores o elementos mejorables en la implementación.

En algunos casos la única posibilidad de analizar un algoritmo es con un estudio experimental una vez implementado. Esto ocurre cuando no somos capaces de obtener teóricamente una fórmula del tiempo de ejecución.

Algunos aspectos a tener en cuenta al hacer el estudio experimental son:

- Se deben hacer experimentos para distintos tamaños del problema.
- Si lo que se quiere es estudiar el tiempo en el caso más desfavorable o más favorable, se identificarán esas entradas y se calcularán los tiempos para ellas. Si se quiere estudiar el tiempo promedio se generarán entradas aleatorias.
- Ejecuciones distintas de un mismo programa para una misma entrada pueden producir tiempos distintos, por lo que se deben realizar distintos experimentos para un tamaño dado, y tomar la media de estos tiempos como el tiempo para ese tamaño. Se puede decidir descartar los casos extremos que reflejarían ejecuciones anómalas (más usuarios en el sistema, trabajo del sistema operativo, etc.).
- Los tiempos para los distintos tamaños se representan en una gráfica como la de la figura 6.1. La variación del tiempo con el tamaño de la entrada debe corresponderse con la obtenida teóricamente, aunque puede haber diferencias debidas a aspectos que no se tienen en cuenta en el estudio teórico, como son las diferencias en la velocidad de acceso a los datos según se encuentren en una zona u otra de la jerarquía de memorias, más usuarios en el sistema o procesos del sistema ejecutándose, etc. Por ejemplo, si se ha obtenido teóricamente $t(n) = n^2$, la gráfica debe tener forma de parábola.

- La comparación del tiempo teórico y experimental se puede hacer simplemente observando la gráfica, pero también se puede hacer por un **ajuste por mínimos cuadrados** de los datos experimentales a una función del tipo obtenido en el estudio teórico. Por ejemplo, si en el estudio teórico hemos obtenido que el tiempo de ejecución es de la forma $t(n) = an^2 + bn + c$ (una parábola), los valores de a , b , y c se pueden obtener haciendo el ajuste con los resultados experimentales. De esta forma se obtienen los valores de las constantes para el sistema concreto donde hacemos los experimentos, y podremos predecir el tiempo de ejecución para otros tamaños de la entrada.
- Ya que el término de mayor orden es el que tiene más importancia en el tiempo de ejecución cuando aumenta el tamaño del problema, puede ser suficiente considerar sólo este término y comprobar si, para los distintos tamaños con los que se ha experimentado, el valor de la constante del término de mayor orden es aproximadamente igual. Por ejemplo, si el tiempo es una parábola consideramos sólo an^2 , y si hemos obtenido $t(10) = 2$ seg, $t(20) = 7$ seg y $t(40) = 30$ seg, los valores de a que obtenemos son $a = 0,02$ para $n = 10$, $a = 0,0175$ para $n = 20$, y $a = 0,01875$ para $n = 40$. Con estos datos podríamos concluir que los resultados experimentales confirman los teóricos, aunque tres experimentos no son suficientes para sacar conclusiones definitivas.
- Para seleccionar los tamaños con los que hacemos los experimentos debemos tener en cuenta que para tamaños pequeños los datos pueden estar en memoria caché, para tamaños medianos estarán en la memoria principal, y para tamaños grandes puede ser necesario trabajar con memoria secundaria haciendo *swapping*. Por tanto el acceso a los datos tiene coste distinto para rangos distintos de la entrada, y si queremos estimar el valor de las constantes que aparecen en el tiempo teórico debemos decidir en qué rango de tamaños vamos a hacer el estudio. Lo más normal es hacer el estudio para unos tamaños medianos, en los que los datos no estén mayoritariamente en la caché cuando se accede a ellos, pero tampoco se haga *swapping*.

Si el estudio experimental no confirma las predicciones del teórico, hay que identificar dónde está el error: en el estudio teórico, en la implementación, o en la forma en que se han realizado los experimentos; y actuar en consecuencia, posiblemente modificando el programa.

Otra posibilidad en el estudio experimental consiste en tomar tiempos de distintas partes del programa, para identificar las partes más costosas e intentar optimizarlas dedicándoles más tiempo de programación, utilizando por ejemplo una estructura de datos más sofisticada que nos permita llevar a cabo algunas operaciones de forma más eficiente aunque sea a costa de dificultar la programación.

Ejercicios resueltos

Ejercicio 6.1 Calcula el tiempo de ejecución de la siguiente función. Obtén también el valor final de la función.

```

operación recursiva(n:entero):entero
  si n=1 hacer
    devolver 1
  sino
    devolver 2*recursiva(n-1)
  finsi

```

Solución.

Sea *a* el tiempo constante debido a la comprobación de la condición $n = 1$, *b* el tiempo constante de la devolución de 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 devolución del resultado.

De este modo:

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

Si $n \leq 0$, no tiene sentido la llamada a la función. Suponemos, por tanto, $n > 0$.

Desarrollando:

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

El valor final de *recursiva(n)* es 2^{n-1} , lo que se demuestra por inducción:

recursiva(1)=1,
y si *recursiva(n)=2ⁿ⁻¹*, *recursiva(n+1)=2*2ⁿ⁻¹=2ⁿ*.

Ejercicio 6.2 Calcula el tiempo de ejecución de:

```

operación uno(n:entero)
  x:=0
  y:=0
  para i:=1,...,n hacer
    si par(i) entonces
      para j:=i,...,n hacer
        x:=x+1
      finpara
    sino
      para j:=1,...,i-1 hacer
        y:=y+1
      finpara
    finsi
  finpara

```

Solución.

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

El **para** consume un tiempo $\sum_{i=1}^n (b + t(i))$, donde $t(i)$ es el tiempo consumido en el cuerpo del **para** 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 **para**.

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

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

El tiempo de ejecución del procedimiento es:

si n es par:

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

y en el caso impar se procede de modo análogo.

Ejercicio 6.3 Dado el programa:

```
operación programa
  i:=1
  mientras  $i \leq n$  hacer
    si  $a[i] \geq a[n]$  entonces
       $a[n]:=a[i]$ 
    finsi
     $i:=i * 2$ 
  finmientras
```

Calcula el número promedio de asignaciones del array.

Solución.

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:

$$t(n) = \sum_{i=2}^{\log n + 1} \frac{1}{i} \approx \int_1^{\log n + 1} \frac{1}{i} di = \ln i \Big|_1^{\log n + 1} = \ln(\log n + 1)$$

Ejercicio 6.4 Dado el siguiente procedimiento:

```
operación contar( $izq, der$ :entero):entero
  si  $izq \geq der$  entonces
    devolver 0
  sino si  $a[izq] < a[der]$  entonces
    devolver contar( $izq * 2, der$ ) + 1
  sino
```

```

devolver contar(izq*2,der)
finsi
```

donde *a* es un array global de valores enteros, calcular, para valores iniciales de *izq*=1 y *der*=*n*, el 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 $\text{contar}(izq * 2, der)$, 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 **devolver**.

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:

$$t(1, n) = t(2, n) + c = t(4, n) + 2c = \dots = t(n, n) + kc = a + c \log n$$

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 de la forma $\log n$. 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.

Ejercicio 6.5 Un número capicúa está formado por una cadena de *n* dígitos, *c:array[1..n]* **de** 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 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 i ≤  $\lfloor \frac{n}{2} \rfloor$  Y c[i]=c[n-i+1] hacer
    i:=i+1
finmientras
```

devolver $i > \lfloor \frac{n}{2} \rfloor$

La probabilidad de que se cumpla $c[i] = c[n - i + 1]$ es $\frac{1}{10}$ cada vez, ya que, 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 el tiempo promedio sólo hay que contar el número promedio de veces que se pasa por el interior del bucle, porque 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 $p(c[1] = c[n] \text{ y } c[2] = c[n - 1]) = \frac{1}{10^2}$. 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 $\frac{10}{9} \left(\frac{1}{10} - \frac{1}{10^{\lfloor \frac{n}{2} \rfloor + 1}} \right) \leq \frac{1}{9}$. Por lo tanto el tiempo de ejecución es constante.

Ejercicios propuestos

Ejercicio 6.6 Dado el programa:

```

encontrado:=falso
para  $i:=1, \dots, n-1$  hacer
    si  $x[i]=y[1]$  entonces
        si  $x[i+1]=y[2]$  entonces
            encontrado:=verdadero
            lugar:=i
        finsi
    finsi
     $i:=i+1$ 
finpara

```

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

Ejercicio 6.7 Dado el programa:

```

cont:=0
para  $i:=1, \dots, n$  hacer
    para  $j:=1, \dots, i-1$  hacer
        si  $a[i] < a[j]$  entonces
             $cont:=cont+1$ 
        finsi
    finpara
finpara

```

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

Ejercicio 6.8 Dado el programa:

```

p:=0
para  $i:=1, \dots, n$  hasta
     $p:=p+i*i$ 

```

```

para  $j:=1,\dots,p$  hasta
    escribir( $a[p,j]$ )
finpara
finpara

```

Obtener el número de instrucciones que se ejecutan.

Ejercicio 6.9 Dado el programa:

```

max:=- $\infty$ 
para  $i:=1,\dots,n$  hasta
    para  $j:=1,\dots,m$  hasta
        si  $a[i,j] > max$ 
             $max:=a[i,j]$ 
        finsi
    finpara
finpara

```

Obtener una cota inferior y superior del número de veces que se ejecuta la asignación interna, y calcular el número promedio de veces que se ejecuta dicha instrucción.

Ejercicio 6.10 Obtener el número de comparaciones y de asignaciones (por separado) de elementos de los tipos que se están ordenando para los casos más favorable y más desfavorable de los dos algoritmos del ejemplo 6.2. Si para un determinado tipo de datos y sistema (máquina y compilador) el coste de las comparaciones es el doble que el de las asignaciones, y si el resto de las instrucciones suponemos que tiene un coste despreciable, determinar cuánto más rápido es ordenar2 que ordenar1 en el caso más desfavorable con $n = 100$, $n = 1.000$ y $n = 10.000$.

Cuestiones de autoevaluación

Ejercicio 6.11 Si para un algoritmo $t_m(n) = n^2$ y $t_M(n) = 2n^2$, podemos decir de $t_p(n)$ que: a) $t_p(n) = \frac{3}{2}n^2$, b) $t_p(n)$ tiene forma de parábola, c) no sabemos nada sobre $t_p(n)$.

Ejercicio 6.12 Con las suposiciones del ejemplo 6.3, estudiar el tiempo de ejecución en el caso más favorable, más desfavorable y promedio, de la búsqueda binaria en un array ordenado.

Referencias bibliográficas

Las nociones básicas sobre el estudio de tiempos de ejecución se pueden encontrar en multitud de libros clásicos de programación o algoritmos [Aho74], [Aho88], [Brassard90], [Brassard97], [Cormen90], [Horowitz78], [Wirth80] y [Wirth87].

Las nociones básicas de matemáticas necesarias para hacer el tipo de estudios que aparece en este capítulo se encuentran en libros elementales de análisis matemático, matemática discreta y estadística. Algunos libros de programación también incluyen un capítulo dedicado a las herramientas matemáticas necesarias para este tipo de estudios

y los que se harán en los capítulos siguientes. Capítulos bastante completos se pueden encontrar en el libro clásico [Knuth85] y en los más recientes [Cormen90], [Brassard97], [Heileman98] y [Baase00].

Capítulo 7

Notaciones asintóticas

El tiempo de ejecución y la ocupación de memoria de un algoritmo dependen de una serie de factores externos al algoritmo (sistema donde se ejecute, compilador usado, detalles de programación, etc.), por lo que el estudio que se realice nos dará unos parámetros cuyos valores dependen de estos factores externos. Por esto, se puede hacer el estudio independientemente de las constantes multiplicativas, y obtener sólo la forma en que crece la función tiempo o la de ocupación de memoria en función del tamaño y el contenido de la entrada. Por otro lado, los problemas de tiempo y memoria aparecen normalmente cuando el tamaño del problema aumenta, por lo que tiene más interés el estudio para tamaños grandes. En este sentido se utilizan las notaciones asintóticas que estudiamos en este capítulo: indican cómo crecen las funciones asintóticamente, es decir, cuando el tamaño del problema tiende a infinito.

Objetivos del capítulo:

- Diferenciar claramente el concepto de tiempo de ejecución y ocupación de memoria de una ejecución particular del de complejidad de un algoritmo.
- Conocer las notaciones de complejidad O (o-grande, orden), Ω , Θ y o (o-pequeña), diferenciando claramente el significado y utilidad de cada una.
- Saber calcular (y reconocer intuitivamente) la complejidad algorítmica de una gama amplia de algoritmos.
- Saber comparar las complejidades de distintas funciones.
- Conocer la complejidad de las funciones que aparecen más frecuentemente en el estudio de algoritmos, así como la relación entre las complejidades de estas funciones.

Contenido del capítulo:

7.1. Notaciones asintóticas	23
7.1.1. La notación O	24
7.1.2. La notación Ω	25
7.1.3. La notación Θ	25
7.1.4. La notación o (o-pequeña)	26
7.2. Propiedades de las notaciones asintóticas	27
7.3. Complejidades más frecuentes	29
7.4. Extensiones de las notaciones asintóticas	30
7.4.1. Notaciones con varios parámetros	30
7.4.2. Notaciones condicionales	31
Ejercicios resueltos	32
Ejercicios propuestos	41
Cuestiones de autoevaluación	43
Referencias bibliográficas	44

7.1. Notaciones asintóticas

Como ya hemos visto en el capítulo anterior, cuando se estudia un algoritmo se modeliza el tiempo de ejecución o la ocupación de memoria contando el número de instrucciones que se ejecutan o la cantidad de datos que se almacenan. Pero el coste de las instrucciones y la memoria que se necesita para almacenar un dato dependerán del sistema donde se vaya a ejecutar el programa con el que implementamos el algoritmo, del lenguaje que usemos para implementarlo, de las opciones de compilación, del tipo de datos que usemos, etc. Todos estos factores hacen que sea conveniente hacer el estudio dependiendo de unos parámetros cuyos valores reales dependerán de todos estos factores. Por tanto, lo que nos interesa saber de un tiempo de ejecución de un algoritmo es más la forma en que crece que las constantes multiplicativas. Por ejemplo, nos interesa saber si el tiempo tiene forma de parábola, más que si es la parábola $t(n) = 2n^2 + 4n + 1$, ya que esta fórmula puede representar el número de instrucciones que se ejecutan, pero el tiempo de ejecución depende del coste de las instrucciones, que pueden ser en un sistema diez veces más costosas que en otro, o si los datos con los que se trabaja son números en doble precisión más costosas que si son enteros.

Además, cuando el tamaño del problema crece, los términos de menor orden pierden importancia. Con la parábola anterior $t(2) = 17$, donde 8 corresponde al término de mayor orden, $2n^2$, y 9 a los de menor orden, $4n + 1$; pero $t(1.000) = 2.004.001$, donde 2.000.000 corresponde al término de mayor orden, y 4.001 a los de menor orden. Cuando estudiamos el tiempo nos interesa poder predecir tiempos de ejecución para los distintos tamaños del problema, pero los problemas de tiempo los tendremos con tamaños grandes. En el ejemplo, si cada instrucción se ejecuta en un microsegundo, con $n = 2$ el tiempo sería de 17 microsegundos, mientras que con $n = 1.000$ sería de más de 33 minutos, lo que ya es un tiempo considerable y puede ser un problema dependiendo del tipo de aplicación que se esté haciendo.

Por todo esto, en el estudio de algoritmos se suele utilizar **notaciones asintóticas**, que representan o acotan la forma en que crece una función (que en nuestro caso será de tiempo de ejecución o de ocupación de memoria) cuando el tamaño del problema tiende a infinito y sin tener en cuenta las constantes que la afectan.

Consideraremos funciones $t : N \rightarrow R^+$, pues las funciones que representan tiempos de ejecución u ocupación de memoria toman valores para tamaños de problema números naturales, y puede dar como resultado valores reales (el tiempo de ejecución podría ser de 1,0123456 segundos). El tamaño del problema puede ser un número en doble precisión, que no es un número natural, pero su representación en el ordenador vendrá dada por una serie de bits, con lo que el dominio de datos que se puede representar es numerable e isomorfo a N . Lo mismo pasa cuando el tiempo es función de varios parámetros (por ejemplo de las dimensiones de dos matrices rectangulares que queremos multiplicar). En este caso tendríamos $t : N^3 \rightarrow R^+$, pero de nuevo es N^3 isomorfo a N . Por tanto, por simplicidad estudiaremos funciones con dominio los naturales y rango los reales positivos. Dado que nos interesa lo que pase asintóticamente (cuando n tiende a infinito), también podemos considerar funciones que tomen valores negativos para un número finito de valores de $n \in N$, como $t(n) = n^2 - 8$, o que no estén definidas para un número finito de tamaños, como $t(n) = \log(n - 10)$.

7.1.1. La notación O

La notación O se utiliza para acotar superiormente, salvo constantes y asintóticamente, la forma en que crece una función.

Definición 7.1 Dada una función $f : N \rightarrow R^+$, 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:

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

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 $\log(n-10) \in O(\log n)$, aunque la función $f(n) = n^2 - 8$ no es una función de $N \rightarrow R^+$ pues toma valores negativos, y la función $f(n) = \log(n-10)$ tampoco es de $N \rightarrow R^+$ pues no está definida para valores menores que 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 o una medida de la memoria necesaria, en función de la dimensión de la entrada. La función t puede ser difícil de calcular, por lo que se podría 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:

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

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

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 $O(n^2 + 2n - 5) = O(n^2)$.

Supongamos b, c, d y e constantes. Con polinomios en n (variable) se cumple:

- $O(c) = O(d)$
- $O(c) \subset O(n)$
- $O(cn + b) = O(dn + e)$
- $O(p) = O(q)$ 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.

7.1.2. La notación Ω

La notación Ω se utiliza para acotar inferiormente (salvo constantes y asintóticamente) la forma en que crece una función.

Definición 7.2 Dada una función $f : N \rightarrow R^+$, 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:

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

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

$$\Omega(f) \leq \Omega(g) \Leftrightarrow \Omega(f) \subseteq \Omega(g) \Leftrightarrow \forall t \in \Omega(f), t \in \Omega(g)$$

con lo que intentamos encontrar el mayor $\Omega(f)$ tal que $t \in \Omega(f)$.

Para la notación omega sirven las mismas relaciones que para el orden, pero cambiando el sentido de las desigualdades. Con polinomios se cumple (las constantes son positivas y el coeficiente principal de los polinomios también):

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

Hay una gran diferencia en la utilidad de las dos notaciones anteriores. Mientras la notación O nos da una cota superior que nos sirve para descartar posibles problemas, la notación Ω nos sirve para detectar problemas que se darían seguro. Por ejemplo, si la ocupación de memoria está acotada superiormente por n^2 y tenemos suficiente memoria para almacenar 2.048^2 datos, estamos seguros de que no tendremos problemas de memoria hasta $n = 2.048$, aunque puede que sigamos sin tener problemas para tamaños mayores. Si queremos seguir sin tener problemas de memoria hasta $n = 4.096$, basta con que pongamos cuatro veces más memoria en nuestro sistema, ya que $4.096^2 = (2*2.048)^2 = 4*2.048^2$. Si n^2 es una cota inferior, al tener espacio para 2.048^2 datos, es muy posible que tengamos problemas de memoria para tamaños mayores que 2.048 e incluso algo menores.

7.1.3. La notación Θ

Las notaciones O y Ω sirven para acotar superior e inferiormente una función, pero lo ideal sería saber la forma que tiene la función, lo cual no es siempre posible en el estudio de funciones tiempo de ejecución u ocupación de memoria. En los casos en que la cota superior e inferior coinciden, sabemos la forma exacta en que crece la función. En estos casos en que se puede obtener la forma en que crece (asintóticamente y salvo constantes) se utiliza la notación **orden exacto**.

Definición 7.3 Dada una función $f : N \rightarrow R^+$, 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:

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

Se puede comprobar fácilmente que la definición de Θ es equivalente a la siguiente definición alternativa, similar a las dadas para O y Ω :

Definición 7.4

$$\Theta(f) = \{t : N \rightarrow R^+ / \exists c, d \in R^+, \exists n_0 \in N, \forall n \geq n_0 : cf(n) \leq t(n) \leq df(n)\}$$

Cuando se estudia un algoritmo contando el número de instrucciones y se obtiene que $t(n) = n^2 + 2n - 5$, se sabe que su orden exacto es $\Theta(n^2)$, independientemente del coste de cada una de las instrucciones. 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. Si podemos obtener el número de instrucciones en el caso más favorable, $t_m(n)$, tenemos una cota inferior del tiempo, y por tanto $t \in \Omega(t_m)$. Si obtenemos t_M , lo que tenemos es $t \in O(t_M)$.

No tenemos relación de inclusión entre los conjuntos de órdenes exactos, pues el orden exacto nos establece una relación de equivalencia en el conjunto de las funciones, y los órdenes de dos funciones, o son iguales o son disjuntos¹.

7.1.4. La notación o (o-pequeña)

Algunas veces no interesa perder la información de los coeficientes del término de mayor orden. En este caso se utiliza la notación **o-pequeña**. Esto ocurre, por ejemplo, si queremos comparar dos algoritmos que tienen el mismo orden exacto. En este caso uno puede ser el doble de rápido que el otro si el coeficiente principal que aparece en su tiempo de ejecución es la mitad del que aparece en el tiempo del otro algoritmo. Para hacer este tipo de comparaciones es necesario hacer algunas suposiciones sobre los tiempos de ejecución, pues que uno sea $t_1(n) = 2n^2$ y otro $t_2(n) = 4n^2$ no quiere decir obligatoriamente que el primer algoritmo sea el doble de rápido que el segundo. Puede ser que todas las operaciones del primer algoritmo sean sobre números en doble precisión, y las del segundo sean n^2 sobre datos en doble precisión y $3n^2$ sobre datos enteros. Si en el sistema en que se ejecutan el trabajo con enteros es cuatro veces más rápido que el trabajo con datos en doble precisión, será más rápido el segundo algoritmo. Por tanto, para hacer una comparación utilizando el término de mayor orden, será necesario suponer que todas las instrucciones que se están contando tardan lo mismo, o hacer el conteo de

¹Por ejemplo, todas las parábolas (con coeficiente principal positivo porque consideramos funciones con rango en R^+) están en el mismo orden exacto que representamos por $\Theta(n^2)$, y todas las cúbicas en el de $\Theta(n^3)$, pero la intersección de los dos conjuntos es el conjunto vacío.

instrucciones dependiendo de parámetros que aparecerán en el término de mayor orden. Con los dos algoritmos anteriores tendríamos $t_1(n) = 2dn^2$ y $t_2(n) = (d + 3e)n^2$, con d el coste de una operación con datos de doble precisión y e el de las operaciones con enteros.

Definición 7.5 Dada una función $f : N \rightarrow R^+$, llamamos **o pequeña de f** al conjunto:

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

En el caso en que t sea un polinomio de grado m , $t(n) = a_m n^m + \dots + a_1 n + a_0$, es $t \in O(n^m)$ y $t \in o(a_m n^m)$, con lo que vemos que se tiene en cuenta el término de mayor orden.

Tampoco tenemos una relación de orden entre los conjuntos de las o-pequeñas, ya que esta notación también establece una relación de equivalencia en el conjunto de las funciones. Lo que sí tenemos es la inclusión $o(f) \subset \Theta(f)$.

7.2. Propiedades de las notaciones asintóticas

En este apartado vemos algunas de las propiedades de las notaciones asintóticas. Algunas son consecuencia de la relación de orden entre conjuntos, y otras son bastante intuitivas, por lo que no haremos todas las demostraciones. Las propiedades con la notación O tienen su equivalente con la Ω .

Propiedad 7.1 Transitividad

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 $f \in \Omega(h)$.

demuestracción:

Porque $f \in O(g) : \exists c \in R^+ \text{ y } n_0 \in N / \forall n > n_0, f(n) \leq cg(n)$

Porque $g \in O(h) : \exists d \in R^+ \text{ y } n_1 \in N / \forall n > n_1, g(n) \leq dh(n)$

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 $f \in O(h)$

Como ya hemos dicho, algunas propiedades son muy intuitivas. Esto pasa con la anterior, pues si f está acotada superiormente por g (salvo constantes y asintóticamente) y g superiormente por h , f estará acotada superiormente por h . Consecuencia inmediata de la propiedad anterior es la siguiente.

Propiedad 7.2 Si $f \in O(g)$ entonces $O(f) \subseteq O(g)$.

Si $f \in \Omega(g)$ entonces $\Omega(f) \subseteq \Omega(g)$.

Propiedad 7.3 Relación pertenencia/contenido

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

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

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

demuestracción:

i)

 $\Rightarrow:$

$$f \in O(f) = O(g) \text{ y } g \in O(g) = O(f),$$

 $\Leftarrow:$

$f \in O(g) \Rightarrow$ (por la propiedad 7.2) $O(f) \subseteq O(g)$, y la otra inclusión igual.

ii)

 $\Rightarrow:$

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

 $\Leftarrow:$

$f \in O(g) \Rightarrow$ (por la propiedad 7.2) $O(f) \subseteq O(g)$.

La relación de inclusión entre órdenes y entre omegas es una relación de orden parcial, por lo que hay funciones que no cumplen $f \in O(g)$ ni $g \in O(f)$. Por ejemplo:

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

y

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

Propiedad 7.4 Propiedad del máximo

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

demostración:

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

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

Por la propiedad 7.3 i), se deduce que $O(f+g) = O(\max(f, g))$.

De la propiedad 7.4 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 bucle 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.

Lo mismo pasa con el Ω , que basta obtenerlo de la parte más larga del algoritmo. Quizás esto choque con la idea de que la parte más corta del algoritmo será una cota inferior, y por tanto debería ser $\Omega(f+g) = \Omega(\min(f, g))$. Esto no es así pues la parte menor es una cota inferior del total, por lo que sólo se cumple $\Omega(\min(f, g)) \subseteq \Omega(f+g)$, pero la otra inclusión no es cierta.

Propiedad 7.5 Equivalencia entre notaciones

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 7.3.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 definición 7.4, $\exists d, e \in R^+, \exists n_0 \in N / \forall n \geq n_0 : df(n) \leq t(n) \leq ef(n)$, por lo que a partir de un cierto n_0 se cumple que $\frac{d}{b}g(n) \leq t(n) \leq eg(n)$, y por la definición 7.4, $t \in \Theta(g)$, 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.

La siguiente propiedad nos da una forma de comparar dos funciones obteniendo la relación entre sus órdenes. La comparación se hará calculando límites cuando el tamaño tiende a $+\infty$, pues sólo tienen sentido tamaños positivos. El caso de la notación o-pequeña no se incluye porque en su misma definición aparece ya la obtención de un límite.

Propiedad 7.6 Relación límites/órdenes

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

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

$$\text{ii) } \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \Rightarrow O(f) \subset O(g) \text{ y } \Omega(g) \subset \Omega(f).$$

$$\text{iii) } \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = +\infty \Rightarrow \Omega(f) \subset \Omega(g) \text{ y } O(g) \subset O(f).$$

demonstración:

i) Todas las igualdades son equivalentes por la propiedad 7.5, 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 7.3 se demuestra.

ii) $\exists \delta/f < \delta g$, a partir de un cierto n_0 , y esto implica que $f \in O(g)$. Se deduce de la propiedad 7.3.

Además, no puede ser $O(f) = O(g)$, pues sería $g \in O(f)$, y por tanto $\exists c \in R^+$ con $g \leq cf$, de donde $\frac{f}{g} \geq \frac{1}{c}$, con lo que el límite no puede ser cero.

iii) $\forall M, f > Mg$, a partir de un cierto n_0 , y esto implica que $f \in \Omega(g)$. Se deduce de la propiedad 7.3.

De forma similar al caso anterior se comprueba que $\Omega(f) \neq \Omega(g)$.

7.3. Complejidades más frecuentes

Ya hemos comparado los órdenes de polinomios. El orden de un polinomio $a_nx^n + \dots + a_0$, con a_n positivo es $O(x^n)$.

El orden de un sumatorio $\sum_{i=1}^m i^n$ es el del polinomio $O(m^{n+1})$. Esto se puede demostrar aproximando $\sum_{i=1}^m i^n \approx \int_0^m x^n dx$. Calculando la integral tendremos también el coeficiente del término de mayor orden, que será $o(\frac{1}{n+1}m^{n+1})$.

La medida logarítmica 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 $O(\log_2 n)$. Cuando se hace una operación para $\frac{n}{3}$, otra para $\frac{n}{9}$, y así sucesivamente, aparece la complejidad $O(\log_3 n)$. Dado que dos logaritmos se diferencian en una constante ($\log_a c = \log_b c * \log_b a$) utilizaremos, para los órdenes, normalmente log sin indicar la base, y en algunos casos usaremos logaritmos neperianos, pues con ellos es más fácil calcular límites y derivadas.

Las medidas más frecuentes que aparecen en el estudio de los algoritmos son las polinómicas y logarítmicas y producto de éstas. También aparecen en algunos casos raíces, y exponenciales y factoriales en tiempos de ejecución de algoritmos inaplicables cuando el tamaño del problema crece.

Algunas relaciones entre órdenes de complejidad son: $O(1) \subset O(\log n) \subset O(\sqrt{n}) \subset O(n) \subset O(n \log n) \subset O(n \log n \log n) \subset O(n^2) \subset O(n^3) \subset \dots \subset O(2^n) \subset O(n!) \subset O(n^n)$.

Con la notación Ω se invierten las inclusiones.

Todas las inclusiones se pueden demostrar utilizando la propiedad 7.6.

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

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

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

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

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

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

En general, con p y q enteros positivos se cumple $O((\log n)^p) \subset O(n^{\frac{1}{q}})$. Esto se demuestra aplicando reiteradamente la propiedad 7.6.

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

7.4. Extensiones de las notaciones asintóticas

7.4.1. Notaciones con varios parámetros

Puede ocurrir que tengamos un algoritmo en el que el tiempo de ejecución (o la ocupación de memoria) dependa de más de un parámetro. Esto ocurre, por ejemplo, en la multiplicación de matrices rectangulares de dimensiones $n \times m$ y $m \times r$, donde el tiempo dependerá de n , m y r ; o en algoritmos que trabajen con grafos con n nodos y a aristas, donde el tiempo puede ser función de n y a . En estos casos se pueden usar notaciones asintóticas con varios parámetros. La definición del orden sería:

Definición 7.6 Dada una función $f : N^m \rightarrow R^+$, 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, \dots, n_m) suficientemente grandes. Se denota $O(f)$, y será:

$$O(f) = \{t : N^m \rightarrow R^+ / \exists c \in R^+, \exists n_1, n_2, \dots, n_m \in N, \\ \forall k_1 \geq n_1, k_2 \geq n_2, \dots, k_m \geq n_m : t(k_1, k_2, \dots, k_m) \leq cf(k_1, k_2, \dots, k_m)\}$$

De manera similar se pueden definir las notaciones Ω y Θ de funciones con varios parámetros.

En la práctica, para obtener el orden de una función hay que tomar los términos de mayor orden de cada variable y los de mayor orden de varias variables juntas.

Ejemplo 7.1 Como ejemplo consideramos la función $t(n, m, r) = nmr + mr^2 + nr + n^3 + n^2r$. Para obtener su orden exacto hay que decidir qué monomios están acotados por otros. nmr no está acotado por ningún otro monomio, pues en ninguno aparecen las tres variables. mr^2 no está acotado por ninguno porque en el único en que aparecen m y r es el nmr , donde r aparece con un exponente menor. Lo mismo ocurre con n^3 y n^2r . El único acotado superiormente es nr , que está acotado por el n^2r y el nmr . Por tanto, $t(n, m, r) \in O(nmr + mr^2 + n^3 + n^2r)$. Por otro lado, $t(n, m, r) \in \Omega(nmr + mr^2 + n^3 + n^2r)$, ya que los valores son positivos. Podemos concluir que $t(n, m, r) \in \Theta(nmr + mr^2 + n^3 + n^2r)$.

Como hemos visto en el ejemplo, pueden aparecer funciones con varios parámetros de los que no podemos eliminar la mayoría de los términos para obtener sus notaciones. Trabajar con varios parámetros puede ser engoroso, por lo que muchas veces se reduce el estudio a funciones con un único parámetro:

- Se pueden estudiar los órdenes en cada una de las variables considerando las demás fijas.
- Otra posibilidad es considerar todas las variables con el mismo valor. Esto tiene sentido en el caso de la multiplicación de matrices, donde se puede estudiar matrices cuadradas, con lo que $n = m = r$. No tiene sentido en el estudio de algoritmos sobre grafos, pues el número de aristas y el de nodos no será igual.
- En algunos casos se puede restringir el estudio a algunos tipos de problemas. Con grafos podemos estudiar grafos con pocas aristas (dispersos o escasos), con lo que $a \approx 0$, o con muchas aristas (densos), con $a \approx n^2$.

7.4.2. Notaciones condicionales

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

Definición 7.7 Dada una función $f : N \rightarrow R^+$ y $P : N \rightarrow B$, donde B es el tipo booleano, definimos el **orden de f según P** como:

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

Se cumple la inclusión $O(f) \subseteq O(f|P)$.

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 condición del tipo $\exists k \in N, n = b^k$, es decir, n es potencia de b , y posteriormente quitar la restricción utilizando el siguiente teorema:

Teorema 7.1 Si $b \geq 2$ es un entero y $f : N \rightarrow R^+$ 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 $t : N \rightarrow R^+$ es eventualmente no decreciente tal que $t(n) \in \Theta(f|n$ potencia de $b)$, entonces $t(n) \in \Theta(f)$.

Ejemplo 7.2 En el estudio del tiempo de la ordenación por mezcla tenemos la ecuación $t(n) = 2t(\frac{n}{2}) + bn$, pero esta ecuación es válida sólo si el tamaño del problema es potencia de dos. En el caso general se tiene $t(n) = t(\lfloor \frac{n}{2} \rfloor) + t(\lceil \frac{n}{2} \rceil) + bn$, donde $\lfloor \cdot \rfloor$ representa al entero más próximo menor y $\lceil \cdot \rceil$ al entero más próximo mayor.

Se puede obtener el orden exacto para tamaños potencia de dos: $t(n) \in \Theta(n \log n|n$ es potencia de 2).

A partir de esto, y utilizando el teorema 7.1, 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.

Hay que demostrar que la función t general (no la restringida a tamaños de la entrada potencia de dos) es eventualmente no decreciente, pero sólo conocemos t por su ecuación. 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 de manera similar se demuestra la desigualdad para n impar.

Ejercicios resueltos

Ejercicio 7.1 Obtener O y Ω , y Θ del tiempo promedio para el algoritmo de búsqueda binaria:

```

i:=1
j:=n
repetir
  m:= $\frac{i+j}{2}$ 
  si a[m]>x entonces
    j:=m-1
  sino
    i:=m+1
  finsi
hasta i>j O 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 $t(n) = 2 + 4 + t_r(\frac{n}{2}) = 2 + 4 + 4 + t_r(\frac{n}{4}) = \dots = 2 + 4(\log n + 1)$, con lo que $t \in O(\log n|n$ potencia de 2).



Para quitar la restricción de que n sea potencia de 2 se aplicará el teorema 7.1. $\log n$ es creciente y 2-armónica, pues $\log 2n = 1 + \log n \in O(\log n)$. Además, $t(n) = 6 + t_r(\lfloor \frac{n-1}{2} \rfloor)$ o $t(n) = 6 + t_r(\lfloor \frac{n}{2} \rfloor)$, 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:

$$2 + 4 \left(\frac{p}{n} + 2 \frac{2p}{n} + 3 \frac{4p}{n} + \dots + \log n \frac{2^{\log n - 1} p}{n} \right) = 2 + 4 \frac{p}{n} \sum_{i=1}^{\log n} i 2^{i-1} \approx 2 + 2 \frac{p}{n} \int_0^{\log n} x 2^x dx$$

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

Ejercicio 7.2 Tenemos claves formadas por una serie no limitada de campos (*campo₁*, *campo₂*, ...), 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 *campo₁*, si son iguales comparando el *campo₂*, etc. 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 habrá 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 $\frac{1}{2}$. Por lo tanto, el número de comparaciones es una $\Theta(1)$.

Ejercicio 7.3 Calcular o (o-pequeña) del número promedio de asignaciones a la variable max del algoritmo:

```
max:=a[1]
para i:=2,...,n hacer
    si a[i]>max entonces
        max:=a[i]
    finsi
finpara
```

Solución.

La primera asignación se ejecuta siempre, y la segunda (la interior al **si**) se ejecutará o no dependiendo de si $a[i] > max$, por lo que el número de asignaciones 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 $\lim_{n \rightarrow \infty} \frac{1 + \ln n}{1 + \ln(n+1) - \ln 2} = \lim_{n \rightarrow \infty} \frac{\frac{1}{n}}{\frac{1}{n+1}} = 1$ las dos cotas tienen el mismo o y $A(n) \in o(\ln n)$.

Ejercicio 7.4 a) Dado el programa:

```
i:=1
j:=n
mientras i<j hacer
    si a[i]<a[j] entonces
        i:=i*2
    sino
        j:=j/2
    finsi
finmientras
```

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 **mientras** 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 $t_w(i, j) = 3 + t_w(i, \frac{j}{2})$. En cualquier caso se ejecutan tres instrucciones por cada pasada por el **mientras**, 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 **mientras** sean cuales sean los datos en el array, por lo que $t(n) \in \Theta(\log n | n = 2^k)$.

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 **mientras**; 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 **mientras** 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 **mientras** independientemente de la distribución de los datos en el array, por lo que $t(n) \in \Theta(n)$.

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

Ejercicio 7.5 Dado el programa:

```

cont:=0
para  $i:=1,\dots,n$  hacer
     $j:=1$ 
    mientras  $j < i$  hacer
        si  $a[i] < a[j]$  entonces
             $cont:=cont+1$ 
             $j:=j*2$ 
        sino
             $j:=j+1$ 
        finsi
    finmientras

```

finpara

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 7.1.

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 **para** y la comprobación de salida del **mientras**; y t_w es el tiempo empleado en el **mientras**.

El caso más favorable será cuando $a[i] < a[j]$ cada vez que se pase por la condición del **si**. En este caso j varía multiplicándose por dos en cada paso, con lo que $t_m(n) \geq a + \sum_{i=1}^n (b + c \log i)$, siendo c la constante correspondiente a la comprobación del **mientras** 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 $t(n) \in \Omega(n \ln n | n = 2^k)$.

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 **para** que con tamaño n , más la última pasada correspondiente al valor $i = n+1$.

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

Al no coincidir Ω y O no existe un orden exacto para $t(n)$.

Ejercicio 7.6 Dado el siguiente algoritmo para obtener el segundo mayor elemento en un array:

```

max1 := -∞
max2 := -∞
para i:=1,...,n hacer
  si a[i]>max1 entonces
    max2 := max1
    max1 := a[i]
  sino si a[i]>max2 entonces
    max2 := a[i]
  finsi
finpara

```

- Obtener Θ del algoritmo obteniendo O y Ω .
- Calcular σ (σ -pequeña) del número promedio de instrucciones que se ejecutan, suponiendo que cada línea en el programa es una instrucción.
- Dar un programa distinto para resolver el mismo problema, y que tenga una σ (σ -pequeña) 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 **para** con valores de $i = 1$ a n , por lo que $t(n) = a + \sum_{i=1}^n t_{\text{para}}(i)$, donde a es una constante y dentro del **para** 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 $t_{\text{para}}(i)$ 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   para i:=1,...,n hacer
4     si a[i]>max1 entonces
5       max2 := max1
6       max1 := a[i]
7     sino si a[i]>max2 entonces
8       max2 := a[i]
9   finsi
10  finpara

```

y no consideraremos las líneas 9 y 10.

Las líneas 1 y 2 se ejecutan una vez cada una, la 4 se ejecuta n veces, y la 3 consideraremos que se ejecuta $n+1$ veces: las n veces que se entra dentro del **para** 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 $2(1 + \sum_{i=2}^n \frac{1}{i})$. 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 $\sum_{i=2}^n (1 - \frac{1}{i})$. 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:

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

Como en las dos cotas el 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 aumentar i es cada vez más improbable que $a[i] > max2$.

El programa podría ser:

```
max1 := -∞
max2 := -∞
para i:=1,...,n hacer
  si a[i]>max2 entonces
    si a[i]>max1 entonces
      max2 := max1
      max1 := a[i]
    sino
      max2 := a[i]
  finsi
finpar
```

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

Ejercicio 7.7 Dado el siguiente esquema:

```
der:=0
izq:=0
para i:=1,...,n hacer
  si par(a[i,i]) entonces
    para j:=i+1,...,n hacer
      si a[i,j]>0 entonces
        der++
      finsi
    finpar
  sino
    para j:=1,...,i-1 hacer
      si a[i,j]>0 entonces
        izq++
      finsi
    finpar
  finsi
finpar
```

calcular:

- Θ del tiempo de ejecución.
- o (o-pequeña) del número promedio de instrucciones que se ejecutan.

Solución.

a) El tiempo de ejecución se puede poner en la forma $t(n) = a + \sum_{i=1}^n t_{\text{para}}(i)$, donde $t_{\text{para}}(i)$ representa el tiempo de ejecución en el paso por el **para** con índice i . Dentro del **para** se ejecuta un segundo **para** con índice j , que puede variar de $i+1$ a n o de 1 a $i-1$.

Lo más favorable es ejecutar el **para** 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 $t_m(n) = a + \sum_{i=1}^{\frac{n}{2}} \left(b + \sum_{j=1}^{i-1} c \right) + \sum_{i=\frac{n}{2}+1}^n \left(b + \sum_{j=i+1}^n c \right) = a + bn + \sum_{i=1}^{\frac{n}{2}} c(i-1) + \sum_{i=\frac{n}{2}+1}^n c(n-i) = a + bn + c\frac{n^2}{4} - c\frac{n}{2} \in \Theta(n^2)$, con lo que $t(n) \in \Omega(n^2)$.

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 $t_M(n) = a + \sum_{i=1}^{\frac{n}{2}} \left(b + \sum_{j=i+1}^n c \right) + \sum_{i=\frac{n}{2}+1}^n \left(b + \sum_{j=1}^{i-1} c \right) = a + bn + c\frac{3n^2}{4} - c\frac{n}{2} \in \Theta(n^2)$, con lo que $t(n) \in O(n^2)$. 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 $t(n) \in \Theta(n^2)$.

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

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

Ejercicio 7.8 Dado el esquema:

```

m:=a[1]
i:=2
mientras i<=n Y a[i]>m hacer
    m:= $\frac{m*(i-1)+a[i]}{i}$ 
    i++
finmientras

```

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 **mientras**, 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 entra, las dos actualizaciones del **mientras**, y la última comparación del **mientras** 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 **mientras**, por lo que se sale de este al ser $i > n$. Por lo tanto, se entra en el **mientras** 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 **mientras** (para comprobar que no se entra); el segundo 3 corresponde a las dos actualizaciones internas al **mientras** y a la comprobación de este; 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 $j = 2, 3, \dots, i$. 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 $p(a[i] > m) = (\frac{1}{2})^{i-1}$. Por tanto la fórmula será:

$$t_p = 3 + 3 \sum_{i=2}^n \left(\frac{1}{2}\right)^{i-1} = 3 + 3 \frac{\frac{1}{2} - \left(\frac{1}{2}\right)^n}{1 - \frac{1}{2}} = 6 - \frac{3}{2^{n-1}}$$

y $t_p(n) \in \Theta(1)$, y $t_p(n) \in o(6)$.

Ejercicio 7.9 Calcula el orden exacto del tiempo promedio de ejecución de la función:

```

operación funcion2(a:array[1..n] de entero; i,d:entero):entero
  si i ≥ d entonces
    devolver 1
  sino
    si par(a[i]) Y par(a[d]) entonces
      i := i +  $\frac{d-i}{2}$ 
      devolver (1+funcion2(a,i,d))
    sino
      d := d -  $\frac{d-i}{2}$ 
      devolver (funcion2(a,i,d))
    finsi
  finsi

```

cuando se llama con $\text{funcion2}(a, 1, n)$. 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 **si** o por la segunda, se ejecutan las mismas operaciones de comprobación del **si**, actualización de un índice, y

el **devolver**, todo esto llevará un tiempo constante (b), y la única diferencia es que si la condición del **si** 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 **si**. 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:

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

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

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

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 $t(n) \in \Theta(\log n | n \text{ potencia de 2})$.

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 $i = 2, 3, \dots, n$, 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 $t(n+1) = t\left(\frac{n+1}{2}\right) + k > t\left(\frac{n+1}{2} - 1\right) + k = t(n)$.

Ejercicios propuestos

Ejercicio 7.10 Calcula una buena cota superior de la complejidad del algoritmo:

```

operación dos(a:array[1..n] de entero;x,y:entero)
    si x≠y entonces
        si 2a[x]<a[y] entonces
            dos(a,x,y/2)
        sino
            dos(a,2x,y)
        finsi
    finsi

```

suponiendo que los datos en el array están ordenados de manera creciente, y valores iniciales $x = 1$, $y = n$.

Ejercicio 7.11 Dado el programa:

```

i:=1
j:=1
mientras i≤n Y j≤n hacer
    si a[i,j+1]<a[i+1,j] entonces
        j:=j+1
    sino
        i:=i+1
    finsi
finmientras

```

Calcular su Ω , O y Θ .

Ejercicio 7.12 Dado el programa:

```

max:=−∞
para i:=1,...,n hacer
    para j:=1,...,m hacer
        si a[i,j]>max entonces
            max:=a[i,j]
        finsi
    finpara
finpara

```

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

Ejercicio 7.13 a) Calcular el orden exacto y la función o (o pequeña) de la función:

```

operación funcion1(n:entero):entero
    si n=1 entonces
        devolver 1
    sino
        devolver (2*funcion1(n-1)+1)
    finsi

```

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?

Ejercicio 7.14 Dado el esquema:

```

para j:=1,...,n hacer
    para i:=1,...,j hacer
        si a[i,j]<a[1,j] entonces
            a[1,j]:= a[i,j]
        finsi
    finpara
finpara
para i:=1,...,n hacer
    si a[1,i]<a[1,1] entonces
        a[1,1]:= a[1,i]
    finsi
finpara

```

Estudiar su O y Ω , cotas para la constante en la notación o , y o del número promedio de asignaciones de elementos del array.

Ejercicio 7.15 a) Encontrar O y Ω del algoritmo:

```

max:=0
para i:=1,...,n hacer
    cont:=1
    j:=i+1
    mientras j≤n Y a[i]≤a[j] hacer
        j:=j+1
        cont:=cont+1
    finmientras
    si cont>max entonces
        max:=cont
    finsi
finpara

```

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

Ejercicio 7.16 Dado el programa:

```

para i:=1,...,n hacer
    j:=i+1
    mientras j≤n Y a[j]<a[i] hacer
        a[i]:=a[j]
        j++
    finmientras
finpara

```

- a) Obtener O del programa.
- b) Obtener Ω del programa.
- c) Obtener Θ del número promedio de instrucciones ejecutadas.

Cuestiones de autoevaluación

Ejercicio 7.17 Llamamos t al tiempo de ejecución de un algoritmo, t_m al tiempo en el caso más favorable, t_M el tiempo en el caso más desfavorable, y t_p al tiempo promedio. Decir si son ciertas o falsas las siguientes afirmaciones:

- a) $t_p \in O(t_M)$
- b) $t \in \Omega(t_M)$
- c) $t_m \in O(t_M)$
- d) $t_m \in \Theta(t_M) \Rightarrow t_p \in \Theta(t_m)$
- e) $t_m \in \Omega(t_M) \Rightarrow t \in \Theta(t_m)$
- f) En cualquier algoritmo $t_m \neq t_M$
- g) $(t_M)^2 \in \Omega(t_M)$
- h) $(t_p)^2 \in \Omega(t_p)$
- i) $t_p + t_m + t_M \in \Theta(t_M)$
- j) $t_p \in \Theta(\frac{t_M+t_m}{2})$

Ejercicio 7.18 Ordenar los órdenes y los omega de las siguientes funciones: $\ln(n+1)$, $\ln \log n$, $n^{\frac{1}{4}} + \log n$, $n \log n - n \ln n$, $\ln \log n + \log \ln n$. Identificar las funciones que tienen el mismo orden exacto.

Ejercicio 7.19 Calcular la o-pequeña de las siguientes funciones:

- a) $t(n) = \sum_{i=1}^n i^3$
- b) $t(n) = \sum_{i=1}^n \left(i \sum_{j=1}^i j \right)$
- c) $t(n) = \sum_{i=1}^{\log n} i^2$
- d) $t(n) = \sum_{i=1}^n i \ln i$
- e) $t(n) = n^4 - \sum_{i=1}^{\frac{n}{2}} i^3$

Ejercicio 7.20 Obtener O y Ω para el algoritmo de multiplicación de matrices:

```

para i:=1,...,n hacer
    para j:=1,...,n hacer
        suma:=0
        para k:=1,...,n hacer
            suma:=suma+a[i,k]*b[k,j]
        finpara
        c[i,j]:=suma
    finpara
finpara

```

Ejercicio 7.21 ¿Qué relación existe entre $O(t(n)|n \text{ es par})$ y $O(t(n)|n \text{ es potencia de } 2)$? En general, ¿qué relación existe entre el orden condicionado de una función, usando una condición más o menos débil?

Ejercicio 7.22 Si $t(m, n, r) = n^2 + mr + m$, contestar si son verdaderas o falsas las siguientes afirmaciones:

- a) $t(m, n, r) \in \Theta(n^2)$.
- b) $t(m, n, r) \in O(n^2)$.
- c) $t(m, n, r) \in \Theta(n^2 + mr)$.
- d) $t(m, n, r) \in O(n^2mr)$.

Referencias bibliográficas

Nociones sobre complejidad algorítmica se pueden encontrar en [Aho88], [Weis95], y capítulos más extensos en [Baase00], [Brassard90], [Brassard97], [Cormen90] y [Peña98].

El cálculo de límites se puede consultar en libros básicos de análisis matemático.

Capítulo 8

Ecuaciones de recurrencia

En el estudio de algoritmos aparecen frecuentemente ecuaciones de recurrencia, en las que el tiempo (o la ocupación de memoria) para un cierto tamaño de problema se pone en función del tiempo para tamaños menores. Este tipo de ecuaciones las encontramos en algoritmos recursivos, pero también pueden plantearse en programas iterativos. En este capítulo se estudian las técnicas básicas de resolución de ecuaciones de recurrencia, centrándonos en los tipos que aparecen normalmente en el estudio de algoritmos.

Objetivos del capítulo:

- Entender el significado de las ecuaciones de recurrencia.
- Conocer las técnicas básicas de resolución de ecuaciones de recurrencia: expansión de la recurrencia, método de la ecuación característica y utilización de fórmulas maestras.
- Comprender el papel que desempeñan las condiciones iniciales (casos base) en la resolución de ecuaciones recurrentes, y saber determinar cuáles se deben aplicar para cierta ecuación dada.
- Identificar las ecuaciones más frecuentes en el estudio de algoritmos y los costes a que dan lugar.

Contenido del capítulo:

8.1. Ecuaciones de recurrencia	47
8.2. Expansión de la recurrencia	48
8.3. Método de la ecuación característica	49
8.3.1. Ecuaciones lineales homogéneas	49
8.3.2. Ecuaciones lineales no homogéneas	51
8.3.3. Cambio de variable	52
8.3.4. Transformación de la imagen	52
8.4. Técnica de la inducción constructiva	53
8.5. Fórmulas maestras	54
Ejercicios resueltos	56
Ejercicios propuestos	57
Cuestiones de autoevaluación	58
Referencias bibliográficas	58

8.1. Ecuaciones de recurrencia

En el análisis de algoritmos se suele llegar a ecuaciones de recurrencia que se presentan en como:

$$t(n) = \begin{cases} b & \text{si } n \leq n_0 \\ g(t(n), t(n-1), \dots, t(n-k), n) & \text{si } n > n_0 \end{cases} \quad (8.1)$$

Estas ecuaciones aparecen normalmente en el estudio de algoritmos recursivos, pero muchas veces se pueden plantear en el estudio de tiempos de algoritmos iterativos, cuando se pone el tiempo hasta la iteración i en función del tiempo hasta la iteración $i - 1$.

Ejemplo 8.1 En el algoritmo recursivo de las Torres de Hanoi tenemos:

```

operación Hanoi( $n, i, j, k$ : entero) //Paso de  $n$  aros de  $i$  a  $j$  teniendo a  $k$  de pivote
  si  $n=1$  entonces
    mover( $i, j$ )
  sino
    Hanoi( $n-1, i, k, j$ )
    mover( $i, j$ )
    Hanoi( $n-1, k, j, i$ )
  finsi
```

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

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

lo que da lugar a una ecuación de recurrencia.

Ejemplo 8.2 Usando un algoritmo iterativo para calcular el factorial de un número, podemos poner $t(n) = a + t_{\text{para}}(n)$, y plantear una ecuación de recurrencia para la obtención de t_{para} :

$$t_{\text{para}}(n) = \begin{cases} b & \text{si } n = 1, 2 \\ t_{\text{para}}(n-1) + c & \text{si } n > 2 \end{cases}$$

Las técnicas que veremos para la resolución de ecuaciones de recurrencia son:

- **Expansión de la recurrencia.** Esta es la técnica utilizada en los ejemplos de los capítulos anteriores, ya que es el método más intuitivo.
- **Método de la ecuación característica.** Consiste en obtener, a partir de la ecuación de recurrencia y de manera sistemática, una ecuación polinómica cuyas soluciones proporcionan la solución de la ecuación de recurrencia.
- **Utilización de fórmulas maestras.** En el estudio de algoritmos se repiten con mucha frecuencia ecuaciones de algunos tipos básicos, por lo que se podría tener la solución de estos tipos y, dado un algoritmo, identificar a qué tipo pertenece su ecuación.

8.2. Expansión de la recurrencia

Algunas ecuaciones de recurrencia se pueden resolver expandiendo la recurrencia hasta llegar a un caso base. Después de algunos pasos de expansión de la recurrencia se obtiene una fórmula general que relaciona la función para el tamaño original con otros tamaños menores, y a partir de esta fórmula general se obtiene la fórmula que la relaciona con el caso base. Puede ser necesario demostrar por inducción las fórmulas, aunque en muchos casos la relación es evidente y no es necesaria la demostración por inducción.

Ejemplo 8.3 Expandimos la recursión que aparece en el problema de las Torres de Hanoi:

$$\begin{aligned}t(n) &= 2t(n-1) + 1 = 2(2t(n-2) + 1) + 1 = 2^2t(n-2) + 1 + 2 = \\&= 2^2(2t(n-3) + 1) + 1 + 2 = 2^3t(n-3) + 1 + 2 + 2^2\end{aligned}$$

Aquí podríamos deducir que la fórmula general es:

$$t(n) = 2^k t(n-k) + 1 + 2 + 2^2 + \dots + 2^{k-1} = 2^k t(n-k) + 2^k - 1$$

La demostración rigurosa se podría hacer por inducción. Para $k = 1$ tenemos la ecuación de recurrencia original. Si suponemos que se cumple para k pasos, hay que demostrar que la fórmula sigue siendo válida para $k+1$:

$$\begin{aligned}t(n) &= 2^k t(n-k) + 2^k - 1 = 2^k(2t(n-(k+1)) + 1) + 1 + 2 + \dots + 2^{k-1} = \\&= 2^{k+1}t(n-(k+1)) + 1 + 2 + \dots + 2^k = 2^{k+1}t(n-(k+1)) + 2^{k+1} - 1\end{aligned}$$

Se llega al caso base 1 cuando $n-k=1$, con lo que $k=n-1$, y

$$t(n) = 2^{n-1}t(1) + 2^{n-1} - 1 = 2^n - 1 \in \Theta(2^n)$$

No en todos los casos es tan simple la expansión de la recurrencia, y pueden ser preferibles los otros métodos que vemos a continuación.

Ejemplo 8.4 La recurrencia que aparece en el cálculo de los números de Fibonacci recursivamente es:

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

Donde sustituimos las constantes por el valor uno si sólo estamos interesados en la obtención del orden exacto. Expandimos el valor $t(n-1)$ en la ecuación:

$$t(n) = t(n-2) + t(n-3) + 1 + t(n-2) + 1 = 2t(n-2) + t(n-3) + 2$$

Sustituyendo $t(n-2)$ queda:

$$t(n) = 2(t(n-3) + t(n-4) + 1) + t(n-3) + 2 = 3t(n-3) + 2t(n-4) + 4$$

Y expandiendo $t(n-3)$ obtenemos:

$$\begin{aligned} t(n) &= 3t(n-3) + 2t(n-4) + 4 = 3(t(n-4) + t(n-5) + 1) + 2t(n-4) + 4 = \\ &\quad 5t(n-4) + 3t(n-5) + 7 \end{aligned}$$

Vemos que en este caso no es tan fácil obtener una fórmula general que nos permita obtener el valor final hasta llegar al caso base, y es preferible utilizar otra técnica para resolver la ecuación. En general, esto ocurrirá siempre que aparezca más de un término recurrente en t .

8.3. Método de la ecuación característica

Por este método, a una ecuación de recurrencia se le asocia una ecuación polinómica cuyas soluciones proporcionan unas soluciones básicas de la ecuación de recurrencia. La solución general es combinación lineal de las soluciones básicas, y las constantes de la combinación lineal se calculan imponiendo casos base. Por tanto, los valores de los casos base no influyen normalmente en el orden exacto, pero sí en la o-pequeña, donde se tiene en cuenta la constante del término de mayor orden.

No vemos cómo se resuelve cualquier tipo de ecuaciones de recurrencia, sino algunos tipos de los más usuales en el estudio de algoritmos.

8.3.1. Ecuaciones lineales homogéneas

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

$$a_0t(n) + a_1t(n-1) + \dots + a_kt(n-k) = 0$$

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

En un caso con sólo dos términos en t tendríamos la recurrencia:

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

y expandiendo la recurrencia se obtiene $t(n) = a^n$. Por tanto, buscar soluciones de la forma $t(n) = x^n$, con x constante, puede ser una buena idea. De este modo:

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

con lo que, dividiendo por x^{n-k} habrá que resolver la ecuación característica:

$$a_0x^k + a_1x^{k-1} + \dots + a_k = 0$$

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

Ejemplo 8.5 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 $t(n) = c_1 (-1)^n + c_2 4^n$, e imponiendo las condiciones iniciales tenemos:

$$\begin{aligned} t(0) &= c_1 + c_2 = 0 \\ t(1) &= -c_1 + 4c_2 = 1 \end{aligned}$$

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 $p(x) = (x-s)^m q(x)$, 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 $a_0 n x^{n-1} + a_1 (n-1) x^{n-2} + \dots + a_k (n-k) x^{n-k-1}$, y con $x=s$ queda $a_0 n s^{n-1} + a_1 (n-1) s^{n-2} + \dots + a_k (n-k) s^{n-k-1}$, y multiplicando por s tendremos $a_0 n s^n + a_1 (n-1) s^{n-1} + \dots + a_k (n-k) s^{n-k}$, con lo que se ve que $t(n) = n s^n$ es solución de la ecuación de recurrencia.

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

Ejemplo 8.6 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, aplicando el método de Ruffini, como $(x-1)(x-2)^2 = 0$, por lo que la solución general de la recurrencia es $t(n) = c_1 1^n + c_2 2^n + c_3 n 2^n$, e imponiendo las condiciones iniciales tenemos el sistema de ecuaciones:

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

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

Este ejemplo 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.

En el estudio de tiempos de ejecución no pueden aparecer ecuaciones de este tipo porque implicaría que para resolver un problema de tamaño n se resuelven otros de tamaño menor y no se hace nada más (tiempo cero por ser homogénea), lo que no es posible. Sí pueden aparecer ecuaciones homogéneas si lo que queremos es contar algún tipo concreto de instrucciones.

8.3.2. Ecuaciones lineales no homogéneas

Si consideramos el caso:

$$a_0t(n) + a_1t(n-1) + \dots + a_kt(n-k) = b^n p(n)$$

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

$$(a_0x^k + a_1x^{k-1} + \dots + a_k)(x - b)^{d+1} = 0$$

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

Ejemplo 8.7 Si tenemos

$$t(n) - 2t(n-1) = 3^n(n+1) \quad (8.2)$$

multiplicando por 3 tendremos:

$$3t(n) - 6t(n-1) = 3^{n+1}(n+1) \quad (8.3)$$

y sustituyendo en la ecuación 8.2 n por $n+1$ tenemos:

$$t(n+1) - 2t(n) = 3^{n+1}(n+2) \quad (8.4)$$

y restando a 8.4 la ecuación 8.3:

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

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:

$$a_0t(n) + a_1t(n-1) + \dots + a_kt(n-k) = b_1^n p_1(n) + b_2^n p_2(n) + \dots$$

se tiene como ecuación característica:

$$(a_0x^k + a_1x^{k-1} + \dots + a_k)(x - b_1)^{d_1+1}(x - b_2)^{d_2+1}\dots = 0$$

Ejemplo 8.8 Se puede resolver así la recurrencia del cálculo de los números de Fibonacci de forma recursiva. Tenemos $t(n) - t(n-1) - t(n-2) = 1$, por lo que la ecuación característica es $(x^2 - x - 1)(x - 1) = 0$, y las soluciones básicas son $\left(\frac{1+\sqrt{5}}{2}\right)^n$, $\left(\frac{1-\sqrt{5}}{2}\right)^n$ y 1^n . La solución general es combinación de las tres básicas, y para obtener las constantes habría que imponer tres casos base. La constante en el término de mayor orden no puede ser cero, pues el tiempo nos daría constante (término 1^n) o tomaría valores negativos para entradas de tamaño par o impar (término en $\left(\frac{1-\sqrt{5}}{2}\right)^n$). Por tanto, sin calcular las constantes sabemos que $t(n) \in \Theta\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$.

8.3.3. 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 8.9 Si

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

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

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

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 7.1 para quitar esa restricción.

8.3.4. Transformación de la imagen

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

Ejemplo 8.10 Si

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

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

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

y tomando logaritmos:

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

Se hace una transformación de la imagen $v(k) = \log t(2^k)$, con lo que queda:

$$v(k) = \begin{cases} \log 6 & \text{si } k = 0 \\ k + 2v(k-1) & \text{si } k > 0 \end{cases}$$

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 $v(k) = (3 + \log 3)2^k - k - 2 = \log t(2^k)$, y tomando exponentes y simplificando queda que $t(n) = \frac{24^n}{4n}$.

En este caso no se puede aplicar el teorema 7.1 para quitar esa restricción, pues $\frac{24^n}{4n}$ no es 2-armónica. Por lo tanto, sólo conocemos el orden de la función para tamaños del problema potencia de dos.

8.4. Técnica de la inducción constructiva

Como hemos visto, el método de la ecuación característica necesita que las ecuaciones de recurrencia sean lineales. Si la recurrencia es no lineal, se puede intentar aplicar transformación de la imagen, para convertirla en lineal. Otra posible forma de resolver algunas recurrencias no lineales es utilizar la técnica de inducción. De este modo, si suponemos que $t \in \Theta(f)$ ($O(f)$ o $\Omega(f)$), lo podemos demostrar por inducción.

Ejemplo 8.11 Como ejemplo vemos la ecuación de recurrencia:

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

Si tratamos de expandir la recurrencia queda:

$$\begin{aligned} t(n) &= n(n-1)t(n-2) + bn(n + (n-1)^2) = \\ &= n(n-1)(n-2)t(n-3) + bn(n + (n-1)^2 + (n-1)(n-2)^2) \end{aligned}$$

Comprobamos que en el término en t va a aparecer $n!$, y en con b aparecen n términos con multiplicaciones parciales del factorial. Por tanto, es razonable suponer 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 $t(n) \in O(n!)$.

Como directamente no hemos podido demostrarlo, vamos a probar con una acotación algo más restrictiva. 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 a partir de $n = 4$. Por tanto, podemos tomar 4 como caso base y desarrollando queda $u \geq \frac{23}{6}b + a$.

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

8.5. Fórmulas maestras

Ya que en el estudio de algoritmos suelen repetirse ecuaciones de recurrencia de unos ciertos tipos, podemos tener **fórmulas maestras** en las que, dependiendo del valor de unos parámetros, se sepa cuál es el orden exacto de la función.

Algunos ejemplos se muestran a continuación.

Ejemplo 8.12 Con $a, b, b' \in R^+$ y $c \in N$ con $c > 0$ y

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

$$a < 1 \Rightarrow f \in \Theta(1),$$

$$a = 1 \Rightarrow f \in \Theta(n),$$

$$a > 1 \Rightarrow f \in \Theta(a^{\frac{n}{c}}).$$

Ejemplo 8.13 Si $a, b, b' \in R^+$, $c \in N$ con $c > 0$ y $d \in R$ y:

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

$$a < 1 \Rightarrow f \in \Theta(n),$$

$$a = 1 \Rightarrow f \in \Theta(n^2),$$

$$a > 1 \Rightarrow f \in \Theta(a^{\frac{n}{c}}).$$

Ejemplo 8.14 Si $a, b, b' \in R^+$ y $c \in N$ con $c > 1$, y:

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

$$a < 1 \Rightarrow f \in \Theta(1),$$

$$a = 1 \Rightarrow f \in \Theta(\log_c n),$$

$$a > 1 \Rightarrow f \in \Theta(n^{\log_c a}).$$

Ejemplo 8.15 Si $a, b, b' \in R^+$ y $c \in N$ con $c > 1$, $d \in R$, y:

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

$a < c \Rightarrow f \in \Theta(n)$,

$a = c \Rightarrow f \in \Theta(n \log_c n)$,

$a > c \Rightarrow f \in \Theta(n^{\log_c a})$.

Ejemplo 8.16 Si $b, k \in N^+$, y $a, c, p \in R^+$, con $a \geq 1$, $b \geq 2$, y dada la siguiente ecuación:

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

$a < b^k \Rightarrow t \in \Theta(n^k)$,

$a = b^k \Rightarrow t \in \Theta(n^k \log n)$,

$a > b^k \Rightarrow t \in \Theta(n^{\log_b a})$.

En todos los casos vemos que los casos base no influyen en el orden, siempre que las constantes sean mayores que cero.

Utilizando estas fórmulas maestras se pueden obtener los tiempos de algunos de los ejemplos estudiados con anterioridad.

Ejemplo 8.17 En el problema de las Torres de Hanoi la ecuación es $t(n) = 2t(n-1) + 1$, que se corresponde con la fórmula del ejemplo 8.12, con $a = 2$, $c = 1$, $b = 1$ y $n_0 = 1$. Al ser $a > 1$, $t(n) \in \Theta(2^n)$, lo que ya habíamos obtenido expandiendo la recurrencia.

Ejemplo 8.18 En la ordenación por mezcla la ecuación es $t(n) = 2t\left(\frac{n}{2}\right) + n + 1$, que se corresponde con la fórmula del ejemplo 8.15, con $a = 2$, $c = 2$, $b = d = 1$. Al ser $a = c$, $t(n) \in \Theta(n \log n)$.

La demostración de que las fórmulas son ciertas se puede hacer en algunos casos expandiendo la recurrencia y en otros obteniendo la ecuación característica. Vemos un ejemplo y dejamos los otros como problemas.

Ejemplo 8.19 Resolvemos la recurrencia del ejemplo 8.12. Expandimos la recurrencia: $f(n) = b + af(n-c) = b + ab + a^2f(n-2c) = \dots = b(1 + a + a^2 + \dots + a^{k-1}) + a^kf(n-kc)$, con k el menor entero tal que $n - kc \leq n_0$.

Si $a < 1$:

$$f(n) = \frac{b(1-a^k)}{1-a} + a^kb' < \frac{b}{1-a} + b', \text{ con lo que } f \in \Theta(1).$$

Si $a = 1$:

$f(n) = bk + b'$, y al ser $n - kc \geq 0$ es $k \leq \frac{n}{c}$, y $f \in O(n)$. Como es $\frac{n-n_0}{c} < k$ tenemos también $f \in \Omega(n)$, y $f \in \Theta(n)$.

Si $a > 1$:

al ser k el menor entero tal que $n - kc \leq n_0$, tendremos $n - (k-1)c > n_0 \Rightarrow$

$$n - n_0 + c > kc \Rightarrow \frac{n-n_0}{c} + 1 > k \Rightarrow f(n) = b \frac{(a^k-1)}{a-1} + a^kb' \leq b \frac{a^{\frac{n-n_0}{c}+1}-1}{a-1} + a^{\frac{n-n_0}{c}+1}b', \text{ y}$$

se obtiene $f \in \Theta(a^{\frac{n}{c}})$.

Ejercicios resueltos

Ejercicio 8.1 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 (fórmula maestra del ejemplo 8.16).

Solución.

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

Si $a = b^k$, la ecuación sería $(x - a)^2 = 0$, y la solución $t(n) = c_1a^m + c_2ma^m \in \Theta(n^k \log n)$.

Y si $a \neq b^k$, la ecuación característica es $(x - a)(x - b^k) = 0$, con lo que la solución general será de la forma $t(n) = c_1a^m + c_2b^{km}$.

Deshaciendo el cambio queda:

$$t(n) = c_1a^{\log_b n} + c_2b^{k \log_b n} = c_1n^{\log_b a} + c_2n^k$$

Faltaría calcular c_1 y c_2 , pero para esto necesitaríamos algún caso base, pero comparando $\log_b a$ y k vemos que si $\log_b a > k$ ($a > b^k$) el orden es $\Theta(n^{\log_b a})$, si $\log_b a < k$ ($a < b^k$) es $\Theta(n^k)$.

Ejercicio 8.2 Calcular el tiempo de ejecución, en función del valor de n , del siguiente algoritmo.

```

operación algoritmo(x:array[1..MAX,1..MAX] de tipo;n:entero)
var i,j:entero; a,b,c:array[1..MAX,1..MAX] de tipo
    si n>1 entonces
        para i:=1,...,n/2 hacer
            para j:=1,...,n/2 hacer
                a[i,j]:=x[i,j]
                b[i,j]:=x[i,j+n/2]
                c[i,j]:=x[i+n/2,j]
            finpara
        finpara
        algoritmo(a,n/2)
        algoritmo(b,n/2)
        algoritmo(c,n/2)
    finsi

```

Solución.

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

De este modo:

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

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

Desarrollando se llega a:

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

Ejercicios propuestos

Ejercicio 8.3 Demostrar las fórmulas maestras que aparecen como ejemplos 8.13, 8.14 y 8.15.

Ejercicio 8.4 El tiempo de ejecución de un determinado programa se puede expresar con la siguiente ecuación de recurrencia:

$$t(n) = \begin{cases} 2n & \text{si } n \leq 10 \\ 2t(\lfloor \frac{n}{2} \rfloor) + 3t(\lfloor \frac{n}{4} \rfloor) + 2n + 1 & \text{si } n > 10 \end{cases}$$

a) Estudia el tiempo de ejecución del algoritmo, para los valores de n que sean potencia de 2. A partir del tiempo $t(n)$, expresa la complejidad usando las notaciones O , Ω o Θ . (No hace falta calcular el valor de las constantes de $t(n)$, se supondrán todas positivas.)

b) Muestra las condiciones iniciales que deberían aplicarse para calcular las constantes en la fórmula de $t(n)$. Sólo es necesario indicar los valores de n que son las condiciones iniciales y escribir alguna de las ecuaciones que surgen de esas condiciones. No es necesario resolverlas.

c) La afirmación $t(n) \in \Omega(\log n)$ ¿es correcta en este caso? ¿es una buena cota para el orden de complejidad del programa? Justifica las respuestas.

Ejercicio 8.5 Resolver la ecuación recurrente: $t(n) - 2t(n-1) = n + 2^n$, con $n > 0$, y $t(0) = 0$.

Ejercicio 8.6 Resolver la siguiente ecuación recurrente, indicando las condiciones iniciales que se debería aplicar:

$$t(n) = \begin{cases} 1 & \text{si } n < n_0 \\ 8t(n-2) + 2^n + 1 & \text{en otro caso} \end{cases}$$

Cuestiones de autoevaluación

Ejercicio 8.7 Resolver la ecuación recurrente $t_n - 3t_{n-1} - 2 = 0$, con $n \geq 1$, y $t_0 = 0$. Hacerlo expandiendo la recursión, por medio de la ecuación característica, y utilizando una fórmula maestra. Calcular las constantes que aparecen en la solución.

Ejercicio 8.8 Resolver la ecuación recurrente $t_n - 2t_{n-1} = n + 2^n$, con $n \geq 1$, y $t_0 = 0$. Hacerlo expandiendo la recursión y por medio de la ecuación característica. Obtener las constantes que aparecen en la solución.

Ejercicio 8.9 Resolver la ecuación recurrente $t(n) = 2t(n-1) - t(n-2) + n - 1$, con $n \geq 2$, $t(1) = 1$, $t(2) = 2$. Obtener las constantes que aparecen en la solución.

Ejercicio 8.10 Resolver la ecuación recurrente $t(n) = t\left(\frac{n}{2}\right) + t\left(\frac{n}{4}\right) + n$, con $n \geq 2$, $t(1) = 1$, $t(2) = 2$. Obtener las constantes que aparecen en la solución.

Ejercicio 8.11 Resolver la ecuación recurrente

$$t(n) = \begin{cases} 1 & \text{si } n = 1 \\ n^2 t^3\left(\frac{n}{2}\right) & \text{si } n > 1 \end{cases}$$

Obtener las constantes que aparecen en la solución.

Referencias bibliográficas

Las referencias más útiles para este capítulo coinciden con las de los dos capítulos anteriores, y son algunos capítulos de [Cormen90], [Brassard97] y [Baase00]. Se ha seguido principalmente [Brassard97], del que también se han tomado algunos de los ejemplos.

Se puede encontrar más información sobre resolución de ecuaciones de recurrencia en libros de matemáticas, pero los métodos explicados en este capítulo y en los correspondientes de los tres libros referenciados son los más usados en el análisis de algoritmos.

Capítulo 9

Divide y vencerás

En este capítulo y en los siguientes estudiaremos técnicas de diseño de algoritmos. 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. Estas técnicas no se pueden aplicar a todo tipo de problemas. En algunos casos una técnica concreta se puede aplicar pero no da lugar a soluciones óptimas y en otros casos sí.

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

Objetivos del capítulo:

- Comprender la técnica de resolución de problemas por subdivisión en problemas más pequeños.
- Conocer los esquemas básicos de los algoritmos divide y vencerás.
- Aprender a estudiar el coste de los algoritmos divide y vencerás.
- Saber identificar problemas susceptibles de ser resueltos con esta técnica.
- Conocer algunos de los ejemplos típicos de problemas que se resuelven eficientemente por divide y vencerás.

Contenido del capítulo:

9.1.	Método general	61
9.1.1.	Esquema general	61
9.1.2.	Esquema recursivo	61
9.2.	Análisis de tiempos	62
9.2.1.	Caso de dos subproblemas	62
9.2.2.	Caso de a subproblemas	63
9.3.	Búsqueda del máximo y mínimo	64
9.3.1.	Método directo	64
9.3.2.	Con divide y vencerás	65
9.3.3.	Comparación	67
9.4.	Ordenación por mezcla	67
9.4.1.	Algoritmo divide y vencerás	67
9.4.2.	Estudio	69
9.5.	Ordenación rápida	70
9.5.1.	Algoritmo divide y vencerás	70
9.5.2.	Estudio	71
9.5.3.	El problema de selección	72
9.6.	Multiplicación de enteros largos	73
9.6.1.	Multiplicación de Karatsuba y Ofman	74
9.7.	Multiplicación de matrices	76
9.7.1.	Multiplicación de Strassen	77
Ejercicios resueltos		78
Ejercicios propuestos		93
Cuestiones de autoevaluación		94
Referencias bibliográficas		95

9.1. Método general

9.1.1. Esquema general

La técnica consiste en resolver un problema dividiéndolo en una serie de subproblemas del mismo tipo, resolver estos y obtener la solución del problema original a partir de las soluciones de los subproblemas.

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

Algoritmo 9.1 Esquema general de la técnica **divide y vencerás**.

```

operación divide_vencerás(p:problema):solución
    dividir(p,p1,p2,...,pk) // pi son subproblemas de p
    para i:=1,...,k hacer
        si:=resolver(pi)
    finpara
    devolver combinar(s1,s2,...,sk)

```

Para que pueda aplicarse la técnica divide y vencerás necesitamos que se cumpla:

- El problema original debe poder dividirse fácilmente en un conjunto de subproblemas, del mismo tipo que el problema original pero con una resolución más sencilla (menos costosa).
- La solución de un subproblema debe obtenerse independientemente de la de los otros.
- Normalmente los subproblemas deben ser de tamaños parecidos.
- Como mínimo necesitamos que haya dos subproblemas. Si sólo tenemos un subproblema hablamos de **técnicas de reducción** (o simplificación).
- Necesitamos un método (más o menos directo) de resolver los problemas de tamaño pequeño.
- Es necesario tener un método de combinar los resultados de los subproblemas.

9.1.2. Esquema recursivo

Normalmente para resolver los subproblemas se utilizan llamadas recursivas al mismo algoritmo (aunque no necesariamente). De este modo, en el esquema anterior la función resolver consistiría en una llamada a la misma función divide_vencerás, y cuando el problema a resolver es suficientemente pequeño se resuelve de forma directa.

Vemos un esquema recursivo en el que los datos de entrada son globales y se determina el subproblema en que se trabaja por medio de **dos índices**, y en el que se resuelve el problema dividiéndolo de manera recursiva en dos subproblemas hasta llegar a un caso base que se resuelve directamente. Un esquema será:

Algoritmo 9.2 Esquema recursivo de la técnica divide y vencerás cuando se divide el problema recursivamente en dos subproblemas hasta llegar a un caso base.

```

operación divide_vencerás(p,q:índice):solución
var m:índice
    si pequeño(p,q) entonces
        devolver solucion(p,q)
    sino
        m:=dividir(p,q)
        devolver combinar(divide_vencerás(p,m),divide_vencerás(m+1,q))
    finsi
```

donde:

- **pequeño** es una función que determina si el tamaño es suficientemente pequeño para resolver el problema directamente,
- **solucion** 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. \Rightarrow Aquí añadimos condiciones Para saber qué Combinar

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

Ejemplo 9.1 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 el resultado final se obtiene variando los palos origen, destino y pivote, y haciendo un movimiento de un aro.

9.2. Análisis de tiempos

A partir de un esquema algorítmico se puede hacer un estudio general del tiempo de ejecución de los algoritmos que siguen ese esquema.

9.2.1. Caso de dos subproblemas

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

$$t(n) = \begin{cases} g(n) & \text{si } n \text{ es suficientemente pequeño} \\ 2t\left(\frac{n}{2}\right) + f(n) & \text{en otro caso} \end{cases} \quad (9.1)$$

donde $g(n)$ es el tiempo de comprobar si el problema es suficientemente pequeño y de generar la solución y $f(n)$ el de comprobar si es pequeño, dividir el problema y combinar los resultados.

Desarrollando obtenemos:

$$\begin{aligned} t(2^k) &= 2t(2^{k-1}) + f(2^k) = 2(2t(2^{k-2}) + f(2^{k-1})) + f(2^k) = \dots = \\ &2^m g(2^{k-m}) + \sum_{i=0}^{m-1} (2^i f(2^{k-i})) \end{aligned} \quad (9.2)$$

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

Ejemplo 9.2 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 9.2) el tiempo:

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

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 9.3 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 9.2) el tiempo:

$$t(n) = ng(1) + \sum_{i=0}^{k-1} (2^i f(2^{k-i})) = nc + \sum_{i=0}^{k-1} (2^i d 2^{k-i}) = nc + d 2^k k = nc + dn \log n$$

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

Pero si resolvemos el problema directamente el tiempo es $g(n) = cn^2 \in \Theta(n^2)$, 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.

9.2.2. Caso de a subproblemas

Si el problema se divide en a llamadas recursivas de tamaño $\frac{n}{b}$ ($a > 1$ y $b \geq 2$), y la combinación requiere $f(n) = dn^p \in \Theta(n^p)$, entonces la recurrencia es:

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

con lo que tenemos la fórmula maestra del ejemplo 8.16, y

$$t(n) \in \begin{cases} \Theta(n^{\log_b a}) & \text{si } a > b^p \\ \Theta(n^p \log n) & \text{si } a = b^p \\ \Theta(n^p) & \text{si } a < b^p \end{cases}$$

Ejemplo 9.4 Si realizamos 4 llamadas recursivas con subproblemas de tamaño $\frac{n}{2}$ entonces $a = 4$ y $b = 2$, y $f(n) \in \Theta(n)$. Al ser $a > b^p$, $t(n) \in \Theta(n^{\log_2 4}) = \Theta(n^2)$

9.3. 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.

9.3.1. Método directo

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

Algoritmo 9.3 Cálculo del máximo y mínimo, método directo.

```

operación MaxMin(a:array[1..n] de tipo;var max,min:tipo)
var i:entero
      max:=a[1]
      min:=a[1]
      para i:=2,...,n hacer
          si a[i]>max entonces
              max:=a[i]
          sino si a[i]<min entonces
              min:=a[i]
          finsi
      finpara
```

Para estudiar este algoritmo vamos a contar 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 y 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

$$\sum_{i=2}^n \left(\frac{1}{i} + 2 \frac{i-1}{i} \right) = \sum_{i=2}^n \left(2 - \frac{1}{i} \right) = \\ 2(n-1) - \sum_{i=2}^n \frac{1}{i} \approx 2(n-1) - \int_1^n \frac{1}{x} dx = 2(n-1) - \ln n$$

Hemos obtenido el número de comparaciones en el caso más favorable, más desfavorable y promedio y en los tres es 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 **para** (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 $\Theta(\log n)$.

9.3.2. 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 9.4 Cálculo del máximo y mínimo, con divide y vencerás.

```
operación MaxMinDV(a:array[1..n] de tipo;i,j:entero;var max,min:tipo)
    si i<j-1 entonces
        mit:= $\frac{i+j}{2}$ 
        MaxMinDV(a,i,mit,max1,min1)
        MaxMinDV(a,mit+1,j,max2,min2)
        si max1>max2 entonces
```

```

max:=max1
sino
  max:=max2
finsi
si min1<min2
  min:=min1
sino
  min:=min2
finsi
sino si i==j-1 entonces
  si a[i]<a[j] entonces
    max:=a[j]
    min:=a[j]
  sino
    max:=a[i]
    min:=a[j]
finsi
sino
  max:=a[i]
  min:=max
finsi

```

En este caso, para calcular el tiempo de ejecución, podemos suponer que n es una potencia de dos y aplicar después el teorema 7.1 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 $t(n) = c_1n + c_2$, 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, 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 $t(n) = 2n - 2$.

9.3.3. Comparación

Hay que hacer varias consideraciones:

- En el caso de MaxMinDv hemos calculado el número de comparaciones y asignaciones de elementos del 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 punteros) puede ser preferible el método divide y vencerás, pues el coste de las comparaciones puede ser $O\left(\frac{3}{2}n\right)$ y en el método directo $O(2n)$
- Hay que tener en cuenta también que el método divide y vencerás es recursivo, lo que puede producir un mayor tiempo de ejecución por gestionar las llamadas recursivas, y que estas se gestionarán más o menos eficientemente dependiendo del compilador con el que trabajemos.

9.4. Ordenación por mezcla

9.4.1. Algoritmo divide y vencerás

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 número de elementos menor o igual que un cierto valor $base$. En el caso base los datos se ordenan por un método directo, como puede ser inserción, burbuja, sacudida, etc. Si fuera $base = 1$ no se haría nada, pues el subproblema, que consta de un único dato, estaría ordenado. 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.

Un algoritmo ajustándonos al esquema divide y vencerás puede ser:

Algoritmo 9.5 Ordenación por mezcla.

```

operación Mergesort(a:array[1..n] de tipo; p,q:índice; base:entero)
  var m:índice
  // Es pequeño si el tamaño es menor que un caso base
  si q-p<base entonces
    OrdenacionDirecta(a,p,q)
  sino
  // El array original es dividido en dos trozos de tamaño igual
  // (o lo más parecido posible), es decir  $\lfloor \frac{n}{2} \rfloor$  y  $\lceil \frac{n}{2} \rceil$ 
  m:= $\frac{p+q}{2}$ 
  Mergesort(a,p,m,base)
  Mergesort(a,m+1,q,base)
  Merge(a,p,m,q)
  finsi
```

donde Merge es:

```

operación Merge(a:array[1..n] de tipo; p,m,q:índice)
  var b:array[p..q] de tipo; h,i,j,k:índice
  h:=p
  i:=p
  j:=m+1
  mientras h $\leq m$  Y j $\leq q$  hacer
    si a[h] $\leq a[j]
      b[i]:=a[h]
      h:=h+1
    sino
      b[i]:=a[j]
      j:=j+1
    finsi
    i:=i+1
  finmientras
  si h>m entonces
    para k:=j,...,q hacer
      b[i]:=a[k]
      i:=i+1
    finpara
  sino
    para k:=h,...,m hacer
      b[i]:=a[k]
      i:=i+1
    finpara
  finsi
  para k:=p,...,q hacer
    a[k]:=b[k]$ 
```

finpara

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

- **pequeño.** Es la comprobación de que $q - p < base$.
- **solucion.** Es la ordenación de los datos del array entre los índices p y q por un método directo.
- **dividir.** Es $m := \frac{p+q}{2}$.
- **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 utilizando un array auxiliar b del mismo tamaño que a , y usando un parámetro booleano en el procedimiento Mergesort que indique si hay que mezclar de a en b o de b en a .
- Si el método es recursivo, podría emplear un mayor tiempo de ejecución y una mayor ocupación de memoria que si no lo es.

9.4.2. 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 **mientras** 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 **mientras**, tendremos un coste $5(\frac{n}{2} + l)$ del **mientras**, y un coste $3(\frac{n}{2} - l)$ 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:

$$t(n) = \begin{cases} an^2 & \text{si } n \leq base \\ 2t\left(\frac{n}{2}\right) + bn + c & \text{si } n > base \end{cases} \quad (9.3)$$

Donde el término an^2 corresponde a la solución por un método directo, y el $bn + c$ a la mezcla. Si expandimos la recursión hasta llegar al caso base tenemos:

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

Como acaba la recursión cuando $\frac{n}{2^k} = base$, haciendo el cambio queda:

$$t(n, base) = an \cdot base + bn \log n - bn \log base + c\left(\frac{n}{base} - 1\right)$$

Para calcular el valor óptimo del caso base dado un tamaño de problema fijo, n , se deriva la función del tiempo respecto de la variable $base$ y se calcula el valor que anula la derivada:

$$\frac{\partial t}{\partial base} = an - bn \frac{\log e}{base} - \frac{cn}{base^2} = 0$$

de donde se obtiene el valor óptimo del caso base resolviendo una ecuación de segundo grado. Este valor óptimo dependerá del valor de b , que sabemos que está entre 6 y $\frac{13}{2}$, y el de a , que depende del método directo que se use para ordenar los casos base.

9.5. Ordenación rápida

9.5.1. Algoritmo divide y vencerás

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 (**pivote**) y se obtienen los menores o iguales que él almacenándolos en a en las posiciones de la p a la m , y los mayores almacenándolos en las posiciones de la $m + 1$ a la q .

Un algoritmo puede ser:

Algoritmo 9.6 Ordenación rápida.

```
operación Quicksort(a:array[1..n] de tipo;p,q:índice;base:entero)
var j:entero
  si q-p<base entonces
    OrdenacionDirecta(a,p,q)
  sino
    j:=particion(a,p,q)
    Quicksort(a,p,j-1,base)
    Quicksort(a,j+1,q,base)
  finsi
```

En la partición se toma un elemento que actúa de pivote, se obtiene la posición j que le corresponde y se realiza una ordenación relativa ($a[i] \leq a[j] \leq a[k]$, si $p \leq i < j < k \leq q$). Vemos una versión en la que se toma como pivote el primer elemento del subarray que se está tratando ($a[p]$):

```
operación particion(a:array[1..n] de tipo;izq,der:índice):entero
var temp,v:tipo;i,d:índice
  v:=a[izq]
  i:=izq
  d:=der+1
  repetir
    i:=i+1
  hasta i≥der O a[i]>v
  repetir
    d:=d-1
  hasta a[i]≤v
```

```

mientras  $i < d$  hacer
     $temp := a[i]$ 
     $a[i] := a[d]$ 
     $a[d] := temp$ 
    repetir
         $i := i + 1$ 
        hasta  $a[i] > v$ 
    repetir
         $d := d - 1$ 
        hasta  $a[d] \leq v$ 
    finmientras
     $a[izq] := a[d]$ 
     $a[d] := v$ 
    devolver  $d$ 

```

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

- **pequeño.** Es la comprobación $q - p \leq base$.
- **solucion.** Es resolver el subproblema por un método directo.
- **dividir.** Es particionar. En este caso en la división del problema, además de obtener el índice por el que se partitiona, se hace una ordenación relativa.
- **combinar.** No se hace combinación ya que la ordenación relativa que se hace en partition nos asegura que el array quedará ordenado al final.

9.5.2. Estudio

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

$$t(n) = \begin{cases} an^2 & \text{si } n \leq base \\ t(n-1) + bn + c & \text{si } n > base \end{cases}$$

y, expandiendo la recurrencia hasta llegar al caso base, tenemos:

$$\begin{aligned} t(n) &= t(n-2) + b(n+n-1) + 2c = t(n-3) + b(n+n-1+n-2) + 3c = \\ &\quad t(n-k) + b(n+n-1+\dots+n-k+1) + kc \end{aligned}$$

y al ser $n - k = base$ queda:

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

con lo que el orden en el caso más desfavorable es $t_M(n) \in \Theta(n^2)$, y $t(n) \in O(n^2)$.

El caso más favorable se aproxima considerando que el pivote va siempre a la posición central del array, con lo que tenemos una ecuación de recurrencia como la 9.3, y $t_m(n) \in \Theta(n \log n)$, y $t(n) \in \Omega(n \log n)$.

El coste en el caso más desfavorable es mayor que en la ordenación por mezcla, y en el más favorable tiene el mismo orden que la ordenación por mezcla. Sin embargo, la ordenación rápida es mejor en promedio al tener el mismo orden que la ordenación por mezcla pero constantes menores.

Para obtener el tiempo promedio, consideramos por simplicidad el caso base de tamaño 1 y todas las constantes de valor 1. La recurrencia que se tiene es:

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

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.

Consideraremos 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:

$$t(n) = n + 1 + \frac{1}{n} \sum_{i=1}^n (t(i-1) + t(n-i))$$

y multiplicando por n :

$$nt(n) = n(n+1) + 2(t(0) + t(1) + \dots + t(n-1)) \quad (9.4)$$

Sustituyendo en 9.4 n por $n-1$ queda:

$$(n-1)t(n-1) = (n-1)n + 2(t(0) + \dots + t(n-2)) \quad (9.5)$$

Restando a 9.4 la ecuación 9.5, agrupando y dividiendo por $n(n+1)$ queda:

$$\frac{t(n)}{n+1} - \frac{t(n-1)}{n} = \frac{2}{n+1}$$

y desarrollando llegamos a:

$$\frac{t(n)}{n+1} = \frac{2}{n+1} + \frac{2}{n} + \dots + \frac{1}{2} \approx 2 \int_1^{n+1} \frac{1}{x} dx = 2 \ln(n+1)$$

con lo que $t_p(n) \in \Theta(n \log n)$.

9.5.3. El problema de selección

Consideramos un problema que se resuelve con una estructura similar a la de la ordenación rápida pero que no se puede considerar un ejemplo de la técnica divide y vencerás. El problema consiste en, dado un array $a[1..n]$ no ordenado de datos de un cierto tipo con relación de orden (por ejemplo, enteros), y dado un entero s entre 1 y n , encontrar el elemento que se encontraría en la posición s si el array estuviera ordenado.

Si $s = \frac{n}{2}$, tenemos el problema de encontrar la mediana de a , es decir, el valor que es mayor que la mitad de los elementos de a y menor que la otra mitad.

Obviamente el problema se puede resolver ordenando todo el array y tomando el elemento que queda en la posición s , pero hay métodos mejores de resolverlo. Esto requeriría un tiempo $\Theta(n \log n)$. Se puede resolver mejor utilizando el procedimiento particion:

Algoritmo 9.7 Algoritmo de selección.

```

operación Selección(a:array[1..n] de tipo; p,q:índice;s:entero):tipo
var j:índice
    si p=q entonces
        devolver p
    sino
        j:=particion(a,p,q)
        si j<s entonces
            devolver Selección(a,j+1,q)
        sino
            devolver Selección(a,p,j)
        finsi
    finsi
```

Aunque tiene un esquema similar al quicksort, no es un divide y vencerás, sino una **reducción**, ya que el problema se descompone en un único subproblema de tamaño menor.

En el mejor caso, el subproblema es de tamaño $\frac{n}{2}$, con lo que la ecuación de recurrencia es de la forma $t(n) = t\left(\frac{n}{2}\right) + an$, y $t(n) \in \Theta(n)$.

En el peor caso, el subproblema es de tamaño $n - 1$, la ecuación es $t(n) = t(n - 1) + an$, y $t(n) \in \Theta(n^2)$

9.6. Multiplicación de enteros largos

El tipo entero largo Cuando se quiere trabajar con números enteros que exceden la capacidad de los tipos enteros que tenemos, se puede implementar un nuevo tipo **EnteroLargo** como:

```

tipo EnteroLargo=Puntero[nodo]
    nodo=registro
        item:entero
        sig:EnteroLargo
    finregistro
```

Con lo que cada EnteroLargo está compuesto por una serie de enteros de un tamaño determinado (vamos a suponer 1) almacenados en nodos distintos.

Multiplicación directa La multiplicación de dos enteros largos se puede hacer multiplicando todos los nodos de un número por todos los de otro, y sumando los resultados con los desplazamientos adecuados. La complejidad es $\Theta(nm)$ con n y m las longitudes de los dos enteros largos.

Divide y vencerás Se puede hacer la multiplicación con un método divide y vencerás dividiendo los enteros largos x e y de longitud n , descomponiéndolos en dos partes iguales:

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

con lo que:

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

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

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

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

9.6.1. Multiplicación de Karatsuba y Ofman

Utilizando la siguiente fórmula:

$$xy = ac \cdot 10^n + ((a-b)(d-c) + ac + bd) \cdot 10^{\frac{n}{2}} + bd \quad (9.6)$$

queda la recurrencia:

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

Se reduce el número de multiplicaciones de enteros largos de tamaño la mitad, pasando de cuatro a tres multiplicaciones, aunque el valor de la constante b es mayor en este caso al hacerse ocho operaciones de orden lineal, mientras que antes se hacían cinco. Desarrollando tenemos:

$$\begin{aligned}t(n) &= 3 \left(3t\left(\frac{n}{2}\right) + b \frac{n}{2} \right) + bn = \dots = 3^k t(1) + bn \frac{\left(\frac{3}{2}\right)^k - 1}{\frac{3}{2} - 1} = \\&= 3^{\log_2 n} c + 2bn \left(\frac{3^{\log_2 n}}{2^{\log_2 n}} - 1 \right) = n^{\log_3 3} c + 2b(n^{\log_3 3} - n)\end{aligned}$$

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

Tamaño óptimo del caso base Para estimar el tamaño óptimo del caso base consideramos que el método directo tiene un coste $2n^2$, y que las operaciones lineales en el método divide y vencerás tienen un coste $8n$. La ecuación de recurrencia queda:

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

Desarrollando hasta llegar al caso base queda:

$$t(n, \text{base}) = 2n^{\log 3} \text{base}^{2-\log 3} + 16n^{\log 3} \text{base}^{1-\log 3} - 16n$$

y derivando respecto a base y obteniendo el valor que anula la derivada tenemos $\text{base} = \frac{8(\log 3 - 1)}{2 - \log 3} \approx 11$.

Está claro que este valor no coincidirá con el óptimo experimental, pues hemos hecho muchas suposiciones para su estimación, no hemos tenido en cuenta la gestión de la memoria, el trabajo con los índices, y ni siquiera sabemos cómo se va a implementar. A pesar de todo, estimar teóricamente el valor óptimo del caso base tiene interés. Supongamos que tenemos un problema que se puede resolver por un método directo en 100 horas, y somos capaces de diseñar un algoritmo por divide y vencerás que tiene menor orden, y que en el caso óptimo resuelve el problema en 10 horas. No tiene sentido que obtengamos experimentalmente (perdiendo decenas de horas en los experimentos) cuál es el tamaño óptimo del caso base. Una alternativa puede ser dedicar unos pocos minutos a la obtención teórica del caso base óptimo, y resolver el problema con ese tamaño. Posiblemente tardaremos 12 ó 13 horas, que serán algo más del óptimo, pero mucho menos que con el método directo.

Algoritmo 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 9.8 Multiplicación de enteros largos por el método de Karatsuba-Ofman.

```

operación Mult(x,y:EnteroLargo;n,base:entero):EnteroLargo
    si n≤base entonces
        devolver MultBasica(x,y)
    sino
        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,base)))
        asignar(m2,Mult(b,d,n/2,base)))
        asignar(m3,Mult(restar(a,b),restar(d,c),n/2,base)))
        devolver sumar(sumar(Mult10(m1,n),
                                Mult10(sumar(sumar(m1,m2),m3),n/2))),m2)
    finsi
```

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 *base* 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 $n \leq \text{base}$; la función *solucion* 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*; y combinar consiste en llevar a cabo las operaciones indicadas en la fórmula 9.6.

9.7. Multiplicación de matrices

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 $\Theta(n^3)$.

Multiplicación por bloques Para llevar a cabo operaciones matriciales se suele dividir las matrices por bloques. Por ejemplo, si consideramos las matrices divididas en bloques de dimensiones $\frac{n}{3} \times \frac{n}{3}$, tenemos:

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} \quad \begin{pmatrix} B_{11} & B_{12} & B_{13} \\ B_{21} & B_{22} & B_{23} \\ B_{31} & B_{32} & B_{33} \end{pmatrix} \quad = \quad \begin{pmatrix} C_{11} & C_{12} & C_{13} \\ C_{21} & C_{22} & C_{23} \\ C_{31} & C_{32} & C_{33} \end{pmatrix}$$

y cada submatriz C_{ij} se obtiene realizando multiplicaciones directas de matrices de menor tamaño: $C_{ij} = \sum_{k=1}^3 A_{ik}B_{kj}$.

El coste es el mismo que con el método directo, pero experimentalmente se puede obtener un menor tiempo de ejecución con un algoritmo por bloques si se toma el tamaño de bloques de modo que las submatrices quepan en la memoria de menor nivel (la caché), con lo que para las mismas operaciones se producen menos movimientos de datos en la jerarquía de memorias.

Método divide y vencerás Utilizando las ideas de la división por bloques se puede diseñar un algoritmo divide y vencerás. 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 multiplicaciones de matrices y 4 sumas de matrices de dimensión $\frac{n}{2} \times \frac{n}{2}$. En los algoritmos por bloques no se hacen llamadas recursivas para resolver los problemas de menor tamaño, pero en este caso 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:

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

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 $t(n) = (2 + \frac{1}{b}) n^3 - n^2$, con lo que el método divide y vencerás por sí sólo no mejora asintóticamente el tiempo de ejecución de un método directo.

9.7.1. Multiplicación de Strassen

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

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

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

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

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

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

de donde, resolviendo la ecuación de recurrencia e imponiendo los casos base b y $2b$, obtenemos:

$$t(n) = \left(2b^{0,19} + \frac{6}{b^{0,81}} \right) n^{2,81} - 6n^2$$

Derivando la ecuación respecto a b e igualando a cero obtenemos que el tamaño óptimo del caso base es aproximadamente 13.

El método de multiplicación rápida es más rápido que la multiplicación normal asintóticamente, pues su orden es $n^{2,81}$, 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. El punto en que el método divide y vencerás pasa a ser mejor que el directo y el valor óptimo del caso base dependen de factores como el lenguaje o compilador utilizado y la destreza del programador, y en general el valor óptimo del caso base será algo mayor que el obtenido teóricamente.

Ejercicios resueltos

Ejercicio 9.1 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 resultado 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:

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

y el orden total será en promedio:

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

y en el caso más desfavorable:

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

Ejercicio 9.2 Dado un array de N números hacer un programa que, utilizando la técnica divide y vencerás, encuentre una subcadena (cadena con datos en posiciones consecutivas) 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 . 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,...,n hacer //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 hacer //Cálculo del resto de las sumas
    s:=s-x[j-n]+x[j]
    si s>so entonces
        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 $N/2$ y obtener resultados en s_1 y k_1
 resolver con el método anterior con índices $N/2+1$ 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:

operación combinar:

```

    si  $s_1 > s_2$  entonces
         $s := s_1$ 
         $k := k_1$ 
    sino
         $s := s_2$ 
         $k := k_2$ 
    finsi
    directo( $x, N/2-n+2, N/2+n-1, n, s_1, k_1$ )
    si  $s_1 > s$  entonces
         $s := s_1$ 
         $k := k_1$ 
    finsi
finpara
```

Ejercicio 9.3 Siguiendo el esquema divide y vencerás se programa un algoritmo de ordenación:

operación ordenar(i, d :índices):

```

    si  $d - i < 10$  entonces
        ordenarbasico( $i, d$ )
    sino
         $m := (d+i) \text{ div } 2$ 
        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 $\Theta(n^2)$, y el mergesort un $\Theta(n \log n)$, por lo que, tendremos que $t(n) = 2t\left(\frac{n}{2}\right) + bn + a$ si $n \geq 10$, y si $n < 10$ será $t(n) \in \Theta(n^2)$ o $t(n) \in \Theta(n \log n)$ dependiendo de que ordenarbasico sea un método de la burbuja o un

mergesort, respectivamente.

Expandiendo la recurrencia tendremos:

$$\begin{aligned} t(n) &= 2t\left(\frac{n}{2}\right) + bn + a = 2\left(2t\left(\frac{n}{2^2}\right) + b\frac{n}{2} + a\right) + bn + a = 2^2t\left(\frac{n}{2^2}\right) + b2n + a(1+2) = \\ &= \dots = 2^kt\left(\frac{n}{2^k}\right) + bkn + a(1+2+\dots+2^{k-1}) = 2^kt\left(\frac{n}{2^k}\right) + bkn + a(2^k - 1) \end{aligned}$$

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:

$$\begin{aligned} t(n) &= 2^{\log n - \log 10}t(10) + b(\log n - \log 10)n + a\left(\frac{n}{10} - 1\right) = \\ &\frac{n}{10}t(10) + bn\log n - bn\log 10 + a\left(\frac{n}{10} - 1\right) \in \Theta(n\log n) \end{aligned}$$

independientemente de que el caso base se resuelva por la burbuja o mergesort, ya que $t(10)$ será una constante 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 7.1.

Ejercicio 9.4 Suponiendo que tenemos un algoritmo que multiplica matrices triangulares ($a_{ij} = 0$ si $i > j$, es triangular superior) 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:

operación multiplicar(*A,B*:matriz; *d*:entero; var *C*:matriz):

si pequeño(*d*) **entonces**

 mult_completa(*A,B,d,C*)

sino

 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:

$$t(n) = 2^2t\left(\frac{n}{2^2}\right) + \frac{3}{2}n^{2,81}(1 + 2^{-1,81}) + \frac{1}{4}n^2(1 + 2^{-1})$$

A continuación

$$t(n) = 2^3t\left(\frac{n}{2^3}\right) + \frac{3}{2}n^{2,81}\left(1 + 2^{-1,81} + (2^{-1,81})^2\right) + \frac{1}{4}n^2\left(1 + 2^{-1} + (2^{-1})^2\right)$$

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:

$$t(n) = n + \frac{3}{2}n^{2,81}\frac{\frac{n^{1,81}-1}{2^{1,81}-1}}{\frac{n^{1,81}-1}{2^{1,81}-1}} + \frac{1}{4}n^2\frac{\frac{n-1}{2}}{\frac{n-1}{2}} = n + \frac{3}{2^{1,81}-1}n^{2,81} - \frac{3}{2^{1,81}-1}n + \frac{1}{2}n^2 - \frac{1}{2}n \in \Theta(n^{2,81})$$

Ejercicio 9.5 Tenemos el esquema del algoritmo quicksort:

operación quicksort(*izq,der*:índice):

si *izq*<*der* **entonces**

p:=mediana(*izq,der*)

 particionar(*p,izq,der,div*)

 quicksort(*izq,div*)

 quicksort(*div+1,der*)

finsi

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 partitionar es el procedimiento típico de partitionar 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 partitionar 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:

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

Expandiendo la recurrencia tenemos:

$$\begin{aligned} t(n) &= 21n + b + 2t\left(\frac{n}{2}\right) = 21n + b + 2\left(21\frac{n}{2} + b + 2t\left(\frac{n}{2^2}\right)\right) = \\ &= 21n + b + 2\left(21\frac{n}{2} + b + 2^2t\left(\frac{n}{2^2}\right)\right) = 21n + b + 2\left(21\frac{n}{2} + b + 2^2\left(21\frac{n}{2^3} + b + 2t\left(\frac{n}{2^3}\right)\right)\right) = \\ &\dots = 21nk + b(1+2+\dots+2^{k-1}) + 2^kt\left(\frac{n}{2^k}\right) = \\ &= 21nk + b(2^k - 1) + 2^ka = 21n \log n + b(n-1) + an \in \Theta(n \log n | n = 2^k) \end{aligned}$$

Para quitar la condición hay que aplicar el teorema 7.1. 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 $t(n) = 21n + b + t(\lceil \frac{n}{2} \rceil) + t(\lfloor \frac{n}{2} \rfloor)$, y $t(n+1) = 21n + 21 + b + t(\lceil \frac{n+1}{2} \rceil) + t(\lfloor \frac{n+1}{2} \rfloor)$, y por la hipótesis de inducción es $t(\lceil \frac{n+1}{2} \rceil) \geq t(\lceil \frac{n}{2} \rceil)$ y $t(\lfloor \frac{n+1}{2} \rfloor) \geq t(\lfloor \frac{n}{2} \rfloor)$, 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.

Ejercicio 9.6 Suponemos el esquema del método divide y vencerás recursivo:

operación resolverDV(p :problema; i, d :índice):

```

si pequeño( $p, i, d$ ) entonces
    resolverbasico( $p, i, d$ )
sino
     $m := (i+d)/2$ 
    resolverDV( $p, i, m-1$ )
    resolverDV( $p, m, d$ )
    combinar( $p, i, m, d$ )
finsi

```

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

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 contraejemplo, 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 $t(n) = 2^k g(1) + \sum_{i=0}^{k-1} 2^i f\left(\frac{n}{2^i}\right)$.

Si tomamos $g(n) = \ln n$ y $f(n) = 1$, tendremos $t(n) \in O(2^k) = O(n)$, 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 $O(f) = O(n^a)$ con a mayor que uno, tendremos:

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

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

Ejercicio 9.7 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 encontramos 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.

operación metododirecto(*t:array[1..n,1..n]* **de** 0..1;*n:entero*;**var** *f,c,tma:entero*):

tma:=0 //Tamaño máximo actual

fila:=1 //Fila que se está evaluando

mientras *fila+tma≤n* **hacer**

 //Si sobrepasa el tamaño de la tabla no podemos mejorar la solución actual

columna:=1

mientras *columna+tma≤n* **hacer**

si *t[fila,columna]=1* **entonces**

tam:=cuadrado(fila,columna)

si *tam>tma* **entonces**

f:=fila

c:=columna

tma:=tam

finsi

finsi

```

        columna++
finmientras
fila++
finmientras
operación cuadrado(fila,columna:entero):
    para i:=0,...,min(n-fila,n-columna) hacer
        para j:=0,...,i hacer
            si t[fila+i,columna+j]=0 entonces
                devolver i
            finsi
        finpara
        para j:=0,...,i-1 hacer .
            si t[fila+j,columna+i]=0 entonces
                devolver i
            finsi
        finpara
    finpara
    devolver 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 éstas. 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.

```

operación divideyvenceras(t:array[1..n,1..n] de 0..1;n:entero;var f,c,tma:entero):
    si n<=base entonces
        metododirecto(t,n,f,c,tma)
    sino

```

```

divideyvenceras(&t[1,1],n/2,f,c,tma)
divideyvenceras(&t[1,n/2+1],n/2,f2,c2,tma2)
divideyvenceras(&t[n/2+1,1],n/2,f3,c3,tma3)
divideyvenceras(&t[n/2+1,n/2+1],n/2,f4,c4,tma4)
si tma2>tma entonces
  tma:=tma2
  f:=f2
  c:=c2
finsi
si tma3>tma entonces
  tma:=tma3
  f:=f3
  c:=c3
finsi
si tma4>tma entonces
  tma:=tma4
  f:=f4
  c:=c4
finsi
fronteras(t,n,f,c,tma)
finsi

```

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

operación fronteras(*t:array[1..n,1..n] de 0..1;n:entero;var f,c,tma:entero*):

```

fila:=1
mientras fila+tma≤n hacer
  columna:=n/2
  mientras columna≥1 Y t[fila,columna]=1 hacer
    columna--
  finmientras
  columna++
  mientras columna+tma≤n Y columna≤n/2 hacer
    tam:=cuadrado(fila,columna)
    si tam>tma entonces
      f:=fila
      c:=columna
      tma:=tam
    finsi
    columna++
  finmientras
  fila++
finmientras
columna:=1
mientras columna+tma≤n hacer
  fila:=n/2
  mientras fila≥1 Y t[fila,columna]=1 hacer

```

```

fila--
finmientras
fila++
mientras fila+tma≤n Y fila≤n/2
tam:=cuadrado(fila,columna)
si tam>tma entonces
f:=fila
c:=columna
tma:=tam
finsi
fila++
finmientras
columna++
finmientras

```

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: $\sum_{i=\frac{n}{2}}^n i^2(2n - 2i + 1) \in O(n^4)$. Por lo tanto, la ecuación de recurrencia es de la forma $t(n) = 4t\left(\frac{n}{2}\right) + O(n^4)$, cuya solución nos da un $O(n^4)$. 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.

Ejercicio 9.8 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:

```

operación D_V(a:array de datos;i,j:índices)
si i<j entonces
    m1:=(j-i)/3+i
    m2:=((j-i)/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)
finsi

```

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:

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

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 $t_k = c_12^k + c_2k2^k$, y $t(n) = c_1n + c_2n \log_2 n$. Planteando el sistema:

$$\begin{aligned} t(1) &= a &= c_1 \\ t(2) &= 2a + 2b &= 2c_1 + 2c_2 \end{aligned}$$

obtenemos $c_1 = a$ y $c_2 = b$, y $t(n) \in o(bn \log_2 n)$, 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:

$$t(n) = \begin{cases} a & \text{si } n = 1 \\ 3t\left(\frac{n}{3}\right) + f(n) & \text{si } n > 1 \end{cases}$$

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 $t_k = c_13^k + c_2k3^k$, y $t(n) = c_1n + c_2n \log_3 n$. Planteando el sistema:

$$\begin{aligned} t(1) &= a &= c_1 \\ t(3) &= 3a + 2b &= 3c_1 + 3c_2 \end{aligned}$$

obtenemos $c_1 = a$ y $c_2 = b$, y $t(n) \in o(bn \log_3 n)$, condicionado a que n sea potencia de tres. La restricción se puede quitar al igual que en la ordenación por mezcla normal pues $n \log_3 n$ 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 resueltos 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 auxiliar 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 $(a + c)n \log_2 n$.

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 $\frac{5(a+c)}{3}n \log_3 n$.

El método de ordenación por mezcla dividiendo en tres subproblemas sería mejor que el método normal si $\frac{5(a+c)}{3}n \log_3 n < (a + c)n \log_2 n$, lo que ocurre si $5 \log_3 2 < 3$, 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.

operación combinar(*a:array[1..n] de datos; i1,i2,i3,i4:entero*):

```

i:=i1
j:=i2+1
k:=i3+1
l:=i1
mientras i≤i2 Y j≤i3 Y k≤i4 hacer
    si a[i]<a[j] entonces
        si a[i]<a[k] entonces
            b[l]:=a[i]
            i:=i+1
        sino
            b[l]:=a[k]
            k:=k+1
        finsi
    sino
        si a[j]<a[k] entonces
            b[l]:=a[j]
            j:=j+1
        sino

```

```

    b[l]:=a[k]
    k:=k+1
  finsi
  finsi
  l:=l+1
finmientras
si i>i2 entonces
  mezclar(a,j,i3,k,i4,l,b)
sino si j>i3 entonces
  mezclar(a,i,i2,k,i4,l,b)
sino
  mezclar(a,i,i2,j,i3,l,b)
finsi

```

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:

```

operación mezclar(a:array[1..n] de datos ;i1,f1,i2,f2,ib:entero;
  var b:array[1..n] de datos):
  mientras i1≤f1 Y i2≤f2 hacer
    si a[i1]<a[i2] entonces
      b[ib]:= a[i1]
      i1:=i1+1
    sino
      b[ib]:= a[i2]
      i2:=i2+1
    finsi
    ib:=ib+1
  finmientras
  si i1>f1 entonces
    para i:=i2,...,f2 hacer
      b[ib]:= a[i]
      ib:=ib+1
    finpara
  sino
    para i:=i1,...,f1 hacer
      b[ib]:= a[i]
      ib:=ib+1
    finpara
  finsi

```

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 compiendo 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 $(a + 2c)n \log_3 n$.

La ordenación dividiendo en tres subproblemas será mejor que la normal dividiendo en dos si $(a + 2c)n \log_3 n < (a + c)n \log_2 n$ (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.

Ejercicio 9.9 Consideramos el problema de obtener, de n números, los $\frac{n}{10}$ menores ordenados.

- Hacer un programa según el esquema divide y vencerás para resolver el problema.
- Estudiar el coste de dicho programa.
- 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.

```

operación ordenar_primeros(izq,der:índices):
    si der-izq<n/10 entonces
        quicksort(izq,der)
    sino
        m:=(izq+der)/2
        ordenar_primeros(izq,m)
        ordenar_primeros(m+1,der)
        mezclar_primeros(izq,m+1,der,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:

```

operación mezcla_primeros(izq,cen,der,tot:entero):
    i:=izq
    j:=cen
    k:=1
    mientras i<cen Y j≤der Y k≤tot hacer
        si a[i]<a[j] entonces
            b[k]:=a[i]
            i++
        fin si
```

```

sino
   $b[k]:=a[j]$ 
   $j++$ 
finsi
   $k++$ 
finmientras
si  $k < tot$  entonces
  si  $i < cen$  entonces
    mientras  $i < cen$  Y  $k \leq tot$  hacer
       $b[k]:=a[i]$ 
       $k++$ 
       $i++$ 
    finmientras
  sino
    mientras  $j \leq der$  Y  $k \leq tot$  hacer
       $b[k]:=a[j]$ 
       $k++$ 
       $j++$ 
    finmientras
  finsi
finsi
 $i:=izq$ 
 $k:=1$ 
mientras  $k \leq tot$  Y  $i \leq der$  hacer
   $a[i]:=b[k]$ 
   $i++$ 
   $k++$ 
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 $t(n) = 2^k t\left(\frac{n}{2^k}\right) + \frac{n}{10} k$, y como el caso base es cuando $\frac{n}{2^k} = \frac{n}{10}$, tenemos $k = \log 10$, y el tiempo será $n \log n - n \log 10 + \frac{n}{10} \log 10$.
- 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 $t(n) = 2^k t\left(\frac{n}{2^k}\right) + 100(2^k - 1)$ 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 $O(n \log n)$.

Ejercicios propuestos

Ejercicio 9.10 Realizar un algoritmo que, utilizando la técnica de divide y vencerás, encuentre el mayor y segundo mayor elementos de un array. Estudiar el número de comparaciones y asignaciones de elementos del tipo que hay en el array.

Ejercicio 9.11 Dado el problema de obtener el tercer mayor elemento de un array con elementos de un cierto tipo T :

- a) Hacer un programa para resolverlo directamente (sin divide y vencerás).
- b) Calcular Ω y O del número de comparaciones de elementos del tipo T en el algoritmo del apartado a).
- c) Calcular Ω y O del número de asignaciones de elementos del tipo T en el algoritmo del apartado a).
- d) Calcular Θ y σ del número promedio de comparaciones de elementos del tipo T en el algoritmo del apartado a).
- e) Calcular Θ y σ del número promedio de asignaciones de elementos del tipo T en el algoritmo del apartado a).
- f) Hacer un programa para resolverlo por divide y vencerás.
- g) Calcular Θ y σ del número promedio de comparaciones de elementos del tipo T en el algoritmo del apartado f).
- h) Calcular Θ y σ del número promedio de asignaciones de elementos del tipo T en el algoritmo del apartado f).

Ejercicio 9.12 Programar por divide y vencerás, con un esquema recursivo, 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.

Ejercicio 9.13 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 ($DATOS=der-izq+1$).

Ejercicio 9.14 Dar un algoritmo para multiplicar una matriz triangular superior por una completa (considerar matrices cuadradas), y estudiar su tiempo de ejecución.

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.

Ejercicio 9.15 Diseñar un algoritmo de divide y vencerás para multiplicar dos matrices triangulares superiores, suponiendo matrices cuadradas. Estudiar el tiempo de ejecución del algoritmo, comparándolo con el tiempo con un algoritmo clásico.

Ejercicio 9.16 Un algoritmo divide y vencerás divide un problema de tamaño n en 2 subproblemas de tamaño $n - 1$. El algoritmo aplica una solución directa cuando $n = 2$, que tarda un tiempo de 3 ms. Si el tiempo de realizar la combinación es $f(n) = 1,2n$ ms, calcular el tiempo de ejecución exacto del algoritmo.

Cuestiones de autoevaluación

Ejercicio 9.17 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?

Ejercicio 9.18 Dadas dos funciones f y g como en el método divide y vencerás, ecuación 9.1, ¿se cumple que si $O(f) < O(g)$ entonces es mejor aplicar divide y vencerás que el método directo? Encontrar un contraejemplo.

Ejercicio 9.19 Un programa que utiliza la técnica divide y vencerás, divide un problema de tamaño n en a subproblemas de tamaño $\frac{n}{b}$. El tiempo $g(n)$ de la resolución directa (caso base) se considerará constante. El tiempo $f(n)$ de combinar los resultados es $O(n^p)$. Para simplificar consideraremos que $f(n) = dn^p$. Obtener el orden total de ejecución del algoritmo, $t(n)$, en función de los valores a y b .

Comprobar que los resultados anteriores son aplicables para cualquier valor de $g(n)$ y para valores de $f(n)$ de la forma $f(n) = a_p n^p + a_{p-1} n^{p-1} + \dots + a_1 n + a_0$.

Ejercicio 9.20 Supongamos el problema de calcular todos los caminos mínimos en un grafo no dirigido. En general, ¿es posible descomponer fácilmente el problema en subproblemas, de forma que sea posible aplicar divide y vencerás? ¿En qué casos sí sería posible aplicar esta descomposición de forma sencilla? ¿Qué mejora se obtendría en tal caso?

Ejercicio 9.21 Un programa para ordenar cadenas utiliza el método de ordenación por mezcla. La resolución para problemas de tamaño pequeño tarda un tiempo de $g(n) = 2n^2$, mientras que la mezcla requiere $f(n) = 10n$. Calcular cuál debería ser el tamaño del caso base para optimizar el tiempo de ejecución total.

Ejercicio 9.22 En un algoritmo divide y vencerás, en lugar de aplicar el caso base cuando n es menor que cierto valor, se aplica la recurrencia siempre un número r de veces. Calcular cuál será el tiempo de ejecución, suponiendo que un problema de tamaño n es dividido en a problemas de tamaño $\frac{n}{b}$, siendo el tiempo del caso base $g(n) = n^q$, y el tiempo de la mezcla $f(n) = n^p$.

Referencias bibliográficas

Dado que el problema de ordenación es uno de los más básicos, en la mayoría de los libros de algoritmos o de estructuras de datos aparecen los algoritmos de ordenación que se estudian en este capítulo [Aho74], [Wirth80], [Horowitz82], [Baase83], [Wirth87], [Aho88], [Kruse89], [Manber89], [Cormen90], [Gonnet91], [Weis95], [Baase00]. Normalmente estos algoritmos aparecen dentro de un capítulo dedicado a problemas de ordenación junto con algunos métodos directos. El libro clásico de Knuth [Knuth87] está dedicado a algoritmos de ordenación y búsqueda y, si bien no incluye los métodos tal como los hemos visto aquí, sí contiene un estudio detallado de los algoritmos, por lo que puede ser interesante para aquellos que quieran aprender más sobre el estudio de algoritmos.

También la multiplicación de matrices por el método de Strassen (o Winograd, que es una cierta implementación de este método) y otros problemas aritméticos aparecen en muchos libros, normalmente más avanzados que los anteriores [Aho74], [Baase83], [Manber89], [Cormen90], [Gonnet91], [Weis95], [Baase00].

La técnica divide y vencerás se estudia en profundidad en pocas referencias. En algunas aparece dentro de un capítulo dedicado a técnicas de diseño [Aho88], [Galve93], [Weis95], [Baase00], o se incluyen algunas ideas de divide y vencerás dentro de un capítulo dedicado a algún tipo de problemas [Kruse89], [Manber89], [Cormen90], [Harel97].

Los libros que abordan las técnicas de diseño tal como aquí hacemos con un enfoque paradigmático (guiados por la técnica de diseño y no por el tipo de problemas que se resuelven) son escasos. Los más utilizados son los de Brassard y Bratley [Brassard90], [Brassard97]. Estos libros tienen capítulos distintos para las distintas técnicas. En particular tienen uno dedicado a la técnica divide y vencerás, donde se incluyen los ejemplos vistos en este capítulo, algunos ejemplos más y una serie de problemas propuestos.

Capítulo 10

Algoritmos voraces

Los algoritmos voraces (ávidos, avariciosos, de avance rápido o greedy en inglés) se utilizan normalmente en problemas de optimización. En algunos casos obtienen soluciones óptimas, pero otras veces sólo alcanzan una aproximación a una solución óptima, con lo que es necesario la posterior realización de una búsqueda local para mejorar la solución obtenida. Trabajan formando una solución paso a paso, tomando en cada paso la decisión que en ese momento parece mejor, de modo que son algoritmos rápidos pero que normalmente no aseguran obtener la solución óptima.

Objetivos del capítulo:

- Comprender la técnica de resolución de problemas por avance rápido y los distintos casos que se pueden presentar en la resolución de problemas por esta técnica: obtención de la solución óptima, de una solución no óptima, o no obtención de la solución.
- Aprender a estudiar el coste de los algoritmos de avance rápido, e indentificar cómo el coste de las operaciones básicas influye en el coste final.
- Saber identificar problemas susceptibles de ser resueltos con esta técnica.
- Conocer algunos de los ejemplos típicos de problemas que se resuelven de manera óptima por avance rápido.
- Conocer algunos ejemplos típicos de problemas donde se usa alguna heurística para obtener una solución por avance rápido.
- Comprender la importancia de la heurística en la resolución de algunos tipos de problemas.
- Entender el método de avance rápido como técnica para obtener una solución inicial a partir de la cuál se puede realizar una búsqueda local, y conocer algunos ejemplos que se pueden resolver de ese modo.

Contenido del capítulo:

10.1. Método general	99
10.1.1. Esquema general	99
10.1.2. Distintas posibilidades en la solución de problemas	100
10.1.3. Análisis del tiempo de ejecución	100
10.2. Ejemplos de avance rápido con técnicas heurísticas	101
10.2.1. Problema del cambio de monedas	101
10.2.2. Paseo del caballo	103
10.2.3. Camino mínimo en grafos multietapa	103
10.3. Problema de la mochila no 0/1	105
10.3.1. Planteamiento	105
10.3.2. Esquema de solución	106
10.3.3. Solución óptima	107
10.4. Secuenciamiento de trabajos a plazos	108
10.4.1. Planteamiento	108
10.4.2. Solución óptima	109
10.4.3. Algoritmo	110
10.5. Heurísticas voraces	112
10.5.1. Problema del viajante	112
10.5.2. Coloración de grafos	114
Ejercicios resueltos	115
Ejercicios propuestos	117
Cuestiones de autoevaluación	118
Referencias bibliográficas	119

10.1. Método general

10.1.1. Esquema general

El método de **avance rápido** se utiliza normalmente para aproximar una solución óptima en problemas de optimización que se resuelven con una serie de decisiones que pueden ser tomadas en orden. Se trata de obtener una solución que debe cumplir unas ciertas restricciones y minimizar o maximizar una función de coste. Una solución se obtiene tomando, de entre una serie de entradas, unas determinadas en algún orden, 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 restricciones 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 10.1 Esquema de avance rápido.

```
operación voraz (C:conjunto_candidatos;var S:conjunto_solución)
    S :=  $\emptyset$ 
    mientras C  $\neq \emptyset$  Y NO solucion(S) hacer
        x:=seleccionar(C)
        C:=C-{x}
        si factible(S  $\cup$  {x}) entonces
            S:=S  $\cup$  {x}
        finsi
    finmientras
    si no solucion(S) entonces
        devolver "No se puede encontrar una solución"
    finsi
```

donde

- Partimos de una solución vacía. *S* contendrá la solución.
- En cada paso se escoge el siguiente elemento para añadir a la solución, entre los candidatos, utilizando para esto la función seleccionar.
- Si el elemento se puede incluir en la solución parcial sin violar las restricciones, lo que se comprueba con la función factible, se añade a la solución.
- Una vez tomada esta decisión no se podrá deshacer.
- El algoritmo acabará cuando el conjunto de elementos seleccionados constituya una solución, o no queden elementos por seleccionar, en cuyo caso se acaba sin encontrar solución.

10.1.2. Distintas posibilidades en la solución de problemas

Cuando se intenta resolver un problema por la técnica de avance rápido se nos pueden presentar varios casos:

- Puede que sepamos que el algoritmo encuentra siempre una solución óptima. Este es el caso ideal, pero la mayoría de las veces no es así. En estos casos será necesario demostrar que el algoritmo encuentra siempre una solución óptima, pues el esquema no nos lo asegura, al tomar cada decisión con información local.
- Lo más usual es que el avance rápido nos dé una solución cercana a una óptima, pero que no sepamos, cuando tenemos la solución, si ésta es óptima o no. Para mejorar la solución obtenida se puede realizar una búsqueda local, que puede mejorar o no la solución que ya tenemos.
- Hay veces que no nos da solución. En este caso puede que sepamos que el problema tiene solución aunque el avance rápido no la encuentre, o puede que ni siquiera sepamos si el problema tiene solución, y en este caso no encontrar solución no nos asegura que no la haya.
- El método se puede aplicar a veces a problemas que no son de optimización. En este caso se trata de encontrar una solución que cumple unas restricciones, si la encontramos tenemos éxito y si no la encontramos no. Se podría considerar un problema de optimización con valor 0 ó 1, un cero indicaría que no se encuentra solución y un 1 que sí, y lo que pretendemos es maximizar ese valor.

En las secciones siguientes veremos ejemplos de cada uno de estos casos.

10.1.3. Análisis del tiempo de ejecución

El orden de complejidad depende de:

- El número de candidatos, ya que el esquema consiste en un bucle por el que se pasa como mucho tantas veces como candidatos tengamos para incluir en la solución. Si antes de evaluar todos los candidatos se encuentra una solución, no hace falta tratarlos todos.
- El número de elementos de una solución. Aunque por el bucle se puede pasar tantas veces como elementos haya en el conjunto de posibles candidatos, se pasará al menos tantas veces como elementos tenga la solución, en caso de que el algoritmo encuentre una.
- La función selección, que es de vital importancia, pues debe seleccionar un elemento prometedor para formar parte de la solución, y esta decisión se toma con información local. Cuanto más local sea la información que se utilice, más rápida será la función, pero la selección será peor. Se mantendrá un compromiso entre el tiempo de ejecución de la función y la bondad de la elección realizada.

- Las funciones solucion y factible son normalmente menos costosas que la selección, y además no influyen en el éxito de la decisión tomada.

Si llamamos N al número de candidatos y M al número de elementos de una solución, se repite el bucle como máximo N veces y como mínimo M , y en cada pasada:

- Se comprueba si el valor actual es solución. Le llamamos coste $f(M)$. Es normalmente $O(1)$ o $O(M)$.
- Se selecciona un elemento entre los candidatos. Coste $g(N)$. En el mejor caso puede estar entre $O(1)$ y $O(N)$, pero entonces la información que se utiliza es muy local para tomar la decisión. Si se utiliza algún método para obtener información menos local (por ejemplo una búsqueda en un árbol de soluciones hasta un cierto nivel) aumenta el coste pero también las posibilidades de éxito.
- Se comprueba si a la solución parcial se le puede añadir el elemento seleccionado sin violar las restricciones del problema. El trabajo es similar al de la función solucion, pero en este caso con una solución parcial. Al coste le llamamos $h(M)$, aunque el tamaño será el de la solución parcial.
- La unión de un nuevo elemento a la solución puede requerir otras operaciones de cálculo dependiendo de la representación de los conjuntos. A su coste le llamamos $j(N, M)$.

Con esas funciones, el tiempo de ejecución tendrá la forma:

$$t(N, M) \in O(N * (f(M) + g(N) + h(M)) + M * j(N, M))$$

En la práctica los algoritmos voraces suelen ser bastante rápidos, encontrándose dentro de órdenes de complejidad polinomiales. El inconveniente es que la solución que se obtiene (si se obtiene) no tiene por qué ser óptima.

10.2. Ejemplos de avance rápido con técnicas heurísticas

Ya hemos dicho que en la función selección se utiliza alguna heurística que nos haga pensar que la decisión que se toma en cada paso es apropiada en el sentido de que nos acerca a una solución óptima. Vemos en esta sección varios ejemplos típicos de problemas que se puede intentar resolver con la técnica de avance rápido pero que no podemos estar seguros de que nos lleve a una solución óptima, y en algunos casos ni siquiera de que nos lleve a solución.

10.2.1. Problema del cambio de monedas

Tenemos monedas de unas cantidades determinadas c_1, c_2, \dots , con una cantidad ilimitada de cada tipo, y queremos devolver una cantidad C de manera que el número de

monedas devueltas sea mínimo. En este caso la restricción viene dada por $\sum_{i=1}^n s_i c_i = C$ y la función de coste que hay que minimizar es $\sum_{i=1}^n s_i$, siendo s_i el número de monedas de valor c_i que se devuelven.

La función de selección puede consistir en tomar la moneda de mayor valor de las que no se han gastado todavía, y tomar la mayor cantidad posible de manera que no excede de lo que queda por devolver. Para esto se realiza un **preprocesamiento** antes de entrar en el bucle, consistente en ordenar las monedas de mayor a menor valor. Una vez preprocesada la entrada, la selección se realiza con un coste constante, pues consiste en tomar la siguiente moneda y dividir la cantidad por devolver por el valor de la moneda.

La inclusión será siempre posible, aunque en algunos casos se almacenará el valor cero que indica que no se ha tomado ninguna moneda de ese tipo.

Identificamos las distintas partes en el esquema:

- Los candidatos iniciales son todos los tipos de monedas disponibles.
- Una solución es un conjunto de monedas que suman la cantidad C . Tendrá la forma $s = (s_1, s_2, \dots, s_n)$, donde s_i indica la cantidad de monedas de valor c_i que se devuelven.
- La función **solucion** comprueba si se cumple la restricción: $\sum_{i=1}^n s_i c_i = C$.
- La función **seleccion** toma $s_{paso} = \lfloor \frac{C - \sum_{i=1}^{paso-1} s_i c_i}{c_{paso}} \rfloor$. Por lo tanto se añade a la solución al mismo tiempo que se selecciona.
- La función **factible** será siempre verdad.

El algoritmo será:

Algoritmo 10.2 Solución del problema de devolución de monedas.

operación Devolver_cambio(C :entero; c :array[1..n] de entero;

var s :array[1..n] de entero)

```

act:=0
para i:=1,...,n hacer
    s[i]:=0
finpara
mientras act≠C hacer
    j:=el mayor elemento de c tal que  $c[j] \leq C - act$ 
    si j=0 entonces
        devolver "No existe solución"
    finsi
    s[j]:=(C-act) div c[j]
    act:=act+c[j]*s[j]
finmientras

```

Este método no lleva siempre a soluciones óptimas, como se ve tomando unas cantidades ilimitadas de monedas de valor 25, 12 y 1, y $C = 36$. Se daría una moneda de 25 y once de 1, mientras que la solución óptima es dar 3 monedas de 12.

Tampoco se puede asegurar que el problema tenga solución, por ejemplo si tenemos monedas de cantidad 25 y 5 y $C = 36$. Si tenemos monedas de valor 1 sí podemos asegurar que el problema va a tener solución y que el algoritmo va a encontrar una solución aunque no sabremos si ésta es óptima.

Y puede que el problema tenga solución y el algoritmo no la encuentre. Esto pasa con valores 25 y 10 y cantidad a devolver 30.

10.2.2. Paseo del caballo

Otro ejemplo es el algoritmo de Warnsdorff para resolver el problema de recorrer con un caballo de ajedrez, situado en una casilla inicial, todas las casillas de un damero sin pasar dos veces por una misma casilla.

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. Podemos guardar la solución como un array de pares o como un array de movimientos, codificando los ocho movimientos que puede hacer un caballo.

No tenemos un problema de optimización pero se puede ir tomando decisiones de manera que intentemos, de los ocho posibles movimientos desde una casilla, ir a una configuración que parezca prometedora en el sentido de que sea factible llegar a una solución desde ella. 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. De esta manera se intenta elegir antes casillas por las que hay más problemas para pasar, y dejar para el final casillas a las que es más fácil acceder.

Se puede ver que dependiendo del tamaño del tablero el algoritmo nos lleva a una solución o no encuentra solución, como se ve en los siguientes tableros:

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.

10.2.3. Camino mínimo en grafos multietapa

Como vimos en el capítulo 5, un grafo multietapa es un grafo (V, A) (V es el conjunto de nodos y A el de aristas) en el que los nodos estarán agrupados en niveles, N_0, N_1, \dots, N_l , y sólo hay aristas entre nodos de niveles consecutivos: si $v_k \in N_i$ y $v_m \in N_j$, y $j \neq i + 1$, no existe la arista (v_k, v_m) . Cada arista del grafo tiene asociado un peso.

En la figura 10.1 se muestra un grafo multietapa con cinco niveles.

El problema consiste en, dado un nodo origen del primer nivel ($v_o \in N_0$) y un nodo destino en el último nivel ($v_d \in N_l$), encontrar el camino de longitud mínima que nos lleva del nodo origen al destino, siendo la longitud del camino la suma de los pesos de las aristas que lo componen.

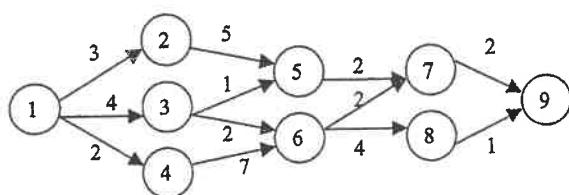


Figura 10.1: Grafo multietapa.

Para encontrar una solución aproximada del problema por avance rápido lo que hay que hacer es:

- Decidir cómo va a ser una solución, de manera que se pueda construir dando una sucesión de pasos. En nuestro caso, como cada camino tiene que tener una arista que lleve de N_i a N_{i+1} , el número de aristas será l , y estas aristas se pueden tomar en orden: en el paso i , con $i = 1, 2, \dots, l$ la que une N_{i-1} con N_i .
- Determinar el tipo de bucle en el algoritmo. Puede ser un bucle **para** de 1 a l , aunque puede que lleguemos a un nodo desde el que no hay arista al nivel siguiente, ya que no está asegurado que existan todas las aristas posibles entre vértices de dos niveles consecutivos. En este caso podría decidirse acabar, con lo que nos saldríamos del bucle, y se devuelve que no se ha encontrado camino, aunque es posible que lo haya.
- La función **seleccion** puede ser tomar, de las aristas que salen del nodo en el que estamos, la de menor peso. Vemos que el conjunto de candidatos varía de un paso a otro. Se toma una decisión que parece adecuada en ese momento, pero que puede hacer que vayamos a un nodo desde el que el camino hasta el destino sea de peso mayor que el que obtendríamos habiendo ido a otro nodo. Esto hace que podamos no obtener la solución óptima. El coste de esta función es proporcional al número de nodos del nivel siguiente al que estamos; si consideramos que tenemos n nodos y $l + 1$ niveles y que aproximadamente tendremos la misma cantidad de nodos por nivel, el coste será $\frac{n}{l+1}$. El único caso distinto se da en el nivel $l - 1$, pues en este caso tenemos una única arista que incluir.
- La función **factible** no es necesaria, pues siempre que se selecciona una arista se puede incluir en la solución.
- Lo aparece función **solucion** de manera explícita, pues el valor seleccionado se incluye al apuntar la arista en la siguiente posición del array **solución**.

Analizamos cómo funciona con el ejemplo del grafo 10.1:

Ejemplo 10.1 La solución constará de 4 aristas.

La primera arista que se toma es la (1,4), de peso 2.

Del nodo 4 sólo se puede tomar una arista, por lo que se incluye la arista (4,6) en la solución.

Del nodo 6 se toma la arista (6,7), de peso 2.

Y del nodo 7 se va al 9 por la única arista posible.

La solución tiene longitud 13 y está formada por las aristas (1,4), (4,6), (6,7) y (7,9). Esta solución no es óptima.

El tiempo de ejecución depende del número de pasos y del coste de cada paso. El número de pasos es l . Los $l - 1$ primeros tienen coste $\frac{n}{l+1}$ aproximadamente, y el último tiene coste 1, por lo que el tiempo será de la forma $\frac{n}{l+1}(l - 1) + 1 \in \Theta(n)$. Tenemos un algoritmo muy rápido pero que no nos lleva a solución óptima. Se puede obtener un algoritmo algo más lento pero que posiblemente obtenga una solución mejor. Una posibilidad es modificar la función selección de manera que desde cada nodo en el nivel N_i se analicen los caminos que llevan a N_{i+2} (caminos con dos aristas), y nos quedemos con la arista de N_i a N_{i+1} del camino mínimo de los analizados. Como del nodo en el que estamos a N_{i+1} hay unas $\frac{n}{l+1}$ aristas, y de cada nodo de N_{i+1} a N_{i+2} la misma cantidad, el coste de la selección será $(\frac{n}{l+1})^2$, salvo en el último paso, que seguirá siendo 1. Por tanto el coste total es $(\frac{n}{l+1})^2(l - 1) + 1 \in \Theta(\frac{n^2}{l})$. Se ha aumentado el coste pero puede que la solución que obtengamos sea mejor, como vemos con el mismo grafo anterior.

Ejemplo 10.2 Se analizan los caminos del nodo 1 al 5 y al 6. El camino más corto es el (1,3,5), con lo que tomamos la arista (1,3).

Se analizan los caminos del 3 al 7 y 8, y se toma la arista (3,5).

Se analizan los caminos del 5 al destino, y tomamos la arista (5,7).

Finalmente se toma la única arista posible, la (7,9).

La solución tiene longitud 9 y está formada por las aristas (1,3), (3,5), (5,7) y (7,9).

Se ha mejorado la solución anterior aumentando el tiempo de ejecución del algoritmo, aunque no podemos asegurar que siempre un aumento en el tiempo nos mejore la solución. La mejora en la solución se obtiene porque cada decisión se toma utilizando información menos local. Se puede aumentar la cantidad de información que se usa para tomar la decisión, analizando en cada paso caminos de longitud tres, cuatro, etc., hasta llegar a caminos de longitud l , en cuyo caso tendríamos la solución óptima pero no sería un método de avance rápido pues se analizan todas las posibles soluciones, lo que corresponde a una técnica de backtracking o ramificación y poda.

10.3. Problema de la mochila no 0/1

10.3.1. Planteamiento

Tenemos n objetos y una mochila. La mochila tiene capacidad M (admite un peso M) y cada objeto i tiene un peso p_i y un beneficio asociado b_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 $\sum_{i=1}^n b_i x_i$, sujeto a $\sum_{i=1}^n p_i x_i \leq M$, $0 \leq x_i \leq 1$ (además b_i y p_i deben ser positivos). Una solución será una ordenación de x_i que satisfaga las restricciones, y será óptima si maximiza el beneficio.

10.3.2. Esquema de solución

Para diseñar una solución por avance rápido hay que tomar una serie de decisiones:

- El conjunto de candidatos está formado por los n objetos.
- Tendremos una solución cuando se haya llenado la mochila o se hayan analizado todos los posibles candidatos.
- La función `seleccion` elegirá para meter en la mochila el objeto más “prometedor” según algún criterio posiblemente heurístico.
- La función `factible` será siempre cierta, pues se pueden meter trozos de objetos.
- La función `objetivo` viene dada por la fórmula $\sum_{i=1}^n b_i x_i$.

Con estas consideraciones un esquema del algoritmo puede ser:

Algoritmo 10.3 Esquema de algoritmo voraz para el problema de la mochila no 0/1.

operación Mochilano0/1(M :entero; b, p :array[1.. n] de entero;

var s :array[1.. n] de real)

```

 $s \leftarrow 0$ 
pesoact:=0
paso:=1
mientras paso< $n$  Y pesoact< $M$  hacer
     $i :=$  seleccionar(objetos)
    si pesoact+ $p[i] \leq M$  entonces
         $s[i] := 1$ 
        pesoact:=pesoact+ $p[i]$ 
    sino
         $s[i] := (M - pesoact) / p[i]$ 
        pesoact:= $M$ 
    finsi
    paso++
finmientras

```

En el algoritmo anterior aparece notación que utilizaremos en algunos de los algoritmos sucesivos. El operador \leftarrow lo usaremos para indicar que el array a la izquierda del operador inicializa todos sus campos con el valor que aparece a la derecha. El operador $++$ (el $--$) indicará que la variable a su izquierda incrementa (decrementa) su valor en uno.

Tenemos varias opciones para seleccionar los objetos. Consideramos las dos siguientes:

- Seleccionar por beneficios. Se eligen los objetos de mayor a menor beneficio.
- Seleccionar por peso. Se eligen los objetos de menor a mayor peso.

Se puede realizar un preprocesamiento para ordenar los elementos de mayor a menor beneficio o de menor a mayor peso, y después de esto la función de selección lo único que hace es tomar los objetos en orden. El coste de la función de selección sería constante, y el del algoritmo proporcional al número de objetos que se meten en la mochila, y a esto hay que sumar el preprocesamiento que tiene coste $\Theta(n \log n)$.

En el siguiente ejemplo vemos que no se obtiene la solución óptima con ninguno de los dos métodos.

Ejemplo 10.3 Consideramos $n = 3$, $M = 20$, $b = (25, 24, 15)$ y $p = (18, 15, 10)$.

Seleccionando por beneficio la solución será $(1, \frac{2}{15}, 0)$ que tiene beneficio $25 + 2\frac{24}{15} = 28,2$.

Y seleccionando por peso: $(0, \frac{2}{3}, 1)$, que tiene beneficio $15 + \frac{2}{3}24 = 31$.

Ninguno de los dos da la solución óptima, pues la solución $(0, 1, \frac{1}{2})$ tiene beneficio $24 + 15\frac{1}{2} = 31,5$.

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.

10.3.3. Solución óptima

Con el siguiente teorema se obtiene un método de avance rápido con el que se encuentra una solución óptima.

Teorema 10.1 Si se ordenan los objetos con $\frac{b_1}{p_1} \geq \frac{b_2}{p_2} \geq \dots$ 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 $X = (x_1, x_2, \dots, x_n)$ 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 $Y = (y_1, y_2, \dots, y_n)$, 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 si no 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}, \dots, y_n para no exceder la capacidad de la mochila. Llamamos a esta nueva solución $Z = (z_1, z_2, \dots, z_n)$, siendo la cantidad de peso que añadimos igual a la que quitamos, por lo que:

$$(z_k - y_k) p_k = \sum_{i=k+1}^n (y_i - z_i) p_i$$

y además

$$\begin{aligned} \sum_{i=1}^n b_i z_i &= \sum_{i=1}^n b_i y_i + (z_k - y_k) b_k - \sum_{i=k+1}^n (y_i - z_i) b_i = \\ &= \sum_{i=1}^n b_i y_i + (z_k - y_k) b_k \frac{p_k}{p_k} - \sum_{i=k+1}^n (y_i - z_i) b_i \frac{p_i}{p_i} \geq \\ &\geq \sum_{i=1}^n b_i y_i + \left((z_k - y_k) p_k - \sum_{i=k+1}^n (y_i - z_i) p_i \right) \frac{b_k}{p_k} = \sum_{i=1}^n b_i y_i \end{aligned}$$

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 10.4 Aplicando al ejemplo 10.3 el método dado en el teorema 10.1 se ordenan los objetos de la forma $b = (24, 15, 25)$, $p = (15, 10, 18)$, y la solución óptima es $p = (1, \frac{1}{2}, 0)$, con beneficio $24 + 15\frac{1}{2} = 31.5$.

10.4. Secuenciamiento de trabajos a plazos

10.4.1. 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 b_i de manera que si la ejecución del trabajo i se empieza antes del plazo d_i se tiene un beneficio b_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 $\sum_{i \in I} b_i$, 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 10.5 Vemos cómo trabaja este método con los siguientes valores: $n = 4$, $b = (100, 10, 15, 27)$ y $d = (2, 1, 2, 1)$.

Habrá 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 $O(n!)$.

10.4.2. Solución óptima

El conjunto solución estará formado por los trabajos en el orden en que se ejecutarían, pudiendo formar parte de la solución sólo trabajos que empiezan a ejecutarse antes de su plazo. Tendremos la solución cuando hayamos tratado todos los candidatos. Una posibilidad para la función de selección es elegir de los candidatos restantes el que tenga mayor valor de beneficio, pero de esta forma no se obtiene la solución óptima, como se ve en el ejemplo anterior, donde se obtendría la solución (1,3), que no es óptima.

Sin embargo, se puede utilizar un método voraz para obtener una solución óptima. El método consiste en añadir en cada paso el trabajo de mayor b_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. Demostramos que este método voraz da lugar a una solución óptima.

Lema 10.1 Si J es un conjunto de k tareas y $\sigma = (s_1, s_2, \dots, s_k)$ es una permutación de J tal que $d_{s_1} \leq d_{s_2} \leq \dots \leq d_{s_k}$, entonces J es realizable $\Leftrightarrow \sigma$ 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 $\exists \rho = (r_1, \dots, r_k)$ con $d_{r_i} \geq i \forall i = 1, 2, \dots, k$. 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 obteniendo 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 .

Repetiendo el proceso llegamos en un número finito de pasos a que σ es realizable.

Teorema 10.2 El método voraz consistente en incluir en cada paso el trabajo con mayor b_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 10.1, 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 $b_a > b_b$ se podría mejorar J ,
- si $b_a < b_b$ el algoritmo habría tomado b y no a ,
- por tanto, es $b_a = b_b$.

y como en cada posición los beneficios son iguales, I es también óptima.

10.4.3. Algoritmo

Basándonos en las observaciones anteriores podemos diseñar un algoritmo voraz óptimo. El problema principal para obtener un algoritmo eficiente es que cada vez que se selecciona un nuevo trabajo para incluirlo en la solución parcial hay que comprobar si existe alguna ordenación que respete los plazos de ejecución. Comprobar todas las posibilidades tiene un coste $n!$, por lo que parece lógico no comprobar todas las posibles ordenaciones, sino hacer una ordenación de forma que las tareas con plazos más tempranos se ejecuten antes.

La estructura del algoritmo voraz será:

- Empezar con una secuencia vacía, con todas las tareas como candidatas.
- Realizar un preprocessamiento consistente en ordenar los candidatos de mayor a menor beneficio.
- En cada paso, hasta que se acaben los candidatos, repetir:
 - Elegir entre los candidatos restantes el que tenga mayor beneficio.
 - Comprobar si es posible añadir la tarea elegida a la solución actual. Para ello se introduce la nueva tarea en la posición adecuada, manteniendo las tareas ordenadas de menor a mayor plazo. Si el nuevo orden (s_1, s_2, \dots, s_k) cumple $d_{s_i} \leq i$, para todo i entre 1 y k , entonces el nuevo candidato es factible. Si no, rechazar el candidato.

En el algoritmo suponemos que los trabajos ya están ordenados de mayor a menor beneficio.

Algoritmo 10.4 Algoritmo voraz para el problema de asignación de trabajos con plazos.

```

operación Trabajos(d:array[1..n] de entero; var s:array[1..n] de 0..n)
    d[0]:=0
    s[0]:=0
    //actúan de centinelas
    k:=1
    s[1]:=1
    //incluye el primer trabajo
    para i:=2,...,n hacer
        r:=k
        //k indica la cantidad de trabajos incluidos,
        //y r tendrá la posición donde se ejecuta el nuevo trabajo
        mientras d[s[r]]>d[i] Y d[s[r]]≠r hacer
            r:=r-1
        finmientras
        //se busca mantener la solución ordenada crecientemente por d
        //se consigue moviendo a la derecha los trabajos siempre que queden dentro de su plazo
        si d[s[r]]≤d[i] Y d[i]>r entonces
            //si están en orden y se puede incluir el trabajo i
            para l:=k,...,r+1 hacer
                s[l+1]:=s[l]
            finpara
            s[r+1]:=i
            //ha puesto i en su lugar de ejecución
            k:=k+1
        finsi
    finpara
```

Para el cálculo del tiempo hay que tener en cuenta que se ordenan los trabajos por el beneficio, lo que tiene coste $\Theta(n \log n)$.

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 **mientras** y del **si** 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 **mientras** ni del **si**, 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 **mientras** se ejecuta $i - 1$ veces y el **para** interno al **si** se ejecuta con índices de $i - 1$ a 1, con lo que el tiempo de ejecución será:

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

por lo que el algoritmo tiene un orden $O(n^2)$.

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 $O(n^2)$. 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 poner el trabajo i en el lugar j conlleva un coste $i - j$, por lo que el tiempo promedio es:

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

donde no hemos considerado las constantes por cuestión de claridad.

10.5. Heurísticas voraces

Existen muchos problemas para los cuales no se conocen algoritmos que puedan encontrar la solución de forma eficiente. Se llaman **problemas NP-completos**. La solución exacta de estos problemas puede requerir un orden factorial o exponencial, ya que se produce el **problema de la explosión combinatoria**. Para resolver estos problemas se hace necesario utilizar **algoritmos heurísticos**. Un algoritmo heurístico (o heurística) puede producir una buena solución (puede que la óptima) pero también puede que no produzca ninguna solución o dar una no muy buena. La solución se basa normalmente en un conocimiento intuitivo del programador sobre el problema.

La técnica de avance rápido se puede usar para obtener soluciones aproximadas a algunos de estos problemas. Se intentará que la solución obtenida esté próxima a una óptima. Esta solución inicial se puede intentar mejorar haciendo 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 con tiempo de ejecución reducido. La función de selección en estos casos se determina de modo heurístico.

10.5.1. 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 sin pasar dos veces por una misma ciudad y minimizando la longitud del camino recorrido.

Es un problema NP, pero necesitamos una solución eficiente. Se trata de un problema de optimización, donde la solución está formada por un grupo de elementos en cierto orden. Usaremos un algoritmo voraz para obtener una solución cercana a la óptima en un tiempo reducido. Tenemos al menos dos posibilidades:

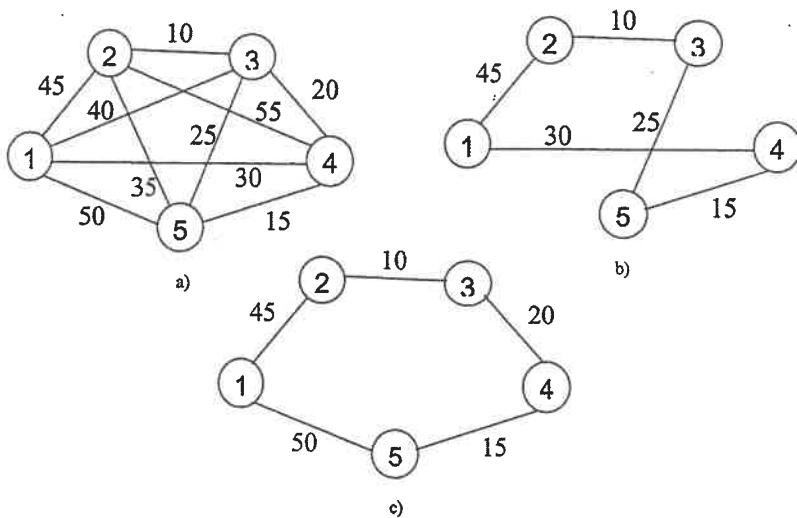


Figura 10.2: Problema del viajante: a) mapa de ciudades, b) y c) dos soluciones distintas.

- Los nodos son los candidatos. Se empieza en un nodo cualquiera y en cada paso nos movemos al nodo no visitado más próximo al último nodo seleccionado. Se evitará cerrar ciclos salvo en el último movimiento.
- Las aristas son los candidatos. Se incluirá en cada paso la arista de menor peso pero garantizando que se pasa dos veces por la misma ciudad y que con la última arista se cierra el ciclo.

Ejemplo 10.6 Como ejemplo, sea el grafo de la figura 10.2.a), que representa el mapa de carreteras que unen las cinco ciudades a recorrer por el viajante.

Utilizando la primera de las dos heurísticas, una solución será el orden en que se visitan los nodos. Inicialmente se selecciona un nodo cualquiera. Después se va seleccionando, de los nodos candidatos, el más próximo al último de la secuencia actual. Se acaba cuando se han incluido los n nodos. Empezando en el nodo 1 se obtiene la solución (1,4,5,3,2) (figura 10.2.b), de coste $30+15+25+10+45=125$, y empezando en el nodo 5 la (5,4,3,2,1) (figura 10.2.c), de coste $15+20+10+45+50=140$.

Con la segunda heurística una solución será un conjunto de aristas que formen un ciclo hamiltoniano, sin importar el orden. Se empezará con un grafo sin aristas, y en los sucesivos pasos se seleccionará la arista candidata de menor coste. Una arista se puede añadir a la solución actual (función factible) si no se forma un ciclo (excepto para la última arista añadida) y si los nodos unidos no tienen grado mayor que 2. En el ejemplo se obtiene la solución (2,3),(4,5),(3,4),(1,2),(1,5) (figura 10.2.c), de coste $10+15+20+45+50=140$. 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 cierra un ciclo, lo que se puede hacer llevando conjuntos de nodos conectados y el coste de la comprobación de que dos nodos no están en el mismo conjunto depende de la implementación de conjuntos que se utilice.

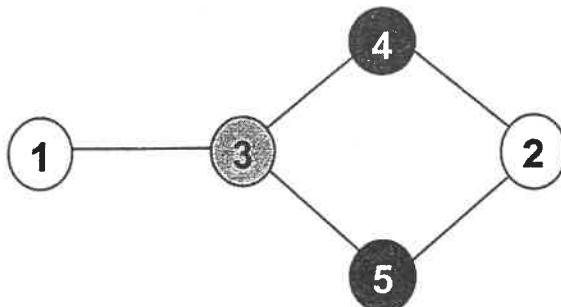


Figura 10.3: Problema de coloración de grafos.

Ninguna de las dos funciones de selección garantiza una solución óptima. Sin embargo, normalmente ambos dan soluciones próximas a la óptima. Se puede intentar obtener soluciones mejores de distintas maneras: se pueden buscar heurísticas mejores; repetir la misma heurística desde varios puntos (en la primera heurística se pueden considerar varios nodos origen). También se puede realizar una búsqueda local a partir de la solución obtenida por avance rápido, por ejemplo, realizando 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.

10.5.2. Coloración de grafos

El problema consiste en, dado un grafo no dirigido, realizar una coloración utilizando el número mínimo de colores. Una coloración es una asignación de un color a cada nodo, de forma que dos nodos unidos con un arco tengan siempre distinto color.

Una solución será un conjunto de pares de la forma $(\text{nodo}, \text{color})$ cumpliendo que para todo (n_i, c_i) y (n_j, c_j) , si (n_i, n_j) es una arista del grafo, entonces $c_i \neq c_j$.

Se puede usar una heurística voraz para obtener una solución. Se empieza con un color c_1 , y todos los nodos sin colorear. Para cada uno de los nodos que no tienen asignado un color, se prueba si es posible asignarles el color actual. Si quedan nodos sin colorear, se escoge otro color y se asigna igual que antes a los nodos no coloreados. La estructura básica del esquema voraz se repite varias veces, una por cada color, hasta que todos los nodos estén coloreados. La función de selección consiste en tomar cualquier nodo no coloreado. La función factible establece que se puede asignar un color al candidato actual si ninguno de sus adyacentes tiene ese mismo color. El algoritmo no garantiza la solución óptima, como vemos en el ejemplo.

Ejemplo 10.7 Consideramos el grafo de la figura 10.3. Con el primer color se colorean los nodos 1 y 2, con el segundo color sólo se colorea el nodo 3, y con el tercer color se colorean los nodos 4 y 5. Se usan tres colores, que no es la solución óptima, que consta de dos colores, uno para los nodos 1, 4 y 5, y otro para los 2 y 3.

Ejercicios resueltos

Ejercicio 10.1 Consideramos el problema de la mochila modificado, donde tenemos una mochila de capacidad M , n objetos con beneficios $b = (b_1, b_2, \dots, b_n)$ y pesos $p = (p_1, p_2, \dots, p_n)$, 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←0
ordenar objetos de mayor a menor b/p
para i:=1,...,n hacer
    si p[i]≤cap entonces
        s[i]:=1
        cap:=cap-p[i]
        ben:=ben+b[i]
    sino si p[i]/2≤cap entonces
        s[i]:=1/2
        cap:=cap-p[i]/2
        ben:=ben+b[i]/2
    finsi
finpara

```

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 $n \log n$, 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.

Ejercicio 10.2 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 la sucesión de números con suma máxima, de entre todas las sucesiones de números, uno de cada fila de la tabla, y con los números en filas consecutivas

adyacentes en vertical o diagonal.

- 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.
- Estudiar el tiempo de ejecución del programa.
- 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:

```
para  $i:=1,\dots,n$  hacer
    seleccionar( $s,i$ )
finpara
```

donde en seleccionar se toman los elementos según hemos explicado:

```
operación seleccionar(var  $s:\text{array}[1..n]$  de entero;  $i:\text{entero}$ ):
```

$$\begin{aligned} & \text{max}:=0 // O \text{ un valor mínimo} \\ & \text{si } i=1 \text{ entonces } // \text{Primera fila} \\ & \quad \text{para } j:=1,\dots,m \text{ hacer} \\ & \quad \quad \text{si } t[1,j]>\text{max} \text{ entonces } // t \text{ representa la tabla} \\ & \quad \quad \quad s[1]:=j \\ & \quad \quad \quad \text{max}:=t[1,j] \\ & \quad \text{finsi} \\ & \quad \text{finpara} \\ & \text{sino} \\ & \quad \text{si } t[i,s[i-1]]>\text{max} \text{ entonces } // \text{Elemento en la misma columna} \\ & \quad \quad s[i]:=s[i-1] \\ & \quad \quad \text{max}:=t[i,s[i]] \\ & \quad \text{finsi} \\ & \quad \text{si } s[i-1]>1 \text{ entonces } // \text{No en la primera columna} \\ & \quad \quad \text{si } t[i,s[i-1]-1]>\text{max} \text{ entonces } // \text{Elemento en la columna anterior} \\ & \quad \quad \quad s[i]:=s[i-1]-1 \\ & \quad \quad \quad \text{max}:=t[i,s[i]] \\ & \quad \text{finsi} \\ & \quad \text{finsi} \\ & \quad \text{si } s[i-1]<m \text{ entonces } // \text{No en la última columna} \\ & \quad \quad \text{si } t[i,s[i-1]+1]>\text{max} \text{ entonces } // \text{Elemento en la columna siguiente} \\ & \quad \quad \quad s[i]:=s[i-1]+1 \\ & \quad \quad \quad \text{max}:=t[i,s[i]] \\ & \quad \text{finsi} \\ & \quad \text{finsi} \end{aligned}$$

finsi

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 un camino con dos pasos a partir de ella. En el ejemplo, desde 7 (columna 1) se puede acceder a 3 (columna 1) o 1 (columna 2), 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, ya que, para cada elemento, es el máximo de él mismo sumado con las posiciones de la siguiente fila adyacentes en diagonal y vertical.

Ejercicios propuestos

Ejercicio 10.3 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.

Ejercicio 10.4 Resolver por avance rápido el problema del camino mínimo en grafo multietapa generalizado, consistente en un grafo con los nodos agrupados por niveles y en el que sólo puede haber aristas de niveles N_i a N_j con $i < j$.

Ejercicio 10.5 Construir un algoritmo voraz para resolver el problema de la minimización del tiempo en el sistema, para un conjunto de n tareas, cada una de las cuales tarda un tiempo predefinido t_i , suponiendo que disponemos de m procesadores para ejecutar las tareas (en lugar de uno solo). ¿Cuál es el orden de complejidad del algoritmo? Mostrar la ejecución para un ejemplo con $m = 4$, $n = 9$ y $t = (7, 1, 6, 2, 3, 4, 2, 5, 7)$.

Ejercicio 10.6 En un sistema monetario disponemos de monedas con valores $1, C, C^2, C^3, \dots, C^n$, siendo $C > 1$. Demostrar que, en este caso, el algoritmo voraz para el cambio de monedas obtiene siempre la solución óptima.

Cuestiones de autoevaluación

Ejercicio 10.7 Contestar si son ciertas o no cada una de las siguientes cuestiones:

- a) Los algoritmos de avance rápido tienen un coste lineal.
- b) Los algoritmos de avance rápido tienen un coste polinómico.
- c) Los algoritmos de avance rápido, cuando se aplican a un problema de optimización, no encuentran la solución óptima.
- d) Los algoritmos de avance rápido encuentran siempre solución, aunque puede que no sea la óptima.
- e) Los algoritmos de avance rápido no se aplican a problemas que no tienen solución.
- f) El método de selección para el problema del caballo conduce siempre a solución cuando la casilla origen es la superior izquierda.
- g) El método de avance rápido sólo se aplica a problemas de optimización.

Ejercicio 10.8 En el problema de la coloración de un grafo, obtener otras funciones de selección y comparar los resultados con los obtenidos con la función de selección utilizada en el ejemplo.

Ejercicio 10.9 Indicar cómo se puede aumentar el volumen de información utilizado en el problema del viajante para hacer la selección de los nodos que se incluyen en el camino solución.

Ejercicio 10.10 Para los algoritmos de Dijkstra, Prim y Kruskal, indicar (de la manera más precisa posible) cómo están definidas las funciones: `solucion`, `seleccionar`, `factible`, `insertar` y `objetivo`. Previamente definir los tipos de datos usados para el conjunto de candidatos y para la solución, en cada uno de estos algoritmos.

Ejercicio 10.11 ¿Se puede considerar la ordenación por selección como un algoritmo voraz? En tal caso, describir cómo serían las funciones del esquema básico.

Referencias bibliográficas

La técnica de avance rápido se utiliza en la solución de algunos problemas sobre grafos (algoritmos de Dijkstra y de Kruskal, etc), por lo que se pueden encontrar algunas de las ideas y ejemplos de avance rápido en capítulos dedicados a grafos [Aho74], [Horowitz82], [Baase83], [Aho88], [Brassard90], [Cormen90], [Brassard97], [Baase00]. Algunas veces se tratan los dos temas juntos en un mismo capítulo [Baase00].

En [Cormen90] se dedica un capítulo a técnicas heurísticas, y se ven algunos ejemplos de avance rápido.

En algunas referencias se trata la técnica someramente dentro de un apartado dedicado a técnicas de diseño [Aho88], [Weis95], [Harel97].

Son pocos los libros que dedican un capítulo completo a este método. Algunos de ellos son textos docentes [Campos95] y [Gonzalo98], que incluye problemas resueltos. Los más utilizados en la preparación de este capítulo han sido [Brassard90], [Brassard97].

Capítulo 11

Programación dinámica

La programación dinámica es una técnica que se suele utilizar para resolver problemas de optimización en los que la solución se obtiene con una serie de decisiones. En esto coincide con el avance rápido pero, a diferencia de este, en la programación dinámica se obtiene la solución óptima, y no una aproximación de ella.

La programación dinámica es una técnica ascendente, en la que se parte de la solución de los problemas de menor tamaño para obtener la del tamaño mayor, que se pretende resolver. El divide y vencerás, por contra, es una técnica descendente, en la que se parte del problema inicial y se va dividiendo en subproblemas.

Objetivos del capítulo:

- Comprender la técnica de resolución de problemas por programación dinámica, e identificar las diferencias con las dos técnicas estudiadas: divide y vencerás y avance rápido.
- Entender la ventaja de la programación dinámica con respecto a otras técnicas en cuanto a tiempo de ejecución, al evitar la repetición de cálculos que aparece en métodos recursivos.
- Saber identificar problemas que cumplen el principio de optimalidad, que es necesario para poder aplicar esta técnica.
- Aprender a estudiar el coste de los algoritmos por programación dinámica, tanto el tiempo de ejecución como la ocupación de memoria.
- Conocer algunos de los ejemplos típicos de problemas que se resuelven por programación dinámica.

Contenido del capítulo:

11.1. Método general	123
11.1.1. Ideas generales	123
11.1.2. Programación dinámica en problemas que no son de optimización	125
11.1.3. Análisis de tiempo de ejecución y ocupación de memoria	126
11.2. Problema del grafo multietapa	127
11.3. Problema del cambio de monedas	128
11.3.1. Planteamiento de la solución por programación dinámica	128
11.3.2. Algoritmo	129
11.4. Problema de la mochila 0/1	130
11.4.1. Planteamiento de solución por programación dinámica	130
11.4.2. Algoritmo	131
11.5. Multiplicación encadenada de matrices	132
11.5.1. Planteamiento	132
11.5.2. Solución por programación dinámica	133
Ejercicios resueltos	134
Ejercicios propuestos	142
Cuestiones de autoevaluación	144
Referencias bibliográficas	145

11.1. Método general

11.1.1. Ideas generales

La programación dinámica se suele utilizar en problemas de optimización, donde una solución está formada por una serie de decisiones. La solución óptima se obtiene siempre, al contrario que pasa en la técnica de avance rápido, donde puede encontrarse solución o no, y la solución que se encuentra puede ser una óptima o no serlo.

Igual que la técnica divide y vencerás, resuelve el problema original combinando las soluciones para subproblemas más pequeños. Sin embargo, la programación dinámica no utiliza recursividad, sino que almacena los resultados de los subproblemas en una tabla, calculando primero las soluciones para problemas más pequeños. Con esto se pretende evitar la repetición de cálculos que se puede dar cuando se resuelve un problema con llamadas recursivas a problemas más pequeños, con lo que se repiten llamadas a problemas idénticos, lo que supone la repetición de cálculos innecesariamente, pues se podrían haber guardado las soluciones obtenidas anteriormente.

Ejemplo 11.1 Los números de Fibonacci están definidos por la recurrencia $F(n) = F(n - 1) + F(n - 2)$, con casos base $F(1) = F(2) = 1$.

Para calcular un número de Fibonacci se puede usar un método recursivo:

```

operación Fibonacci(n:entero):entero
    si n≤2 entonces
        devolver 1
    sino
        devolver Fibonacci(n-1)+Fibonacci(n-2)
    finsi
```

Esta solución se puede ver como un esquema divide y vencerás, pues para resolver un problema de tamaño n resolvemos dos problemas de tamaño menor, y combinamos los resultados sumándolos. De esta forma se repiten muchos cálculos. Por ejemplo, para calcular $F(5)$ tenemos $F(5) = F(4) + F(3) = F(3) + F(2) + F(3)$, con lo que vemos que $F(3)$ se calcula dos veces. Esta repetición de cálculos produce un tiempo exponencial, como podemos ver resolviendo la recurrencia $t(n) = t(n - 1) + t(n - 2) + 1$, que da como resultado $t(n) \in \Theta\left(\frac{1+\sqrt{5}}{2}\right)^n$.

Para resolver este problema se puede calcular los valores de menor a mayor empezando por 1, e ir guardando los resultados hasta llegar al valor n :

```

operación Fibonacci(n:entero):entero
    f[1]:=1
    f[2]:=1
    para i=3,...,n hacer
        f[i]:=f[i-1]+f[i-2]
    fins
    devolver f[n]
```

De esta forma el tiempo de ejecución es $\Theta(n)$.

Con la programación dinámica se evitan cálculos repetidos guardando una tabla de resultados de subproblemas, 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, pues se empieza con el problema original y se descompone en sucesivos pasos en problemas de menor tamaño.

Para poder aplicar a un problema la técnica de programación dinámica a un problema de optimización se debe cumplir el **principio de optimalidad de Bellman**, que dice que cada subsecuencia de una secuencia óptima debe ser secuencia óptima del subproblema correspondiente. Por tanto, el primer paso para aplicar programación dinámica a la solución de un problema es comprobar si se cumple el principio de optimalidad. Algunas veces se cumple dependiendo de la forma en que se enfoque el problema, como vemos en el ejemplo siguiente.

Ejemplo 11.2 Consideramos el grafo multietapa de la figura 10.1. El camino de longitud mínima del nodo 1 al 9 está formado por las aristas (1,3), (3,5), (5,7) y (7,9). Si consideramos como subproblemas la obtención del camino de longitud mínima del nodo 1 a los sucesivos niveles del grafo, el principio de optimalidad no se cumple, pues al segundo nivel el camino mínimo se obtiene con la arista (1,4), que no forma parte de la secuencia solución del problema total.

Tampoco se puede obtener la secuencia óptima como concatenación de secuencias óptimas de subproblemas, pues para pasar del nivel uno al dos la arista óptima es (1,4), para pasar del dos al tres es la (3,5), etc., y concatenando las aristas ni siquiera tenemos un camino del nodo 1 al 9.

Esto no quiere decir que el problema no cumpla el principio de optimalidad, sino que no lo cumple tal como lo hemos enfocado. Si consideramos como subproblema la obtención del camino mínimo del nodo origen a cada uno de los restantes nodos, la arista (1,3) es obviamente la solución óptima para llegar del nodo 1 al 3, la sucesión (1,3), (3,5) es óptima para llegar del 1 al 5, y la sucesión (1,3), (3,5), (5,7) lo es para llegar del 1 al 7.

Los pasos a seguir para obtener un algoritmo de programación dinámica son:

- Obtención de la ecuación recurrente que liga la solución de problemas de un tamaño con soluciones de problemas de tamaños menores.
- Establecimiento de los casos base y de su valor.
- Definición de las tablas a utilizar por el algoritmo, y determinación de la forma en que se rellenan.
- Determinación de cómo se recompone la solución global a partir de los valores de las tablas.

Vemos a continuación cada uno de los pasos en el ejemplo de los números de Fibonacci, y los analizaremos con más detalle en los ejemplos de las secciones siguientes.

Ejemplo 11.3 En el cálculo de los números de Fibonacci la ecuación de recurrencia es la que los define ($F(n) = F(n - 1) + F(n - 2)$), y los casos casos base son $F(1) = F(2) = 1$.

La tabla que se utiliza es el array de 1 a n donde se almacenan los números.

En este caso no hay que recomponer la solución, pues no se obtiene como una secuencia de decisiones, y lo único que importa es el valor del número de Fibonacci.

11.1.2. Programación dinámica en problemas que no son de optimización

La programación dinámica se utiliza normalmente para resolver problemas de optimización, pero sus ideas se pueden utilizar en la solución de otros tipos de problemas, en los que se puede trabajar ascendenteamente, solucionando problemas de tamaños menores, almacenarlos y llegar hasta el tamaño de problema que se quiere resolver. Esto se hace en el cálculo de los números de Fibonacci, y puede hacerse en problemas donde las soluciones para un cierto tamaño se obtienen en función de los de tamaños inferiores por medio de alguna fórmula que sustituye al mínimo o máximo de un problema de optimización. Lo vemos con un ejemplo.

Ejemplo 11.4 Dada una cadena de caracteres (SOLOCASO), y un diccionario formado por un conjunto de cadenas de caracteres (SO, SOL, SOLO, O, LO, LOCA, LOCAS, OCA, OCAS, OCASO, CASO, AS, ASO), se pretende encontrar todas las “frases” que se pueden obtener en la cadena completa utilizando palabras del diccionario. Por una frase entendemos concatenaciones de palabras del diccionario: las frases estarán formadas por palabras del diccionario sin espacios entre ellas. Las palabras del diccionario se pueden repetir en una “frase”.

Se puede ver como un problema de programación dinámica. Se va contando desde el final de la cadena (o el principio) la cantidad de palabras que coinciden desde esa posición hasta el final de la palabra con el trozo correspondiente de cadena. Cuando una palabra del diccionario coincide hasta el final de la cadena se cuenta 1, cuando se sale se cuenta cero; estos son los casos base. Cuando una palabra acaba dentro de la cadena coincidiendo de principio a final se cuenta el número en la posición siguiente a donde acaba. La fórmula de recurrencia consiste en contar en cada posición la suma proporcionada por todas las palabras en el diccionario:

$$NF(pos) = \sum_{pal \in \text{diccionario}} NF(pos + longitud(pos)) * coincide(pal, pos)$$

donde para que sea válida la fórmula ponemos $NF(n + 1) = 1$ (las que acaban en el último carácter), y $NF(m) = 0$ si $m > n + 1$ (las que no coinciden hasta el final), y $coincide(pal, pos)$ devuelve 1 si pal está en la cadena a partir de pos , y cero en otro caso.

Además se puede guardar la información para recomponer las cadenas. La información generada en el ejemplo es:

	S	O	L	O	C	A	S	O
frases	8	3	3	4	1	2	1	1
SO	x						x	
SOL	x							
SOLO	x							
O		x		x				x
LO			x					
LOCA			x					
LOCAS			x					
OCA				x				
OCAS				x				
OCASO				x				
CASO					x			
AS						x		
ASO						x		

Se tiene las soluciones de todos los subproblemas, lo que es típico de la programación dinámica. Las 8 frases se obtienen empezando por la primera posición viendo qué cadenas aparecen:

SO, SOL y SOLO

y después de cada una cuáles van hasta completar la frase:

SO LO CASO, SO LOCA SO, SO LOCAS O, SOL O CASO, SOL OCA SO, SOL OCAS O, SOL OCASO, SOLO CASO.

11.1.3. Análisis de tiempo de ejecución y ocupación de memoria

En general el estudio del tiempo de ejecución y ocupación de memoria es muy simple para esta técnica, al tratarse de llenar una tabla de soluciones.

El tiempo de ejecución depende del tipo de problema a resolver, pero en general será de la forma: Tamaño de la tabla * Tiempo de llenar cada elemento de la tabla.

Otro aspecto importante de los algoritmos de programación dinámica es la alta ocupación de memoria, al necesitarse una tabla para almacenar los resultados parciales.

Muchos de los cálculos y de la ocupación de memoria pueden ser innecesarios, pues los subproblemas a que corresponden no aparecerán en la solución final, pero esto puede ser preferible a la repetición de cálculos que se daría si el problema se resolviera de manera recursiva.

Los aspectos de tiempo de ejecución y ocupación de memoria se verán en los ejemplos del capítulo, y los vemos con el problema de los números de Fibonacci.

Ejemplo 11.5 En el problema de los números de Fibonacci el coste de memoria es lineal, pues la tabla que se utiliza es un array de 1 a n .

El tiempo de ejecución también es lineal, pues el tamaño de la tabla es n y el coste de calcular cada elemento de la tabla es constante.

11.2. Problema del grafo multietapa

En este problema se trata de obtener la longitud mínima en p pasos, con p uno menos que el número de niveles del grafo. Como la solución con p pasos se expresa en función de soluciones de $p - 1$ pasos, estas en función de las de $p - 2$ pasos, etc., habrá que obtener una ecuación que relaciona soluciones con q pasos con las de $q - 1$ pasos, con $1 \leq q \leq p$. Además, para obtener la solución hasta el nodo destino (n) hay que llegar desde un nodo del nivel anterior, a este de otro de nivel anterior, etc., con lo que la ecuación tiene que ligar las distancias hasta nodos de un nivel con las distancias hasta nodos de nivel anterior. La ecuación será:

$$D(x, q) = \min_{y \in N_{i-1}} \{D(y, q-1) + d(y, x)\}$$

si $x \in N_i$ y $d(y, x)$ representa la longitud de la arista que une el nodo y con el x , siendo $d(y, x) = \infty$ si no existe esa arista. En $D(\text{nodo}, \text{pasos})$ el primer parámetro (*nodo*) representa el nodo al que se calcula la distancia, y el segundo (*pasos*) el número de pasos hasta llegar a ese nodo.

El caso base será con cero pasos: $D(x, 0) = 0$ si x es el nodo origen, $D(x, 0) = \infty$ si x no es el nodo origen.

La tabla podría tener dos dimensiones que se corresponden con los parámetros de la función D . Tendría p filas y n columnas. Pero, como en cada paso solo se puede llegar a nodos de un nivel, solo será necesario una fila, que contendrá las distancias para todos los valores de q . Con el grafo de la figura 10.1 sería:

1	2	3	4	5	6	7	8	9

el caso base (paso cero) corresponde al nodo 1. En el paso uno se calculan los valores para los nodos 2, 3 y 4. En el paso dos para los nodos 5 y 6. En el paso tres para los 7 y 8. Y finalmente en el paso cuatro se obtiene la longitud del camino mínimo:

1	2	3	4	5	6	7	8	9
0	3	4	2	5	6	7	10	9

En la posición nueve tenemos la longitud mínima, pero no sabemos el camino a que corresponde. Para poder recomponerlo se utiliza otra tabla de las mismas dimensiones que la anterior, donde se guardan las decisiones tomadas para obtener cada uno de los valores:

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

Esta tabla se usa para recomponer la solución empezando por la última posición. Obtenemos la arista (7,9), después en la posición 7 vemos que se llega a partir del nodo 5. De la posición 5 vamos a la arista 3, y de la 3 a la 1. Los nodos por los que se pasa son: 9, 7, 5, 3, 1.

La ocupación de memoria es lineal.

Para obtener el tiempo de ejecución tenemos en cuenta que hay que calcular n valores, y cada uno se obtiene calculando un mínimo de tantos casos como nodos tenga el nivel anterior. Dependiendo de la implementación puede que ese mínimo se calcule de n valores, con lo que el coste sería $\Theta(n^2)$. Si se usan sólo los nodos del nivel anterior el coste será $\sum_{i=2}^p |N_i| |N_{i-1}|$.

11.3. Problema del cambio de monedas

11.3.1. Planteamiento de la solución por programación dinámica

Recordamos en qué consiste el problema: dado un conjunto de n tipos de monedas, cada una con valor v_i , con un número ilimitado de monedas de cada tipo, y dada una cantidad C , encontrar el número mínimo de monedas que tenemos que usar para obtener esa cantidad.

El algoritmo voraz es muy eficiente, pero no funciona en todos los casos. En algunos casos puede no encontrar solución aunque la haya, y en otros casos puede encontrar una solución no óptima.

Para resolverlo por programación dinámica aplicamos los cuatro pasos indicados.

Los subproblemas vendrán en función del número de monedas i , con $1 \leq i \leq n$, y de la cantidad a conseguir Q , con $0 \leq Q \leq C$. $Cambio(i, Q)$ representa el valor de esa solución, y queremos obtener $Cambio(n, C)$. Dada una moneda de valor v_i , cuando hay que devolver una cantidad Q , se puede dar desde 0 monedas de ese valor hasta $\lfloor \frac{Q}{v_i} \rfloor$ monedas. Por tanto, la fórmula de recurrencia es:

$$Cambio(i, Q) = \min_{k=0,1,\dots,\lfloor \frac{Q}{v_i} \rfloor} \{Cambio(i-1, Q - kv_i) + k\}$$

Los casos base corresponden a $i \leq 0$ o $Q < 0$, con $Cambio(i, Q) = \infty$ para que cualquier solución válida los mejore, ya que no puede haber solución en esos casos. Cuando hay que devolver una cantidad cero, la solución es cero: $Cambio(i, 0) = 0$ si $i \geq 0$.

La tabla de soluciones tendrá tantas filas como tipos de monedas, y tantas columnas como cantidad C a devolver, y cada fila se rellena utilizando la ecuación de recurrencia y la fila anterior; salvo la fila uno, para la que se utilizan los casos base correspondientes a cero monedas.

Para recomponer la solución se utiliza otra tabla auxiliar donde se guardan las decisiones (el número de monedas a dar de un cierto valor) para cada uno de los subproblemas.

Ejemplo 11.6 En el problema con $n = 3$, $C = 7$, y $v = (1, 2, 4)$, las tablas son:

	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7
2	1	1	2	2	3	3	4
4	1	1	2	1	2	2	3

	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7
2	0	1	1	2	2	3	3
4	0	0	0	1	1	1	1

y la solución se obtiene accediendo en la tabla auxiliar a la última fila y columna, con lo que sabemos que se da una moneda de valor 4. En la fila anterior la cantidad a devolver es 3, y en la posición (2,3) hay un 1, con lo que se da una moneda de valor 2, y en la fila 1 la cantidad a devolver es 1, con lo que se da otra moneda de valor 1. La solución es, por tanto, dar una moneda de cada tipo.

Otra posibilidad para resolver este problema consiste en aplicar la fórmula:

$$\text{Cambio}(i, Q) = \min\{\text{Cambio}(i - 1, Q), 1 + \text{Cambio}(i, Q - v_i)\}$$

Al calcularse en cada paso el mínimo de dos valores, con esta fórmula el tiempo de ejecución sería menor.

11.3.2. Algoritmo

El algoritmo será:

Algoritmo 11.1 Problema de la devolución de monedas con programación dinámica.

```

operación Monedas(C:entero; v:array[1..n] de entero; var s:array[1..n] de entero;
var c:entero)
para i:=1,...,n-1 hacer
    para j:=1,...,C hacer
        //Se calcula el mínimo teniendo en cuenta los casos base
        Cambio[i,j]:=mink=0,1,...,[j/vi]{Cambio(i-1,j-kvi) + k}
        Aux[i,j]:=el valor con que se obtiene el mínimo
    finpara
    finpara
    //De la última fila sólo hay que calcular el último valor
    c:=mink=0,1,...,[C/vn]{Cambio(n-1,C-kvn) + k}
    Aux[n,C]:=el valor con que se obtiene el mínimo
    //Y se recomponen la solución
    para i:=n,...,1 hacer
        s[i]:=Aux[i,C]
        C:=C-s[i]v[i]
    finpara

```

La composición de la solución tiene coste lineal, con lo que el coste del algoritmo será el de llenar las tablas, que varía para los distintos valores:

$$\sum_{i=1}^{n-1} \sum_{j=1}^C \left(\lfloor \frac{j}{v_i} \rfloor + 1 \right)$$

y se puede acotar considerando que el máximo no será de más de C valores, con lo que se tiene la cota $O(nC^2)$.

La ocupación de memoria tiene coste $\Theta(nC)$.

11.4. Problema de la mochila 0/1

11.4.1. Planteamiento de solución por programación dinámica

Consideramos el mismo problema de la mochila del capítulo anterior: mochila de capacidad M , y n objetos de beneficios $b = (b_1, b_2, \dots, b_n)$ y pesos $p = (p_1, p_2, \dots, p_n)$; pero en 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 o 1 dependiendo de que no se introduzca o sí se introduzca el elemento en la mochila. El problema consiste en maximizar $\sum_{i=1}^n b_i x_i$ sujeto a $\sum_{i=1}^n p_i x_i \leq M$ y $x_i = 0 \text{ o } x_i = 1 \forall i, 1 \leq i \leq M$.

Podemos llamar a este problema *Mochila*(n, M). Para resolverlo tendremos que resolver problemas más pequeños que llamaremos *Mochila*(i, X), que corresponden al problema de la mochila 0/1 con los objetos numerados del primero al i y con capacidad de la mochila X . Para obtener la ecuación de recurrencia observamos que, para un objeto i , y supuesto que ya se han tomado las decisiones de los $i - 1$ primeros objetos, este objeto puede meterse o no en la mochila, con lo que hay que calcular el máximo de los dos valores correspondientes:

$$\text{Mochila}(i, X) = \max\{\text{Mochila}(i - 1, X), b_i + \text{Mochila}(i - 1, X - p_i)\}$$

Para obtener los casos base hay que determinar valores que pueden aparecer al aplicar la fórmula y que corresponden a configuraciones no válidas. Esto ocurre cuando llegamos a capacidad de la mochila negativa o a problemas con un número negativo de objetos, con lo que se define:

$$\text{Mochila}(i, X) = -\infty, \text{ si } X < 0 \text{ o } i < 0$$

Se asigna el valor $-\infty$ porque corresponden a configuraciones no válidas, y cualquier configuración válida los mejorará al hacer el máximo.

Casos base que corresponden a configuraciones válidas son los de no tener ningún objeto o tener una mochila de capacidad cero. En estos casos el beneficio es cero:

$$\text{Mochila}(0, X) = 0, \text{ si } X \geq 0$$

$$\text{Mochila}(i, 0) = 0, \text{ si } i \geq 0$$

Necesitamos una tabla (*Mochila*) para guardar los beneficios de los subproblemas, con n filas (tantas como objetos) y M columnas (tantas como capacidad de la mochila). Los casos base de cero objetos o capacidad cero se pueden incluir en la tabla o no incluirlos. Esta tabla se rellena por filas, y de la última fila solo es necesario calcular el valor *Mochila*[n, M].

Para recomponer la solución se puede utilizar una tabla auxiliar de las mismas dimensiones, donde en cada posición se almacena el valor cero si el máximo se obtiene no incluyendo el objeto, y el valor uno si se obtiene incluyendo el objeto. La solución se recompone accediendo a la posición $Aux[n, M]$, si es uno se incluye el objeto n en la solución y se actualiza $M = M - p_n$ y si es cero se deja M con el valor que tiene, pues el objeto n no se mete en la mochila. Se accede en la fila $n - 1$ a $Aux[n - 1, M]$ y se hace la misma actualización. Y así hasta llegar a la fila uno.

En este problema esta segunda tabla no es necesaria, porque se puede comprobar si un objeto i se incluye o no en la mochila comparando $Mochila[i - 1, X]$ con $Mochila[i, M]$, si son iguales el objeto i no se incluye, y si son distintos sí se incluye.

La ocupación de memoria es proporcional al tamaño de las tablas (o la tabla), $\Theta(nM)$.

El tiempo de ejecución para llenar las tablas es: tamaño de la tabla *coste del cálculo del máximo $\in \Theta(nM)$. Y el tiempo de recomponer la solución es lineal.

Ejemplo 11.7 En el problema con $n = 3$, $M = 6$, $b = (1, 2, 5)$, y $p = (2, 3, 4)$, la tabla de beneficios queda:

	1	2	3	4	5	6
1	0	1	1	1	1	1
2	0	1	2	2	3	3
3	0	1	2	5	5	6

y la auxiliar sería:

	1	2	3	4	5	6
1	0	1	1	1	1	1
2	0	0	1	1	1	1
3	0	0	0	1	1	1

y la solución se obtiene accediendo en la tabla auxiliar a la última fila y columna, con lo que sabemos que se incluye el objeto tres: $x_3 = 1$, se actualiza $M = 6 - 4 = 2$, se accede a $Aux[2, 2] = 0$, con lo que $x_2 = 0$, no se actualiza M , se accede a $Aux[1, 2] = 1$, con lo que $x_1 = 1$. La solución es $x = (1, 0, 1)$ y el beneficio es 6.

No es necesario almacenar todos los datos de la última fila, ni la tabla auxiliar, porque la solución se puede recomponer accediendo solo a la tabla de beneficios: $Mochila[3, 6] \neq Mochila[2, 6]$, por tanto se toma el objeto 3, se actualiza $M = 2$, $Mochila[2, 2] = Mochila[1, 2]$, y el objeto 2 no se mete en la mochila y, como $Mochila[1, 2] \neq 0$, el objeto 1 se mete en la mochila.

11.4.2. Algoritmo

El algoritmo será:

Algoritmo 11.2 Problema de la mochila 0/1 por programación dinámica.

operación $Mochila(M:\text{entero}; b, p:\text{array}[1..n] \text{ de } \text{entero}; \text{var } s:\text{array}[1..n] \text{ de } \text{entero};$

```

var c:entero
para i:=1,...,n-1 hacer
    para j:=1,...,M hacer
        //Se calcula el mínimo teniendo en cuenta los casos base
        Mochila[i,j]:=min{Mochila[i - 1, j], Mochila[i - 1, j - pi] + bi}
    finpara
finpara
//De la última fila solo hay que calcular el último valor
Mochila[n,M]:=min{Mochila[n - 1, M], Mochila[n - 1, M - pn] + bn}
c:=Mochila[n,M]
//Y se recomponen la solución sin utilizar tabla auxiliar
para i:=n,...,2 hacer
    si Mochila[i-1,M]=Mochila[i,M] entonces
        s[i]:=0
    sino
        s[i]:=1
        M:=M-p[i]
    finsi
finpara
si Mochila[1,M]≠ 0 entonces
    s[1]:=1
sino
    s[1]:=0
finsi

```

El esquema es idéntico al del problema de las monedas, salvo que en este caso se obtiene el mínimo de dos valores, con lo que el coste es $\Theta(nM)$, y no se utiliza una tabla auxiliar de decisiones para la recomposición de la solución, y la ocupación de memoria coincide con la ocupación de la tabla, $\Theta(nM)$.

11.5. Multiplicación encadenada de matrices

11.5.1. Planteamiento

Tenemos las matrices M_1, M_2, \dots, M_n , que queremos multiplicar para obtener $M = M_1 M_2 \dots M_n$. Puesto que el producto es asociativo, habrá muchas formas de realizar las multiplicaciones. Cada colocación de los paréntesis indica un orden en el que se realizan las operaciones. El producto de dos matrices de dimensiones $n \times m$ y $m \times r$ requiere de nmr multiplicaciones escalares. Según el orden de las multiplicaciones, el número total de multiplicaciones escalares necesarias puede variar considerablemente. Por ejemplo, si queremos obtener $M = M_1 M_2 M_3$ con matrices de dimensiones 5×6 , 6×3 y 3×2 , si hacemos las multiplicaciones en el orden $(M_1 M_2) M_3$ el número de multiplicaciones escalares es 120, pero si agrupamos como $M_1 (M_2 M_3)$ es 96.

Queremos obtener el orden en que hay que hacer las multiplicaciones matriciales para que el número de multiplicaciones escalares sea mínimo. Tenemos un problema de

optimización donde la solución se obtiene en $n - 1$ pasos. En cada paso se indica la multiplicación a realizar. Por ejemplo, al orden de multiplicación $M_1(M_2M_3)$ le corresponde la solución $s = (2, 1)$. Se entiende que tras hacer la multiplicación de la posición 2 se obtiene el resultado, M_2M_3 , y al hacer la multiplicación de la posición 1 queda $M_1(M_2M_3)$.

11.5.2. Solución por programación dinámica

Queremos obtener el número mínimo de productos escalares necesarios para realizar la multiplicación entre la matriz 1 y la n . Como para obtener este valor pondremos paréntesis entre cualesquiera dos matrices, necesitaremos calcular los subproblemas $Multi(i, j)$, con $i \leq j$. Por tanto, si en la tabla de subsoluciones la i representa las filas y la j las columnas, solo se calculan los valores de la parte triangular superior.

Las dimensiones de las matrices las consideramos almacenadas en un array $d[0, \dots, n]$, con la matriz M_i de dimensión $d[i - 1] \times d[i]$.

La ecuación de recurrencia es:

$$Multi(i, j) = \min_{i \leq k < j} \{M(i, k) + M(k + 1, j) + d[i - 1]d[k]d[j]\}$$

y los casos base son:

$$Multi(i, i) = 0$$

ya que en ese caso no se hace ninguna multiplicación. Y

$$Multi(i, i + 1) = d[i - 1]d[i]d[i + 1]$$

pues en ese caso sólo hay una posibilidad para hacer la multiplicación.

Usaremos dos tablas, una para las soluciones de los subproblemas y otra para almacenar las decisiones. Las tablas son de dimensión $n \times n$, por lo que la ocupación de memoria es $\Theta(n^2)$, aunque solo se utiliza la parte triangular superior de las tablas. Se llenan por diagonales, empezando por la diagonal principal hasta llegar a la posición $(1, n)$. Vemos el funcionamiento con un ejemplo.

Ejemplo 11.8 Consideramos $n = 4$, y $d = (5, 6, 4, 2, 3)$. Inicialmente tenemos los casos base en la diagonal principal y la primera superdiagonal:

	1	2	3	4
1	0	120		
2		0	48	
3			0	24
4				0

y se obtienen los valores de la segunda y tercera superdiagonal y los de la tabla auxiliar:

	1	2	3	4
1	0	120	108	138
2		0	48	84
3			0	24
4				0

	1	2	3	4
1		1	1	3
2			2	3
3				3
4				

El número mínimo de multiplicaciones es 138, y se obtiene haciendo la última multiplicación la de la posición 3, $(M_1 M_2 M_3) M_4$. Para obtener el orden en que se hace la multiplicación de las matrices de la primera a la tercera se accede a la tabla auxiliar a la posición (1, 3), y la agrupación es $(M_1 (M_2 M_3)) M_4$.

El tiempo de ejecución es $\sum_{i=3}^n \sum_{j=i}^n (i - 1) \in o\left(\frac{n^3}{6}\right)$.

Ejercicios resueltos

Ejercicio 11.1 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. Solo 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 e y la columna, por lo que si el laberinto tiene n filas y m columnas tendremos $0 < x \leq n$ y $0 < y \leq m$. 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:

$$d((a, b), (a + 1, b)) = 1 \text{ si } l(a, b) = l(a + 1, b) = 1$$

$$d((a, b), (a - 1, b)) = 1 \text{ si } l(a, b) = l(a - 1, b) = 1$$

$$d((a, b), (a, b + 1)) = 1 \text{ si } l(a, b) = l(a, b + 1) = 1$$

$$d((a, b), (a, b - 1)) = 1 \text{ si } l(a, b) = l(a, b - 1) = 1$$

$$d((a, b), (c, d)) = +\infty, \text{ en otro caso, incluyendo que uno de los dos esté fuera del tablero.}$$

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:

$$\begin{aligned}
 C((x, y), (z, t)) = \min\{ &C((x, y), (z - 1, t)) + d((z - 1, t), (z, t)), \\
 &C((x, y), (z + 1, t)) + d((z + 1, t), (z, t)), \\
 &C((x, y), (z, t - 1)) + d((z, t - 1), (z, t)), \\
 &C((x, y), (z + 1, t)) + d((z + 1, t), (z, t)) \}
 \end{aligned}$$

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, sustituiremos cada fila de la tabla original por una tabla de las mismas dimensiones que el tablero, y reescribiremos en esa tabla las distancias, pues no es necesario guardar las distancias para un número determinado de pasos una vez se han calculado para un número mayor. Así, 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:

					$t1$						$t2$
0	∞	∞	∞	∞	∞	—	—	—	—	—	—
∞	∞	∞	∞	∞	∞	—	—	—	—	—	—
∞	∞	∞	∞	∞	∞	—	—	—	—	—	—
∞	∞	∞	∞	∞	∞	—	—	—	—	—	—
∞	∞	∞	∞	∞	∞	—	—	—	—	—	—
0	1	∞	∞	∞	∞	—	Iz	—	—	—	—
1	∞	∞	∞	∞	∞	Ar	—	—	—	—	—
∞	∞	∞	∞	∞	∞	—	—	—	—	—	—
∞	∞	∞	∞	∞	∞	—	—	—	—	—	—
∞	∞	∞	∞	∞	∞	—	—	—	—	—	—
0	1	2	∞	∞	∞	—	Iz	Iz	—	—	—
1	∞	∞	∞	∞	∞	Ar	—	—	—	—	—
∞	∞	∞	∞	∞	∞	—	—	—	—	—	—
∞	∞	∞	∞	∞	∞	—	—	—	—	—	—
∞	∞	∞	∞	∞	∞	—	—	—	—	—	—
0	1	2	3	∞	∞	—	Iz	Iz	Iz	—	—
1	∞	3	∞	∞	∞	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	-	Ab	-
De	Ar	Iz	Iz	Iz

Con lo que el camino de longitud mínima tiene longitud 10 y se obtiene utilizando la tabla $t2$ empezando por el nodo (5,5):

(5,5), (5,4), (5,3), (5,2), (4,2), (3,2), (3,3), (2,3), (1,3), (1,2), (1,1)

Ejercicio 11.2 Dada una serie de trabajos a, b, c, \dots 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, \dots 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				tabla1			
	a	b	c		a	b	c
2	4	3	3	2	b	a	b
3	7	7	6	3	b	a	b
4	11	10	10	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 $baba$, con valor 11.

Ejercicio 11.3 Dada una tabla $n \times n$ de 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 $M:\text{array}[1..n, 1..n]$ y otra auxiliar $S:\text{array}[1..n, 1..n]$ 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: $(1, 1), (1, 2), (2, 2), (3, 2), (3, 3)$.

Ejercicio 11.4 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..n] de enteros`, y teniendo de cada tipo una cierta cantidad de monedas, estando estas cantidades almacenadas en un array *cantidad*: `array[1..n] de enteros` (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:

$$M(i, X) = \min_{k=0, 1, \dots, \min\{\lfloor \frac{X}{tipos[i]} \rfloor, cantidad[i]\}} \{M(i - 1, X - k * tipos[i]) + k\}$$

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 hay que devolver una cantidad $X - k * \text{tipos}[i]$.

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 cálculos 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:

```

para  $k:=n, \dots, 1$  hacer
    monedas[ $k$ ]:=s[ $k, C$ ]
     $C:=C-\text{monedas}[k]*\text{tipos}[k]$ 
finpara

```

Para estudiar el tiempo de ejecución hay que tener en cuenta que este 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 solo 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:

$$\sum_{i=1}^n \left(a + \sum_{j=1}^C (b + j) \right) = an + bCn + \frac{C}{2}n + \frac{C^2}{2}n \in \Theta(C^2n)$$

con lo que el tiempo de ejecución es del orden $O(C^2n)$.

Ejercicio 11.5 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 recomponen 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 $M(p, x, y, z, t) = \min\{M(p-1, x, y, z, t), M(p-1, x, y, z-1, t-2) + 1, M(p-1, x, y, z-1, t+2) + 1, M(p-1, x, y, z+1, t-2) + 1, M(p-1, x, y, z+1, t+2) + 1, M(p-1, x, y, z-2, t-1) + 1, M(p-1, x, y, z-2, t+1) + 1, M(p-1, x, y, z+2, t-1) + 1, M(p-1, x, y, z+2, t+1) + 1\}$.

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 menores que 1, 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)	...	(2, 1)	...	(n, n)
0						
1						
:						
$n^2 - 1$						

El problema que se quiere resolver es $M(n^2 - 1, x_o, y_o, x_d, y_d)$, 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 $\Theta(n^4)$, pues hay que llenar 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.

Este problema en concreto se podría resolver de una manera mejor, pues mucha de la información almacenada en las tablas anteriores es innecesaria, ya que 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, con origen en $(1,1)$ y destino $(5,5)$, los pasos sucesivos serían:

0	∞		∞	∞
∞		∞	∞	
∞	1			∞
	∞	∞	∞	
∞	∞		∞	

(-1, -1)				

0	∞	2	∞	∞
∞		∞	∞	
∞	1			∞
	∞	∞	∞	
∞	∞	2	∞	

(-1, -1)			(3, 2)	

0	∞	2	∞	∞
∞		∞	∞	3
∞	1		3	∞
3	∞	∞	∞	3
∞	∞	2	∞	

(-1, -1)		(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.

Si estuviéramos en un problema de minimización del camino con pesos en las aristas, la solución que se obtiene al llegar por primera vez al destino podría mejorarse con un camino con más pasos (recuérdese el problema del camino más corto en un grafo), por lo que no se pararía al llegar al destino, sino cuando en un paso no se modifique ningún valor de la tabla de caminos mínimos.

Ejercicios propuestos

Ejercicio 11.6 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.

Ejercicio 11.7 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 11.1. 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).

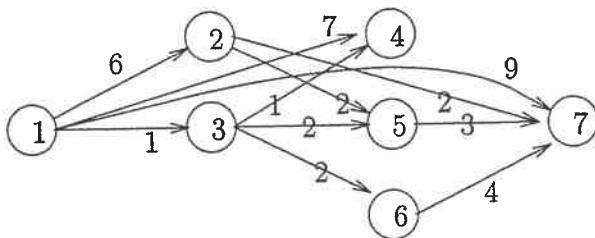


Figura 11.1: Grafo multietapa modificado del ejercicio 11.7.

- Obtener la fórmula que nos permita resolver el problema por programación dinámica e indicar cuáles son los casos base y cuáles son sus valores.
- Indicar cómo se irán rellenando las tablas y cómo se recomponen las soluciones.
- Explicar el funcionamiento con el grafo ejemplo.
- Estudiar la ocupación de memoria.
- Estudiar el tiempo de ejecución.

Ejercicio 11.8 Se tiene un damero de dimensión $n \times n$, con valores enteros en las casillas. Se pretende resolver el problema de, dado un caballo en una posición inicial encontrar el camino de longitud mínima que puede realizar el caballo en exactamente p pasos. La longitud del camino es la suma de los valores de las casillas por las que pasa. Se resolverá el problema por programación dinámica, obteniendo la ecuación de recurrencia, los casos base, y explicando cómo funciona el algoritmo con el siguiente ejemplo, indicando las tablas a utilizar y cómo se rellenan y se recompone la solución:

2	5	4	3
2	1	2	2
3	4	1	2
3	2	2	3

casilla origen la señalada, (2,2), y $p = 3$.

Ejercicio 11.9 Resolver por programación dinámica el problema de la mochila 0/0.5/1, consistente en, dada una mochila de capacidad M y n objetos con beneficios $b = (b_1, b_2, \dots, b_n)$ y pesos $p = (p_1, p_2, \dots, p_n)$, encontrar la asignación de objetos a la mochila que maximice el beneficio si cada objeto puede meterse entero en la mochila, no meterse, o meterse la mitad.

Ejercicio 11.10 Resolver por programación dinámica el problema de dada una cadena de números, encontrar la subcadena ascendente o descendente con más elementos, siendo una subcadena una cadena dentro de la original, con los datos en el mismo orden que en la cadena original, pero que no tienen que estar todos consecutivos. Por ejemplo, en la cadena 5,9,8,4,6,3,7,1,2 la respuesta es 9,8,4,3,1.

Cuestiones de autoevaluación

Ejercicio 11.11 En todos los ejemplos vistos en este capítulo el problema se resuelve con la utilización de tablas. ¿Es necesario siempre la utilización de tablas en la aplicación de la técnica de programación dinámica? Justificar el uso de otras estructuras ilustrándolo con alguno de los ejemplos vistos.

Ejercicio 11.12 Programar por programación dinámica la resolución del problema del grafo multietapa.

Ejercicio 11.13 Identificar tipos de problemas a los que se aplica la técnica de programación dinámica (se pregunta por tipos de problemas, no ejemplos).

Ejercicio 11.14 Contestar si cada una de las siguientes afirmaciones es verdadera o falsa, justificando la respuesta y dando un contraejemplo en caso de que sea falsa:

- a) En la resolución de problemas por programación dinámica se usan siempre dos tablas.
- b) La forma en que se rellenan las tablas es primero la de beneficios y después la auxiliar que se utiliza para recomponer la solución.
- c) El coste de llenar las tablas es mayor que el de recomponer la solución.
- d) El tiempo de ejecución tiene el mismo orden que la ocupación de memoria.
- e) El tiempo de ejecución tiene coste cuadrático.

Referencias bibliográficas

La técnica de programación dinámica se analiza someramente en libros donde aparece junto con otras técnicas de diseño ([Aho88], [Heileman98]).

En otras referencias se dedica un capítulo completo a esta técnica [Troya84], [Cormen90], [Weis95], [Brassard97], [Rabhi99], [Baase00], y se incluyen algunos de los ejemplos estudiados y otros adicionales. Algunos de los ejemplos que se han visto en la parte de estructuras de datos y se pueden resolver por programación dinámica (algoritmo de Floyd para los caminos más cortos, árbol binario de búsqueda óptimo) aparecen en estos capítulos o en otros dedicados a algoritmos sobre grafos o árboles de búsqueda.

Capítulo 12

Backtracking

El backtracking y la ramificación y poda son métodos de resolución de problemas por búsqueda en un árbol de soluciones. Este árbol es un árbol lógico y se recorre de manera sistemática. En el backtracking la búsqueda se realiza en profundidad y retrocediendo (de ahí procede su nombre). La técnica se puede aplicar a distintos tipos de problemas, que incluyen búsqueda de una única solución, de todas las soluciones, o de una solución óptima.

Objetivos del capítulo:

- Entender el concepto de árbol de solución, y su utilización en la resolución de problemas. Comprender que en muchos casos el árbol es sólo una representación lógica del conjunto de todas las posibles soluciones y subsoluciones.
- Comprender la técnica de resolución de problemas por backtracking, entendiendo las características principales del backtracking y el concepto de búsqueda en profundidad.
- Identificar distintos tipos de problemas que se pueden resolver por backtracking y los esquemas para su resolución.
- Comprender el alto coste de un algoritmo por backtracking, lo que hace que para una resolución eficiente sea necesario utilizar alguna técnica de eliminación de nodos y de búsqueda de las soluciones.
- Comparación de la solución de problemas que se han estudiado en otras técnicas con la forma de resolverlos por backtracking, para poder identificar, ante problemas que se pueden resolver con distintas técnicas, cuál es la más apropiada.

Contenido del capítulo:

12.1. Método general	149
12.1.1. Ideas generales	149
12.1.2. Distintos tipos de árboles de soluciones	150
12.1.3. Esquemas de backtracking	152
12.1.4. Análisis de algoritmos de backtracing	157
12.2. Problema de las reinas	157
12.2.1. Planteamiento	157
12.2.2. Soluciones por backtracking	158
12.2.3. Estudio	162
12.3. Problema de la mochila	163
12.3.1. Planteamiento	163
12.3.2. Solución con árbol binario	164
12.3.3. Mejora de la función criterio	165
12.3.4. Modificación del orden de recorrido	166
12.3.5. Solución con árbol combinatorio	167
Ejercicios resueltos	168
Ejercicios propuestos	182
Cuestiones de autoevaluación	183
Referencias bibliográficas	183

12.1. Método general

12.1.1. Ideas generales

El **backtracking** (método de retroceso o vuelta atrás) es una técnica general de resolución de problemas, aplicable a problemas de optimización, a problemas en los que se quiere encontrar una única solución o todas las soluciones, tanto si se sabe que hay solución como si no se sabe.

El backtracking realiza una búsqueda exhaustiva y sistemática en el espacio de soluciones. Por ello, suele resultar ineficiente.

La solución de un problema de backtracking se puede expresar como una tupla (x_1, x_2, \dots, x_n) , satisfaciendo unas restricciones $P(x_1, x_2, \dots, x_n)$ y tal vez optimizando una cierta función objetivo. En cada momento, el algoritmo se encuentra en un cierto nivel k , con una solución parcial (x_1, \dots, x_k) . Si se puede añadir un nuevo elemento a la solución (x_{k+1}) , se genera y se avanza al nivel $k + 1$. Si no, se prueban otros valores de x_k . Si no existe ningún valor posible por probar, se retrocede al nivel anterior, $k - 1$. Dependiendo del tipo de problema que se está resolviendo, el proceso continúa hasta que se obtiene una solución completa del problema, o hasta que no quedan más posibles soluciones que evaluar. El resultado es equivalente a hacer un recorrido en profundidad en el árbol de soluciones. Sin embargo, este árbol es normalmente implícito, y no se almacena en memoria.

Ejemplo 12.1 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. En este caso la secuencia $(1,0,1)$ es una solución válida del problema, mientras que la $(0,1,1)$ no es solución del problema pues no cumple las restricciones.
- 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.

Con cualquiera de estas dos representaciones un backtracking generaría, de una manera sistemática y en profundidad, todas las posibles secuencias, y determinaría cuáles corresponden a soluciones que cumplen la restricción: $\sum_{i=1}^3 x_i n_i = 20$ con la primera representación, y $\sum_{i=1}^3 n_{x_i} = 20$, con $n_0 = 0$, en el segundo caso.

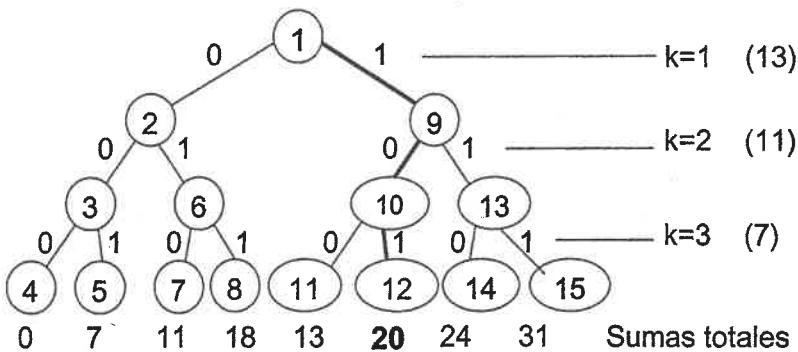


Figura 12.1: Árbol binario de soluciones.

12.1.2. Distintos tipos de árboles de soluciones

Las secuencias de decisiones corresponden a árboles de soluciones, donde cada nodo representa una solución parcial, y en cada paso del algoritmo se está en un nodo del árbol. El árbol indica un orden de ejecución, pero no se almacena en memoria.

Ejemplo 12.2 Los árboles de soluciones para el ejemplo 12.1, serían: el de la figura 12.1, para la primera representación, y el de la figura 12.2, para la segunda representación. 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.

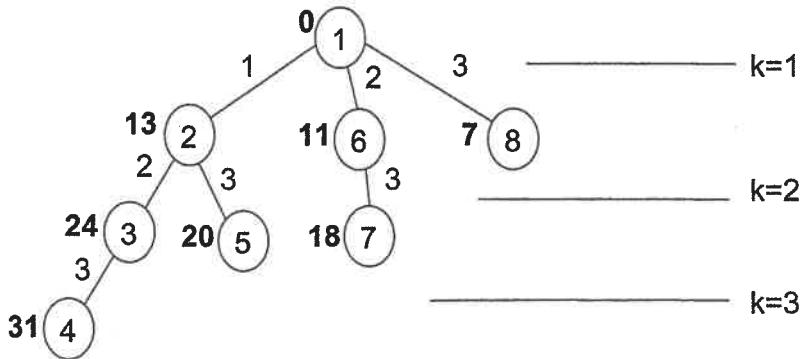


Figura 12.2: Árbol combinatorio de soluciones.

En el ejemplo anterior vemos dos tipos de árboles que se pueden utilizar en la solución de un mismo problema. Dependiendo del problema puede ser necesario utilizar árboles de otros tipos. Algunos de los árboles que nos pueden aparecer en la solución de problemas por backtracking son:

- **Árboles binarios** (figura 12.1), donde en cada nivel se decide si incluir o no en la solución un determinado valor. Un cero indicará que no se incluye y un uno que sí. El número de niveles coincide con la cantidad de decisiones a tomar.

- Árboles combinatorios (figura 12.2), donde en cada nivel se decide qué valor incluir en la solución, y no importa el orden en que se incluyan los valores. El número máximo de niveles coincide con la cantidad de decisiones a tomar, aunque hay nodos terminales en todos los niveles.
- Árboles permutacionales (figura 12.3), donde en cada nivel se decide qué valor incluir en la solución, e importa el orden en que se incluyen los valores. El número de niveles coincide con la cantidad de decisiones a tomar.
- Árboles n -arios, cuando en cada nivel se toma una decisión de entre n posibles. El número de niveles coincide con la cantidad de decisiones a tomar.
- También es posible que en distintos niveles o desde distintos nodos la cantidad de decisiones a tomar sea distinta, con lo que nodos distintos pueden tener número de descendientes distintos. Esto pasa por ejemplo en los árboles combinatorios y permutacionales.

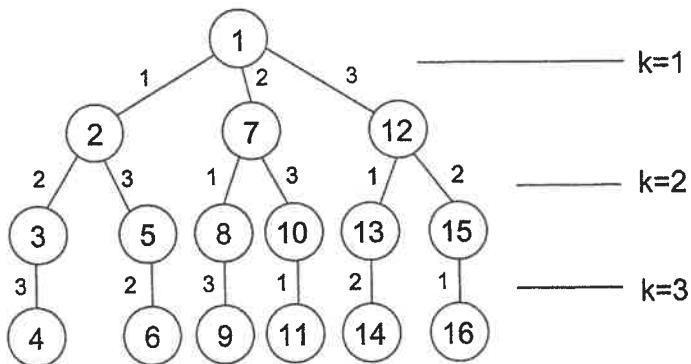


Figura 12.3: Árbol permutacional de soluciones.

Analizando los árboles vemos las características principales de un algoritmo por backtracking:

- El recorrido es un recorrido sistemático y en profundidad. Este recorrido sistemático se hará generando los posibles valores de las soluciones de una manera sistemática. En las figuras vemos que los descendientes de un nodo se generan de izquierda a derecha, pero el orden en que se generan puede variar (por ejemplo, se podrían generar los hijos de derecha a izquierda o generar primero los que se considera que tienen más probabilidad de llevar a la solución que buscamos). Por esto, hay que tener una función de generación que dependerá del nodo por el que vayamos (de la solución total o parcial generada hasta ese momento). En el árbol binario (figura 12.1), los valores a generar por esta función serán 0 y 1 en este orden, y en el árbol combinatorio (figura 12.2) 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, etc.; 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 árbol combinatorio en muchos 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 determinar 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 ejemplo, con árbol binario sólo son nodos solución los terminales en los que la suma de los números sea 20, pero en el árbol combinatorio cualquier nodo en el que la suma sea 20 es 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**.
- En los dos árboles del ejemplo 12.2 hemos representado todos los posibles nodos de acuerdo con la representación de la solución escogida, pero vemos que a partir de algunos nodos (el nodo 13 en el primer caso y el 3 en el segundo caso) no puede haber solución (en este ejemplo porque la suma de la solución parcial que llevamos es 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.

12.1.3. Esquemas de backtracking

En el apartado anterior hemos enumerado las funciones que debe tener un esquema para un algoritmo por backtracking. En este apartado analizamos esquemas que utilizan estas funciones.

Hay distintos tipos de problemas a los que corresponden esquemas distintos:

- Sabemos que hay alguna solución y solo se quiere encontrar una. En este caso el programa parará cuando encuentre la primera.
- Puede no haber solución pero, si la hay, solo 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 tratando. En este caso también puede que no haya 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.

Un esquema, utilizando las funciones antes enumeradas y en el caso en que hay soluciones y solo se quiere obtener una, podría ser el siguiente:

Algoritmo 12.1 Esquema de backtracking no recursivo, cuando se sabe que hay solución y solo se quiere encontrar una.

```

operación Backtracking-nr-1sol (var s:array[1..Max_nivel] de tipo)
    nivel:=1
    fin:=falso
    repetir
        s[nivel]:=generar(nivel,s)
        si solucion(nivel,s) entonces
            fin:=verdadero
        sino si criterio(nivel,s) entonces
            nivel:=nivel+1
        sino
            mientras NO mashermanos(nivel,s) hacer
                retroceder(nivel,s)
            finmientras
        finsi
    hasta fin=verdadero

```

donde:

- La variable *nivel* indica el nivel por el que se va recorriendo el árbol.
- La variable *fin* indica si se ha acabado ya el recorrido del árbol. En este caso se pone a verdadero cuando se encuentra una solución.
- 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 verdadero si el nodo por el que vamos es solución del problema.
- La función *criterio* devuelve verdadero si a partir de un nodo se puede llegar a una solución.
- La función *mashermanos* devolverá verdadero 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 **repetir** hasta que se encuentra una solución, y se supone que la solución se almacena en el array global *s*. El recorrido del árbol (ilógico!) se hace poniendo distintos valores en la solución. Vemos cómo serían las funciones con el ejemplo de la suma de números.

Por simplificar la escritura, en las funciones del siguiente ejemplo, y en las demás a lo largo de este capítulo, no indicaremos el tipo de los parámetros (y en algunos casos no incluiremos los parámetros) ni el valor que devuelven. Tampoco indicaremos en la mayoría de los casos qué parámetros son variables y cuales no. Todo esto es fácilmente deducible sabiendo la utilidad de cada una de las funciones.

Ejemplo 12.3 Volvemos a considerar el problema de obtener un subconjunto del conjunto de números $\{13, 11, 7\}$ cuya suma es 20, y suponemos que se sabe que hay al menos una solución. Utilizamos el árbol binario:

- El array solución se inicializa a -1, para generar los valores válidos (0 y 1) sumando uno al valor que hubiera anteriormente.

operación generar(*nivel,s*):

$$s[nivel] := s[nivel] + 1$$

- Una secuencia es solución si estamos en el último nivel y se cumple la restricción del problema:

operación solucion(*nivel,s*):

$$\text{devolver } nivel = n \text{ Y } \sum_{i=1}^n s[i] * v[i] = C$$

Si se encuentra el valor, independientemente del nivel en el que estemos, la función solucion podría acabar la ejecución del backtracking.

- Se puede seguir a partir de un nodo si no es terminal y la suma no excede del valor que se quiere obtener:

operación criterio(*nivel,s*):

$$\text{devolver } nivel \neq n \text{ Y } \sum_{i=1}^{nivel} s[i] * v[i] \leq C$$

- Los nodos que no corresponden al valor 1 tienen más hermanos:

operación mashermanos(*nivel,s*):

$$\text{devolver } s[nivel] \neq 1$$

- Al retroceder se deja el array solución con los valores iniciales:

operación retroceder(*nivel,s*):

$$s[nivel] := -1$$

$$nivel := nivel - 1$$

Con las funciones generar y retroceder nos movemos por el árbol de soluciones, y el resto de las funciones solo comprueban si se cumplen algunas condiciones.

El nodo por el que estamos en el árbol viene dado por los valores de s y $nivel$. Si queremos evitar los cálculos del sumatorio que se hacen al comprobar si es solución o cumple el criterio, podemos usar otra variable auxiliar $suma$, donde almacenamos la suma actual. Esta variable estará inicializada a cero, y describe el estado del nodo en el que estamos, junto con s y $nivel$, con lo que se modifica en las funciones generar y retroceder:

operación generar($nivel,s,suma$):

```

 $s[nivel]:=s[nivel]+1$ 
si  $s[nivel]=1$  entonces
     $suma:=suma+v[nivel]$ 
finsi
```

operación retroceder($nivel,s,suma$):

```

 $suma:=suma-v[nivel]$ 
 $s[nivel]:=-1$ 
 $nivel:=nivel-1$ 
```

El backtracking se considera muchas veces un método recursivo, mientras que el esquema anterior es no recursivo. El esquema recursivo correspondiente sería:

Algoritmo 12.2 Esquema de backtracking recursivo, cuando se sabe que hay solución y solo se quiere encontrar una.

```

operación Backtracking-r-1sol(var  $s:\text{array}[1..\text{Max\_nivel}]$  de tipo;  $nivel:\text{entero}$ ;
                                var  $fin:\text{booleano}$ )
    si solucion( $nivel,s$ ) entonces
         $fin:=\text{verdadero}$ 
    sino si criterio( $nivel,s$ ) entonces
         $nivel:=nivel+1$ 
        mientras NO  $fin$  Y mashermanos( $nivel,s$ ) hacer
            generar( $nivel,s$ )
            Backtracking-r-1sol( $s,nivel,fin$ )
        finmientras
    finsi
```

donde inicialmente se hace la llamada Backtracking-r-1sol($s,0,fin$), con $fin=\text{falso}$. No aparece función retroceder porque al volver del backtracking regresamos al nivel anterior, ya que el parámetro $nivel$ no es variable. Los valores de s no deben quedar variados al volver de una llamada recursiva, o habría que poner $s[nivel]$ al valor inicial al bajar de nivel (incluir $s[nivel]:=-1$ tras $nivel:=nivel+1$).

Basándose en este esquema inicial, y con pequeñas modificaciones, se pueden diseñar esquemas para los demás tipos de problemas considerados.

Si queremos una única solución pero puede no haberla, la condición de fin tiene que ser que lleguemos a solución o que volvamos al nodo raíz ($nivel=0$), con lo que no se ha encontrado solución: Además, al retroceder tenemos que comprobar que no retrocedemos hasta el raíz:

Algoritmo 12.3 Esquema de backtracking no recursivo, cuando se busca una única solución pero no se sabe si existe.

```

operación Backtracking-nr-1possol(var s:array[1..Max_nivel] de tipo)
    nivel:=1
    fin:=falso
    repetir
        s[nivel]:=generar(nivel,s)
        si solucion(nivel,s) entonces
            fin:=verdadero
        sino si criterio(nivel,s) entonces
            nivel:=nivel+1
        sino
            mientras nivel≠0 Y NO mashermanos(nivel,s)
                retroceder(nivel,s)
            finmientras
        finsi
    hasta fin=verdadero O nivel=0

```

Si se buscan todas las soluciones (puede no haberlas), la condición de fin es que volvamos al nodo raíz. Cada solución que se encuentra hay que tratarla (almacenarla, escribirla por pantalla, ...), y cuando se encuentra solución también hay que comprobar si se cumple el criterio o hay más hermanos, para seguir explorando el árbol:

Algoritmo 12.4 Esquema de backtracking no recursivo, cuando se buscan todas las soluciones.

```

operación Backtracking-nr-todsol:
    nivel:=1
    fin:=falso
    repetir
        s[nivel]:=generar(nivel,s)
        si solucion(nivel,s) entonces
            tratar(s)
        finsi
        si criterio(nivel,s) entonces
            nivel:=nivel+1
        sino
            mientras nivel≠0 Y NO mashermanos(nivel,s) hacer
                retroceder(nivel,s)
            finmientras
        finsi
    hasta nivel=0

```

Un problema de optimización coincide con el caso anterior, pues se quiere encontrar todas las soluciones y de ellas quedarnos con la óptima. La única diferencia es que se mantendrán dos variables auxiliares *VOA* y *SOA* que tendrán el valor y la solución óptima hasta ese momento, y tratar la solución consiste en comparar el valor correspondiente a *s* con *VOA* y si se mejora actualizar *VOA* y *SOA*.

12.1.4. Análisis de algoritmos de backtracing

El tiempo de ejecución depende del número de nodos generados y del tiempo requerido para cada nodo, que viene dado por el coste de las funciones.

El número de nodos depende de la forma del árbol: un árbol binario de n niveles (sin incluir el raíz) tiene $2^{n+1} - 1$ nodos, y uno m -ario $\frac{m^{n+1}-1}{m-1}$, mientras que un árbol combinatorio tiene 2^n , y uno permutacional tiene $n!$ nodos terminales.

En general, suponiendo que una solución sea de la forma: (x_1, x_2, \dots, x_n) , en el peor caso se generarán todas las posibles combinaciones para cada x_i . Si el número de posibles valores para cada x_i es m_i , entonces se generan: m_1 nodos en el nivel 1, $m_1 m_2$ nodos en el nivel 2, y en cada nivel i , $m_1 m_2 \dots m_i$ nodos.

Tendremos tiempos con órdenes de complejidad factoriales o exponenciales.

Se pueden utilizar algunas técnicas para mejorar estos tiempos:

- Utilizar variables auxiliares que nos permitan programar las funciones del esquema con tiempo constante. Ejemplo 12.3, donde se utiliza una variable *suma*. Esto implica un mayor tiempo en la inicialización, pues hay que inicializar estas variables, y en algunas de las funciones que trabajan con las variables, pero puede permitir mantener el tiempo de las funciones constante.
- Utilización de una función criterio mejorada, aunque sea a costa de aumentar el coste de esta función, pues puede permitirnos eliminar nodos y quedarnos lejos de recorrer el número máximo de nodos del árbol. En el ejemplo 12.3 usamos la variable *suma* para evitar seguir por nodos en los que ya hemos pasado del número a obtener. La función criterio se puede mejorar llevando otra variable que tiene la suma de los números que quedan por tratar, y si la suma que llevamos más ese valor no llega a 20 tampoco es necesario que sigamos por ese nodo.
- Se puede recorrer el árbol no utilizando un orden fijo de generación de los nodos, sino generarlos en el orden que en cada momento parece más apropiado para resolver el problema con el que estamos. Sería equivalente a hacer un avance rápido desde cada nodo pero con la posibilidad de retroceder, y cuando se vuelve a un nodo hacer avance rápido pero sin considerar las posibilidades descartadas.

12.2. Problema de las reinas

12.2.1. Planteamiento

El problema de las n reinas consiste en encontrar las posiciones (todas o una) de n reinas en un damero de tamaño $n \times n$, sin que una reina pueda comerse a otra.

El problema se puede resolver de varias maneras:

- Se puede decidir en qué casilla poner cada una de las reinas. La solución es un array de n pares que indican las casillas donde se ponen. Esta solución corresponde a un árbol n^2 -ario de n niveles, donde en cada nivel se decide la casilla donde se pone una reina. Se puede resolver por backtracking usando este árbol, pero es impracticable

incluso para tamaños pequeños. Por ejemplo, si $n = 8$ hay $64^8 = 281.474.976.710.656$ nodos terminales, y si suponemos que se tarda $1 \mu s$ en evaluar cada nodo terminal, se tardaría unos 9 años en obtener todas las configuraciones.

- Otra posibilidad es generar configuraciones que no repiten dos reinas en la misma casilla. Tendríamos un árbol permutacional no completo, con n^2 posibles decisiones y n niveles, o el mismo árbol anterior pero con una función criterio que impida continuar por configuraciones con dos reinas en la misma casilla. El número de nodos terminales es $\frac{n^2!}{(n^2-n)!}$. Si $n = 8$, tendríamos $178.462.987.637.760$ nodos terminales, y si se tarda $1 \mu s$ en evaluar cada nodo terminal, el tiempo necesario es de algo menos de 6 años.
- Otra posibilidad es cambiar de tipo de árbol. Dado que dos reinas no pueden estar en la misma fila, y que tenemos n reinas y n filas, forzosamente tiene que haber una y solo una reina en cada fila. La solución puede consistir en un array de n valores entre 1 y n , que indican la columna donde se pone la reina de la fila correspondiente. El árbol solución será un árbol n -ario de n niveles. Si $n = 8$, el número de nodos terminales es $8^8 = 16.777.216$, que a $1 \mu s$ por nodo terminal representa aproximadamente 17 segundos. Vemos que cambiando el árbol hemos reducido sustancialmente el tiempo de ejecución.
- Se puede evitar poner dos reinas en la misma columna. Esto se consigue con un árbol permutacional de profundidad n , o con el árbol anterior pero evitando continuar por configuraciones con dos reinas en la misma columna. El número de nodos terminales es $n!$. Con $n = 8$ tenemos 40.320 nodos terminales, y a $1 \mu s$ por nodo terminal unas 4 centésimas de segundo. En algunos casos se puede reducir el tiempo teniendo en cuenta las características del problema. En este problema sólo es necesario poner en la primera fila la reina en la primera mitad de las columnas, pues todas las configuraciones válidas con la reina de la primera fila en la segunda mitad de las columnas se pueden obtener de las anteriores por simetría respecto a la columna central. De este modo, con $n = 8$ habría 20.160 nodos terminales.

12.2.2. Soluciones por backtracking

El problema se puede resolver por backtracking por cualquiera de los métodos anteriores, aunque ya vemos que hay que ser cuidadoso en el tipo de árbol y en la función criterio que se utiliza. En todos los casos el coste es prohibitivo (exponencial o factorial), aunque en unos casos más que en otros. 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. Una mejora se consigue mejorando la función criterio de manera que se evite continuar por configuraciones con dos reinas que se coman en diagonal.

Ejemplo 12.4 Consideramos el caso $n = 4$. En la figura 12.4 representamos el árbol de búsqueda. 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.

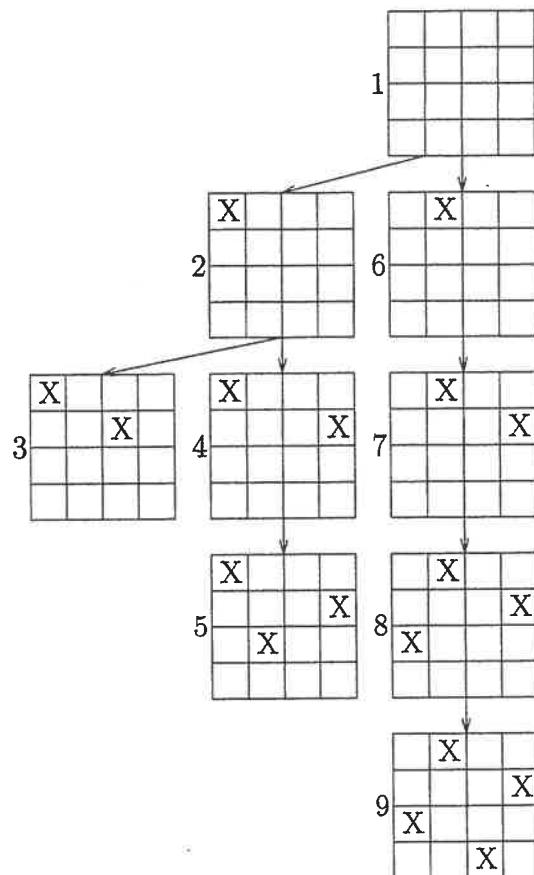


Figura 12.4: Árbol para el problema de las 4 reinas.

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 quiere encontrar todas las posibles configuraciones, podemos considerar que hay configuraciones simétricas, con lo que en la primera fila solo es necesario situar reinas en las columnas 1 y 2, pero no en las 3 y 4 pues, por cada solución que encontraremos 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 pueden 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 columna 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 solo es necesario generar 8, 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 con bajo coste computacional, de manera que no sea mayor el coste añadido en cada nodo que lo que ahorraremos al recorrer menos nodos.

- En el backtracking se hace un recorrido sistemático, pero si tuviéramos 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 seis antes que el dos llegamos a una solución recorriendo 5 nodos y no 9. Esta idea se utilizará en los métodos de ramificación y poda que veremos en el capítulo siguiente, pero se puede usar también para guiar la búsqueda en backtracking.

Diseñaremos un algoritmo suponiendo que buscamos una única solución y que sabemos que existe. Como el esquema es el del algoritmo 12.1, solo hay que decidir las variables que se utilizan, cómo se inicializan, y programar las funciones del esquema.

Una solución s consistirá en una secuencia de valores de 1 a n , indicando $s[i]=j$ que la reina en la fila i ocupa la columna j . s está inicializado a cero, para generar sumando uno al valor que tenga. Además se usa la variable *nivel* para indicar el nivel por el que vamos, y se inicializa a 1. El contenido de estas dos variables determina por qué nodo del árbol lógico vamos.

Solo son posibles nodos solución los terminales, pero además tienen que cumplir que la configuración que se consigue al poner la última reina sea válida:

operación solucion(*nivel,s*):

devolver *nivel=n* Y correcto(*nivel,s*)

Un nodo cumple el criterio cuando se puede seguir por él (no terminal) y la configuración es correcta:

operación criterio(*nivel,s*):

devolver *nivel≠n* Y correcto(*nivel,s*)

Un nodo tiene más hermanos cuando no es el último de los n hijos, y si estamos en el primer nivel si no estamos en la columna central:

operación mashermanos(*nivel,s*):

si *nivel=1* **entonces**

devolver $s[nivel] \leq n/2 + 1$

sino

devolver $s[nivel] \neq n$

finsi

Siempre se genera sumando uno a la columna en que está la reina, ya que s está inicializado a cero:

operación generar(*nivel,s*):

$s[nivel]:=s[nivel]+1$

Antes de retroceder se vuelve a poner s a su valor inicial:

operación retroceder(*nivel,s*):

```
s[nivel]:=0
nivel:=nivel-1
```

En la función 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 o si está en la misma diagonal o en la misma antidiagonal:

operación correcto(*nivel,s*):

```
para i:=1,...,nivel-1 hacer
    si s[nivel]=s[i] O s[i]-s[nivel]=i-nivel O s[i]-s[nivel]=-i+nivel entonces
        devolver falso
    finsi
finpara
devolver verdadero
```

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. Se pueden utilizar arrays de enteros indicando el número de reinas en cada columna, diagonal o antidiagonal. Los arrays serán: *c:array[1..n]*, para las columnas; *d:array[-n+1..n-1]*, para las diagonales, que se obtienen restando a la columna donde está la reina su fila; y *a:array[2..2n]*, para las antidiagonales, que se obtienen sumando la fila y columna que ocupa la reina. Los tres arrays se inicializan a cero. Usando estas variables el coste será constante, pues solo habría que comprobar que la columna, diagonal y antidiagonal donde estamos introduciendo la reina queda con una única reina. Si hay dos reinas la función correcto dirá que la configuración no es válida. Esta mejora en la función correcto conlleva una modificación de las funciones de generación y de retroceso, pues en cada caso hay que actualizar los tres arrays, pero ambas siguen teniendo coste constante.

Al generar, si se está generando el hermano de un nodo, hay que deshacer las decisiones que se hicieron para ese nodo (quitar la reina de esa columna, diagonal y antidiagonal):

operación generar(*nivel,s,c,d,a*):

```
si s[nivel]≠0 entonces
    c[s[nivel]]:=c[s[nivel]]-1
    d[s[nivel]-nivel]:=d[s[nivel]-nivel]-1
    a[s[nivel]+nivel]:=a[s[nivel]+nivel]-1
finsi
s[nivel]:=s[nivel]+1
c[s[nivel]]:=c[s[nivel]]+1
d[s[nivel]-nivel]:=d[s[nivel]-nivel]+1
a[s[nivel]+nivel]:=a[s[nivel]+nivel]+1
```

Al retroceder se actualizan las variables:

operación retroceder(*nivel,s,c,d,a*):

```
c[s[nivel]]:=c[s[nivel]]-1
```

```

d[s[nivel]-nivel]:=d[s[nivel]-nivel]-1
a[s[nivel]+nivel]:=a[s[nivel]+nivel]-1
s[nivel]:=-1
nivel:=nivel-1

```

Y la función correcto queda de coste constante:

operación correcto(*nivel,s,c,d,a*):

```
devolver c[s[nivel]]≠2 Y d[s[nivel]-nivel]≠2 Y a[s[nivel]+nivel]≠2
```

Con estas funciones se están generando nodos que corresponden a configuraciones con dos reinas en la misma columna, diagonal o antidiagonal, para después comprobar que la configuración no es correcta y deshacer la asignación anterior. Se pueden programar las funciones de manera que esto se evite. Por ejemplo, se pueden generar solo configuraciones correctas, con lo que la función solucion solo comprueba si es un nodo terminal, y criterio si no lo es. Un cambio de este tipo puede implicar que las otras funciones sean más difíciles de programar, que haya que combinar varias funciones en una, o que sea conveniente modificar el esquema. Por ejemplo, se puede obtener un programa como el siguiente, en el que se obtienen todas las soluciones y se ha cambiado el esquema y se combinan algunas de las funciones, que se ponen en el código:

operación NReinasBacktracking(**var** *s:array[1..n]* de entero)

```

nivel:=1
s[1]:=0
repetir
    s[nivel]:=s[nivel]+1 //generar
    mientras s[nivel]≤n Y NO correcto(nivel,s) entonces //no criterio
        s[nivel]:=s[nivel]+1 //genera hermano
    finmientras
    si nivel=n Y s[nivel]≤n entonces //solucion
        tratar(s)
    sino si s[nivel]≤n entonces //criterio
        nivel:=nivel+1
        s[nivel]:=0 //genera primer hijo
    sino
        nivel:=nivel-1
    finsi
hasta nivel=0

```

12.2.3. Estudio

La evaluación de un algoritmo de backtracking es compleja, ya que el tiempo de ejecución en cada nodo no es constante y el número de nodos generado es difícil de predecir. Lo que sí se puede hacer es obtener el coste de cada una de las funciones y el número máximo de nodos del árbol.

Todas las funciones anteriores tienen tiempo constante, menos la criterio inicial, que tiene $O(nivel)$, pero que se puede hacer constante aumentando algo el coste de otras funciones, aunque manteniéndolo constante.

Para el número de nodos, tenemos el nodo raíz, $\frac{n}{2}$ nodos en el primer nivel, $\frac{n}{2}(n-1)$

en el segundo, y en general $\frac{n}{2} \frac{(n-1)!}{(n-i)!}$ en el i -ésimo, con lo que el número de nodos es $\frac{n}{2} \sum_{i=1}^n \frac{(n-1)!}{(n-i)!} + 1$. No estamos contando así nada más que los nodos válidos, pero por algunos de los nodos no válidos se hace también trabajo para comprobar que no son válidos y no se generan, o para generarlos y después deshacer la asignación.

Se puede hacer una **estimación estadística** del número de nodos que se generan. Para esto se toman sucesiones que pueden ser posible solución y se obtiene cuántos nodos se generan con esta expresión, y se hace la media de esos valores.

Ejemplo 12.5 Con 4 reinas podemos considerar que se intenta 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. Este valor está muy alejado de los 9 nodos que se generan, pero estamos considerando el árbol completo y que se generan nodos por los que la función criterio descarta continuar.

12.3. Problema de la mochila

12.3.1. Planteamiento

Volvemos a considerar el problema de la mochila 0/1:

$$\begin{aligned} & \text{maximizar} && \sum_{i=1}^n b_i x_i \\ & \text{sujeto a} && \sum_{i=1}^n p_i x_i \leq M \\ & && x_i = 0, 1 \quad \forall i = 1, 2, \dots, n \end{aligned}$$

donde M es la capacidad de la mochila, cada objeto se mete entero o no se mete ($x_i = 0, 1$), y b_i y p_i son los beneficios y los pesos, respectivamente, de los objetos.

Para obtener un algoritmo por backtracking para resolver este problema hay que empezar viendo sus características y el esquema a que corresponde. Al ser un problema de optimización (maximización) tenemos que recorrer todo el árbol para obtener todas las soluciones y quedarnos con la óptima. El esquema es el del algoritmo 12.4, pero utilizando las variables auxiliares VOA y SOA para guardar la solución óptima actual y su valor, y donde tratar una solución consiste en comparar su valor con VOA y, si es mayor, sustituir VOA y SOA . La condición de fin es que *nivel* valga cero.

Veremos varias soluciones por backtracking de este problema, indicando cómo se representan las soluciones y el código de las funciones `solucion`, `criterio`, `mashermanos`, `generar` y `retroceder`.

12.3.2. Solución con árbol binario

Si utilizamos un árbol binario de soluciones, donde en cada nivel i probamos la posibilidad de incluir o no el objeto i , una solución será de la forma (x_1, x_2, \dots, x_n) , con $x_i = 0, 1$. El array de soluciones será $s:\text{array}[1..n]$ de $-1, 0, 1$, y estará inicializado a -1 para que al ir sumando uno en la función generar se obtengan los valores válidos.

Para evitar cálculos en algunas de las funciones del esquema, utilizaremos las variables b_act y p_act que guardarán el beneficio y el peso acumulado en cada nodo.

Son nodos solución los terminales cuyo peso no excede la capacidad de la mochila:

operación *solucion(nivel,s,p_act)*:

devolver *nivel=n Y p_act≤M*

Un nodo cumple el criterio cuando no es terminal y no excede la capacidad de la mochila:

operación *criterio(nivel,s,p_act)*:

devolver *nivel≠n Y p_act≤M*

Tienen más hermanos los nodos con valor cero:

operación *mashermanos(nivel,s)*:

devolver *s[nivel]=0*

Se genera sumando uno, y si se genera un nodo con valor uno hay que aumentar el peso y el beneficio:

operación *generar(nivel,s,p_act,b_act)*:

si *s[nivel]=0 entonces*

b_act:=b_act+b[nivel]

p_act:=p_act+p[nivel]

finsi

s[nivel]:=s[nivel]+1

Antes de retroceder se vuelve a poner *s* a su valor inicial, y se actualizan el peso y el beneficio:

operación *retroceder(nivel,s,p_act,b_act)*:

b_act:=b_act-b[nivel]

p_act:=p_act-p[nivel]

s[nivel]:=-1

nivel:=nivel-1

Y si el nodo actual representa una solución mejor que la óptima actual se actualiza:

operación *tratar(nivel,s,b_act,VOA,SOA)*:

si *b_act>VOA entonces*

VOA:=b_act

SOA←s

finsi

El coste depende del número de nodos del árbol de soluciones, que es $2^{n+1} - 1$, y del coste de las funciones, que son todas constantes. Por tanto el orden es $O(2^n)$. Pero la función criterio puede evitarnos recorrer algunos nodos y hacer que el número de nodos que se recorre sea mucho menor que esa cota superior.

Ejemplo 12.6 Consideramos el problema con $n = 4$, $M = 7$, $b = (2, 3, 4, 5)$ y $p = (1, 2, 3, 4)$. El árbol de soluciones consta de 31 nodos, y se recorren 31 porque la función

criterio solo eliminaría alguno de los nodos terminales tras generarlos.

Esto quiere decir que se debe intentar mejorar la función criterio.

12.3.3. Mejora de la función criterio

Se puede intentar eliminar más nodos del árbol de soluciones con un criterio más restrictivo. Para cada nodo, se puede obtener una cota máxima del beneficio que se puede obtener a partir de él. Si esta cota es menor que el beneficio de la solución óptima actual, se rechaza ese nodo y sus descendientes. Una cota se puede obtener sumando al beneficio actual el resultado de resolver por avance rápido el problema no 0/1, para lo que se realizará un preprocessamiento consistente en ordenar los objetos de mayor a menor $\frac{b}{p}$. El avance rápido se hará con la capacidad de la mochila que queda por llenar y los objetos que todavía no se han tratado, y como los datos del problema son enteros se puede tomar el valor entero del resultado: $b_act + \lfloor \text{mochilano0/1}(nive+1, M-p_act) \rfloor$. De esta manera la función criterio queda:

operación criterio(*nivel, s, p_act, b_act*):

devolver *nivel* ≠ *n* Y *p_act* ≤ *M* Y *b_act* + $\lfloor \text{mochilano0/1}(nive+1, M-p_act) \rfloor > VOA$

Se pueden eliminar nodos a costa de aumentar el tiempo de ejecución de la función criterio. El coste de esta función es $O(n - nivel)$, por lo que se puede pensar que el orden del algoritmo se obtenga multiplicando los órdenes de la función y del número de nodos, $O(n2^n)$. Calculando el coste teniendo en cuenta el nivel en que está cada nodo se tiene $t(n) = \sum_{i=1}^n 2^i(n-i) = n\sum_{i=1}^n 2^i - \sum_{i=1}^n i2^i \in o((2 - \frac{1}{\ln 2})n2^n)$. No sabemos cuántos nodos se van a podar, por lo que no sabemos si este nuevo criterio nos producirá mejoras en el tiempo de ejecución pero, en general, cuando el tamaño del problema crece la eliminación de un nodo repercute en la eliminación de una amplia parte del árbol, por lo que suele ser preferible utilizar criterios más restrictivos aunque sean más costosos de calcular.

Ejemplo 12.7 Con los datos del ejemplo 12.6, el árbol es el de la figura 12.5. Los números dentro de los nodos indican el orden en que se recorren; los valores de *p* y *b* 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. 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 (*p*), no se ha podado ningún nodo por esta razón. Los nodos terminales son soluciones, por lo que solo 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. Vemos que para eliminar más nodos hay que comprobar si se descarta tratar los hijos de un nodo no solo cuando se genera, sino

también cuando se vuelve a él al retroceder, con lo que podría ser conveniente calcular la cota del beneficio cada vez que se llega a un nodo, con generar o retroceder.

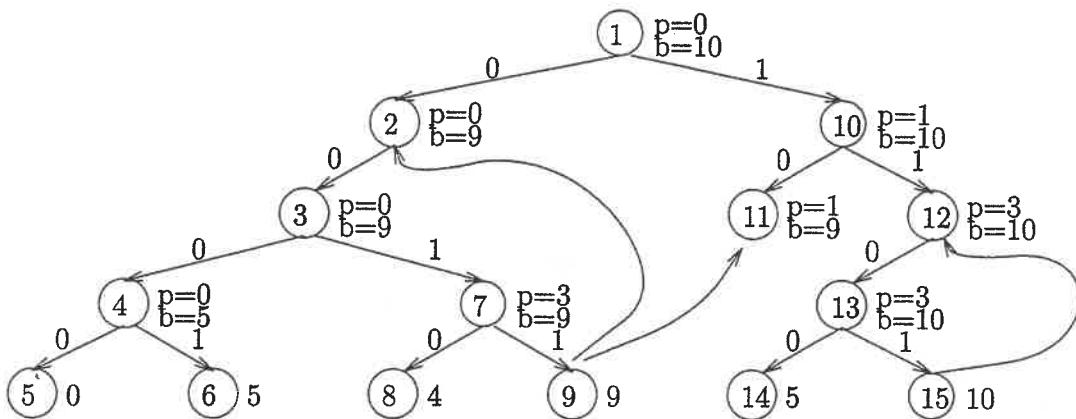


Figura 12.5: Árbol para el problema de la mochila.

Se han generado un total de 15 nodos, mientras que antes generábamos 31. Vemos, por tanto, que el uso de una buena función criterio puede reducir considerablemente el número de nodos generados.

12.3.4. Modificación del orden de recorrido

Otra posible mejora consiste en determinar cuál parece el mejor orden de recorrido del árbol. Ya que nuestro problema es un caso particular del no 0/1 y, en ese caso, los objetos se ordenan de mayor a menor $\frac{b}{p}$, y se van insertando en la mochila los primeros objetos, parece lógico que el empezar metiendo objetos en la mochila lleve a soluciones más cercanas a la óptima y permita eliminar más nodos.

Para trabajar con un árbol binario generando los nodos en el orden 1-0 hay que cambiar la inicialización del array solución, que estará inicializado a dos; y las funciones *mashermanos*, *generar* y *retroceder* quedan:

```

operación mashermanos(nivel,s):
    devolver s[nivel]≠0
operación generar(nivel,s,p_act,b_act):
    si s[nivel]=2 entonces
        b_act:=b_act+b[nivel]
        p_act:=p_act+p[nivel]
    sino si s[nivel]=1 entonces
        b_act:=b_act-b[nivel]
        p_act:=p_act-p[nivel]
    finsi
    s[nivel]:=s[nivel]-1
operación retroceder(nivel,s,p_act,b_act):

```

```
s[nivel]:=2
nivel:=nivel-1
```

Respecto al coste, tenemos el mismo orden del caso anterior, aunque esperamos que se generen menos nodos por la combinación de una buena función criterio con una buena estrategia de recorrido del árbol.

Ejemplo 12.8 Consideramos de nuevo el problema del ejemplo 12.6. El árbol de búsqueda resultante se muestra en la figura 12.6. 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.

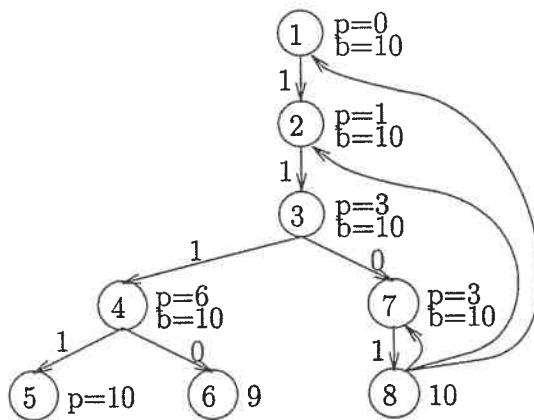


Figura 12.6: Árbol para el problema de la mochila, con recorrido en orden 1-0.

12.3.5. Solución con árbol combinatorio

Ya que el orden en que se incluyen los objetos en la mochila no influye en la solución final, se puede utilizar un árbol combinatorio para resolver el problema, y como estos árboles tienen aproximadamente la mitad de nodos que el árbol binario equivalente, se puede esperar que el tiempo de ejecución se reduzca a la mitad.

Podemos considerar que una solución viene dada por un array s de 1 a n y valores entre 0 y n , indicando $s[j]=j$ que el objeto j se mete en la mochila, y $s[j]=0$ que no se mete ningún objeto.

En este caso todos los nodos son posible solución, por lo que solo hay que comprobar que el peso no excede la capacidad de la mochila:

operación $\text{solucion}(nive, s, p_act)$:

devolver $p_act \leq M$

Un nodo cumple el criterio cuando no excede la capacidad de la mochila y la cota del beneficio supera la óptima actual:

operación $\text{criterio}(nive, s, p_act)$:

devolver $p_act < M \text{ Y } b_act + \lfloor \text{mochilano0/1}(s[nivel]+1, M-p_act) \rfloor > VOA$

No tienen más hermanos los nodos que corresponden al último objeto:

operación *mashermanos(nivel,s)*:

devolver $s[nivel] \neq n$

Se genera sumando uno, salvo en los primeros hijos de nodos a partir del raíz:

operación *generar(nivel,s,p_act,b_act)*:

si $s[nivel] \neq 0$ **entonces**

$b_act := b_act - b[s[nivel]]$

$p_act := p_act - p[s[nivel]]$

finsi

si $nivel \neq 1 \text{ Y } s[nivel] = 0$ **entonces**

$s[nivel] := s[nivel-1] + 1$

sino

$s[nivel] := s[nivel] + 1$

finsi

$b_act := b_act + b[s[nivel]]$

$p_act := p_act + p[s[nivel]]$

Antes de retroceder se vuelve a poner *s* a su valor inicial, y se actualizan el peso y el beneficio:

operación *retroceder(nivel,s,p_act,b_act)*:

$b_act := b_act - b[s[nivel]]$

$p_act := p_act - p[s[nivel]]$

$s[nivel] := 0$

$nivel := nivel - 1$

El número de nodos del árbol es 2^n , y el coste de todas las funciones es constante, salvo la criterio, que tiene coste lineal. Podríamos esperar que un algoritmo que utilizara un árbol de este tipo fuera el doble de rápido que uno que usara un árbol binario, pero no tiene por qué ser así, tal como vemos en el siguiente ejemplo, ya que, más que la forma del árbol, puede ser determinante la función criterio y la estrategia de recorrido del árbol, y en un árbol binario eliminar los descendientes de un nodo puede ser más productivo que en uno combinatorio, dependiendo de en qué nodo estemos.

Ejemplo 12.9 Con el problema del ejemplo 12.6, queda el árbol de la figura 12.7. Al lado de cada nodo que representa una solución válida se representa también el valor que le corresponde. Utilizando un árbol combinatorio se generan 6 nodos en vez de 8 que se generaban con el árbol binario. Vemos que no hemos reducido a la mitad como era de esperar.

Ejercicios resueltos

Ejercicio 12.1 Se quiere hacer un programa por Backtracking que resuelva un rompecabezas consistente en llenar 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, mashermanos y retroceder.

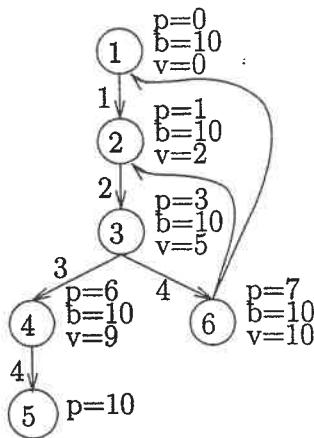
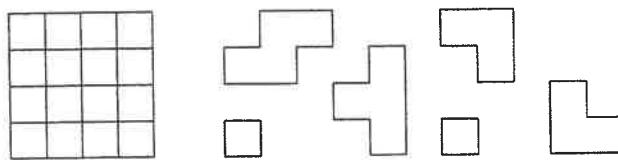


Figura 12.7: Árbol combinatorio para el problema de la mochila.

**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) el de la figura 12.8.

Tendremos un contador que indique por qué nivel vamos (*nivel*) y un array (*nodo:array[1..nivel] de 0..65*) que indicará, por cada nivel del árbol, por cuál de los 65 nodos se va generando.

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

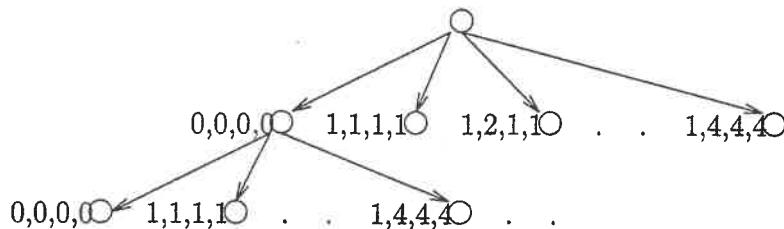


Figura 12.8: Árbol del rompecabezas.

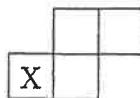
en forma de cuádrupla, que es lo que necesitamos para poder almacenar la solución:

```

operación generar(nivel,nodo):
    nodo:=nodo+1
    si nodo=0 entonces
        solucion[nivel] :=(0,0,0,0)
    sino
        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) mod 4+1
    finsi
```

La función criterio comprobará si hay más hermanos que generar, por lo que será verdadero 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:=verdadero
para i:=0,...,número de cuadrados hacer
    si NO encaja(i,pieza) entonces
        criterio:=falso
```

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 verdadero, si consideramos que con las piezas que nos dan se va a poder construir el rompecabezas, o también que `nivel` sea 0 si no tiene por qué poderse construir el rompecabezas.

Ejercicio 12.2 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 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 el de la figura 12.9. En el nivel i -ésimo se decide el trabajo a realizar por el trabajador i -ésimo.

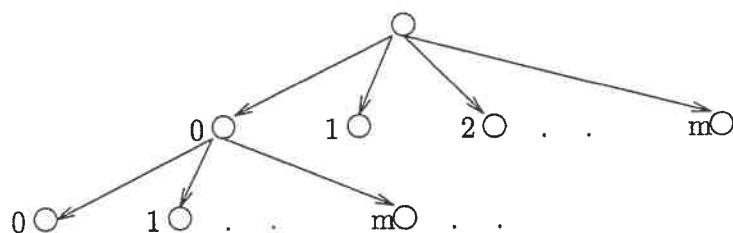


Figura 12.9: Árbol del problema de asignación de trabajos.

Las funciones del esquema del backtracking serán:

operación generar(`solucion,nivel`):

devolver `solucion[nivel]+1`

para lo que necesitaremos (para empezar por 0) que `solucion` esté inicializado a -1.

operación mashermanos(`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.

operación criterio(*solución,nivel*):

devolver *tabla[nivel,solución[nivel]]*=1 Y *asignado[solución[nivel]]*=falso

(si no utilizáramos *asignado* tendríamos que recorrer el array *solución* hasta el nivel actual para ver si se ha asignado el trabajo).

operación solución(*solución,nivel*):

devolver correcto(*solución,nivel*) Y *nivel*=*n*

La condición de fin en este caso sería que una variable booleana *fin* sea verdadero (tomará este valor cuando *solución(solución,nivel)*=verdadero) o *nivel*=0 (en cuyo caso no habría solución para este problema).

Ejercicio 12.3 Resolver el problema del grafo multietapa por medio de un backtracking: decir cómo sería el árbol, cómo se representan las soluciones, cuál es la condición de fin, cómo serían las funciones, 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 (número de nodos del siguiente nivel) hijos válidos, y los nodos de nivel $m - 1$ un máximo de uno (con m el número de niveles del grafo).

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 pueda 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
SOA← 0 //array donde se almacena la solución óptima actual
s← 0
s[1]:=1
VOA=  $\infty$  //valor de la solución óptima actual
valor:= 0 //contendrá el valor de la solución que estamos formando
repetir
  si solución(nivel) entonces
    si valor<VOA entonces
      SOA←s
      VOA:=valor
    finsi
  finsi
  si criterio(nivel) entonces
    nivel:=nivel+1
    s[nivel]:=generar(nivel)
  sino

```

```

mientras nivel $\neq$  1 Y NO mashermanos(nivel) hacer
    retroceder(nivel)
finmientras
si nivel $\neq$  1 entonces
    s[nivel]:=generar(nivel)
finsi
finsi
finsi
hasta nivel= 1
donde las funciones son:
operación solucion(nivel):
devolver nivel=m
    //hemos llegado al último nivel
operación criterio(nivel):
devolver valor<VOA
    //no hemos sobrepasado el VOA y podemos mejorar la solución óptima actual
operación generar(nivel):
    valor:=valor+t[s[nivel-1],s[nivel]+1]
devolver s[nivel]+1
    //genera los hijos independientemente de que exista la arista o no
operación mashermanos(nivel):
devolver s[nivel]<m
operación retroceder(nivel):
    valor:=valor-t[s[nivel-1],s[nivel]]
    s[nivel]:=0
    nivel:=nivel-1

```

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 $m + 1$ 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.

Ejercicio 12.4 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..n] **de** enteros, y teniendo de cada tipo una cierta cantidad de monedas, estando estas cantidades almacenadas en un array *cantidad*:**array**[1..n] **de** enteros (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, qué esquema se usa y programar las funciones.

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 (figura 12.10).

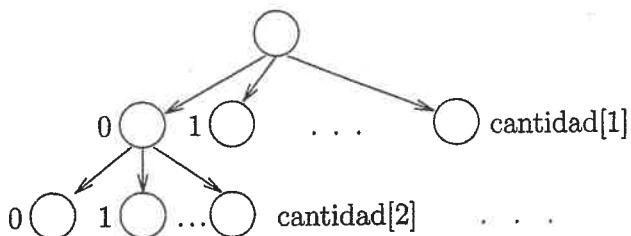


Figura 12.10: Árbol del problema de devolución de monedas.

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 solución será:

operación $solucion(nivel)$:

devolver $nivel=n \text{ Y } \sum_{i=1}^{nivel} tipos[i] * s[i] = C$

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:

operación $solucion(nivel)$:

devolver $nivel=n \text{ Y } suma=C$

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:

operación $criterio(nivel)$:

devolver $nivel \neq n \text{ Y } suma < C$

Un nodo tendrá más hermanos cuando no estemos en el último nodo de ese nivel:

operación $mashermanos(nivel)$:

devolver $s[nivel] < cantidad[nivel]$

En la función $generar$ modificaremos la variable $suma$ 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:

operación $generar(nivel)$:

```

s[nivel]:=s[nivel]+1
si s[nivel]≠0 entonces
        suma:=suma+tipos[nivel]
        monedas:=monedas+1
finsi
operación retroceder(nivel):
    monedas:=monedas-s[nivel]
    suma:=suma-s[nivel]*tipos[nivel]
    s[nivel]:=-1
    nivel:=nivel-1

```

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 empiezemos por el valor cero.

El esquema sería:

```

monedas:=0
suma:=0
SOA← -1
VOA:=∞
s← -1
nivel:=1
generar(nivel)
repetir
    si solucion(nivel) entonces
            si monedas< VOA entonces
                    VOA:=monedas
                    SOA←s
        finsi
    finsi
    si criterio(nivel) entonces
            nivel:=nivel+1
            generar(nivel)
    sino
        mientras nivel> 0 Y NO mashermanos(nivel) hacer
                        retroceder(nivel)
        finmientras
        si nivel>0 entonces
                        generar(nivel)
        finsi

```

```
finsi
hasta nivel= 0
```

Ejercicio 12.5 a) Programar un algoritmo de backtracking no recursivo, según uno de los esquemas vistos, 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.

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 el de la figura 12.11. Por motivos de claridad no aparecen los nodos que representan elecciones repetidas y por los que no se seguirá la búsqueda.

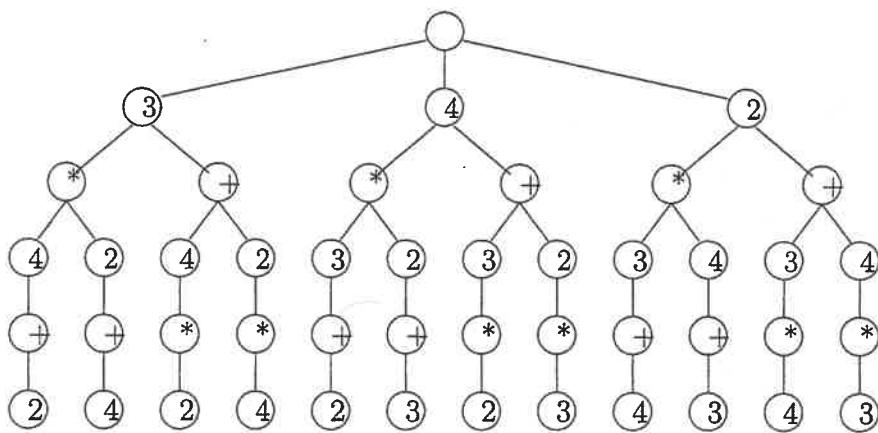


Figura 12.11: Árbol de soluciones del problema 12.5.

Dado que se buscan todas las configuraciones el esquema del algoritmo será:

nivel:= 1

generar(s,nivel)

```

repetir
    si solucion(s,nivel) entonces
        tratar(s)
    finsi
    si criterio(s,nivel) entonces
        nivel++
        generar(s,nivel)
    sino
        mientras nivel ≠ 0 Y NO mashermanos(s,nivel) hacer
            retroceder(s,nivel)
        finmientras
        si nivel ≠ 0 entonces
            generar(s,nivel)
        finsi
    finsi
hasta 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 :

operación solucion(*s,nivel*):

devolver *nivel*= $2n-1$ Y 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 un 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*.

operación criterio(*s,nivel*):

```

si impar(nivel) entonces
    devolver nivel≠ $2n-1$  Y operandos[s[nivel]]=1
sino
    devolver operadores[s[nivel]]=1
finsi

```

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:

operación *generar(s,nivel)*:

```

si s[nivel] ≠ 0 entonces
    si impar(nivel) entonces
        operandos[s[nivel]]-- 
    sino
        operadores[s[nivel]]-- 
    finsi
finsi
s[nivel]++
si impar(nivel) entonces
    operandos[s[nivel]]++ 
sino
    operadores[s[nivel]]++ 
finsi
```

- En *mashermanos* se comprueba si hemos evaluado hasta el operando *n* o el operador *n - 1*:

operación *mashermanos(s,nivel)*:

```

si impar(nivel) entonces
    devolver s[nivel] ≠ n
sino
    devolver s[nivel] ≠ n-1
finsi
```

- 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:

operación *retroceder(s,nivel)*:

```

si impar(nivel) entonces
```

```

operando[s[nivel]]--
sino
  operadores[s[nivel]]--
finsi
  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 commutativa, 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 modificar 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, más 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, etc. De este modo el número de nodos es:

$$\begin{aligned}
1 + n + n(n-1) + n(n-1)(n-1) + n(n-1)(n-1)(n-2) + n(n-1)(n-1)(n-2)(n-2) + \dots = \\
1 + n + n(n-1) + n(n-1)^2(n-1) + n(n-1)^2(n-2)^2(n-2) + \dots + n(n-1)^2(n-2)^2 * \dots * 2^2 * 1 \\
\geq n!(n-1)!
\end{aligned}$$

Ejercicio 12.6 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 $h = (h_1, h_2, \dots, h_n)$, siendo $h_1 + h_2 + \dots + h_n = m$. 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, e 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( $s, nivel, v$ )
repetir
    si solucion( $s, nivel$ ) entonces
        si  $v < VOA$  entonces
             $SOA \leftarrow s$ 
             $VOA := v$ 
        finsi
    finsi
    si criterio( $s, nivel, v$ ) entonces
         $nivel++$ 
        generar( $s, nivel, v$ )
    sino
        mientras  $nivel \neq 0$  Y NO hermanos( $s, nivel$ ) hacer
            retroceder( $s, nivel, v$ )
        finmientras
    finsi
    si  $nivel \neq 0$  entonces
        generar( $s, nivel, v$ )
    finsi
hasta  $nivel = 0$ 

```

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:

operación solucion($s, nivel$):
devolver $nivel = m$ Y $h[nivel] = 0$

Un nodo tendrá más hermanos si no es el correspondiente al último profesor:
hermanos(*s,nivel*):

devolver $s[nivel] < n$

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:

operación criterio(*s,nivel,v*):

devolver $nivel < m \text{ Y } h[nivel] \geq 0 \text{ Y } v < VOA$

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:

operación generar(*s,nivel,v*):

```

si  $s[nivel] \neq 0$  entonces
     $v := v - p[s[nivel], nivel]$ 
     $h[s[nivel]]++$ 
finsi
     $s[nivel]++$ 
     $v := v + p[s[nivel], nivel]$ 
     $h[s[nivel]]--$ 

```

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:

operación 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 12.12 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.

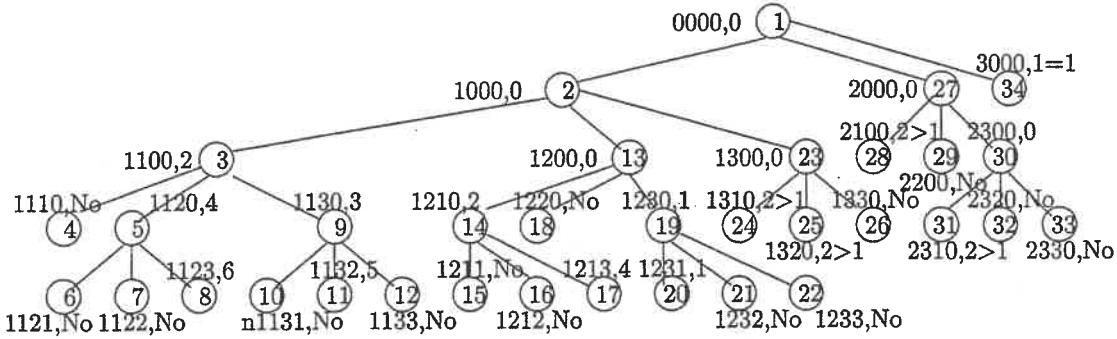


Figura 12.12: Árbol del problema de los horarios.

Ejercicios propuestos

Ejercicio 12.7 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 solo 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

Ejercicio 12.8 Diseñar un programa que encuentre los n enteros positivos x_1, x_2, \dots, x_n que, dado otro entero positivo N , minimicen $\sum_{i=1}^n x_i^2$ y cumplan que $\sum_{i=1}^n x_i = N$.

Ejercicio 12.9 Si tenemos n números enteros x_1, x_2, \dots, x_n y otro entero N , hacer un algoritmo que encuentre un subconjunto $\{y_1, y_2, \dots, y_m\}$ de $\{x_1, x_2, \dots, x_n\}$ que minimice el valor $|N - \sum_{i=1}^m y_i|$.

Ejercicio 12.10 Dado un conjunto de pares de números naturales $S = \{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$, 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 $\{(u_1, v_1), (u_2, v_2), \dots, (u_m, v_m)\}$, hay que minimizar $|\sum_{i=1}^m u_i - \sum_{i=1}^m v_i|$.

Ejercicio 12.11 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 mínima. Nota: el grado de aversión es un valor mayor o igual a 0.

Ejercicio 12.12 Dado un conjunto de números naturales $X = \{x_1, x_2, \dots, x_n\}$ 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 $N_1 \leq S \leq N_2$.

Ejercicio 12.13 Dado un conjunto de números enteros positivos ($\{x_1, x_2, \dots, x_n\}$) y otro número entero positivo N , resolver por backtracking, utilizando alguno de los esquemas no recursivos vistos, el problema de obtener un subconjunto ($\{y_1, y_2, \dots, y_m\}$) de dicho conjunto de números que minimice el valor $D = N - y_1 * y_2 * \dots * y_m$, siendo $D \geq 0$.

Ejercicio 12.14 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.

Ejercicio 12.15 Resolver el problema de la mochila 0-0.5-1, donde un objeto se incluye entero en la mochila, la mitad, o no se incluye.

Ejercicio 12.16 Resolver el problema de las reinas amazonas, consistente en situar en un tablero $n \times n$, n reinas amazonas (mueven como una reina y un caballo) sin que se puedan comer.

Cuestiones de autoevaluación

Ejercicio 12.17 A partir del problema del ejemplo 12.1, donde se sabe que hay solución y se quiere encontrar solo una, plantear enunciados para los otros tipos de problema considerados, y programar las funciones para resolver cada uno de los problemas.

Ejercicio 12.18 Dar esquemas recursivos correspondientes a los esquemas no recursivos 12.3 y 12.4

Ejercicio 12.19 Justificar si es preferible utilizar un método de programación dinámica o uno de backtracking para resolver problemas de optimización.

Ejercicio 12.20 Indicar algunas formas en que la técnica de backtracking se puede combinar con otras técnicas de diseño de algoritmos.

Referencias bibliográficas

El backtracking se considera en muchas referencias como una técnica recursiva, por lo que se estudia dentro de este apartado ([Wirth80], [Collado87], [Wirth87], [Kruse89]) junto con la técnica divide y vencerás. En otros casos se estudia con otras técnicas de recorrido de árboles y grafos ([Brassard90], [Campos95], [Brassard97]), como son la ramificación y poda y los árboles de juegos, que analizamos en los dos capítulos siguientes.

Algunos libros tienen un pequeño apartado dedicado al backtracking dentro de un capítulo de diseño de algoritmos ([Aho88], [Weis95], [Rabhi99]). En casi todos los casos se considera el problema de las reinas, normalmente con una versión recursiva. El problema de la mochila también se estudia en alguna de las referencias, y se encuentran pocos ejemplos más.

El enfoque con que se aborda la técnica en este capítulo está más cercano a [Troya84], donde se estudia el problema de las reinas, se diseñan esquemas y se presenta la idea del estudio estadístico.

Capítulo 13

Ramificación y poda

Al igual que el backtracking, la técnica de ramificación y poda (o ramificación y acotación, o branch and bound, en inglés) se basa en la resolución de problemas por búsqueda en un árbol de soluciones. Este árbol es un árbol lógico y se recorre de manera sistemática. Mientras que en el backtracking la búsqueda se realiza en profundidad, en ramificación y poda la búsqueda es guiada por estimaciones asociadas a cada nodo del árbol, y la eliminación se realiza asociando a cada nodo cotas inferior y superior de una solución óptima a partir de él. Por tanto, la ramificación y poda se aplica normalmente a problemas de optimización, aunque algunas de sus ideas se pueden usar en problemas de otro tipo, o en un recorrido por backtracking para reducir el número de nodos o decidir el orden de recorrido del árbol.

Objetivos del capítulo:

- Comprender la técnica de resolución de problemas por ramificación y poda, entendiendo la importancia de la estimación del beneficio para guiar la búsqueda, y del cálculo de las cotas para podar el árbol de búsqueda.
- Comprender la importancia de la heurística en la resolución eficiente de problemas de alto coste computacional.
- Identificar problemas que se pueden resolver por ramificación y poda y conocer esquemas para su resolución.
- Comparación de la solución de problemas que se han estudiado en otras técnicas con la forma de resolverlos por ramificación y poda. Habrá que diferenciar claramente la forma de recorrido del árbol por backtracking y por ramificación y poda.
- Comprender la utilización de métodos aproximados (como los voraces) como ayuda en la resolución de problemas por ramificación y poda.

Contenido del capítulo:

13.1. Método general	187
13.1.1. Ideas generales	187
13.1.2. Estrategias de poda	188
13.1.3. Estrategias de ramificación	188
13.1.4. Esquema general	191
13.1.5. Análisis de algoritmos de ramificación y poda	191
13.2. Problema de la mochila 0/1	192
13.2.1. Árboles de solución	192
13.2.2. Estimación del beneficio y las cotas	192
13.2.3. Estrategias de ramificación y poda	193
13.2.4. Algoritmo	194
13.3. Secuenciamiento de trabajos con plazos	196
13.3.1. Definición del problema	196
13.3.2. Diseño de una solución	197
13.4. Problema de las reinas	198
Ejercicios resueltos	204
Ejercicios propuestos	219
Cuestiones de autoevaluación	220
Referencias bibliográficas	222

13.1. Método general

13.1.1. Ideas generales

La técnica **ramificación y poda**¹ 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 haber encontrado la solución óptima.

El recorrido no tiene por qué ser en profundidad, sino que se usará una lista de nodos vivos de los que no se han generado los hijos. Los nodos del árbol se generan extrayendo un nodo de la lista de nodos vivos y generando todos sus hijos, que se incluyen después en la lista. La ejecución acaba cuando la lista de nodos vivos queda vacía.

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. Esta estimación se usa para decidir qué zonas del árbol explorar primero, generando en cada paso los hijos del nodo con mayor beneficio estimado (o de menor coste si estamos en un problema de minimización del coste) de entre los que están en la lista de nodos vivos.

Además de la estimación del beneficio, en cada nodo se calculan cotas inferior y superior de una solución óptima a partir de él. Estas cotas se utilizarán para realizar podas en el árbol, de manera similar a como se utiliza la función criterio en el backtracking.

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). La estimación del beneficio está entre las dos cotas. Además de que estos valores sean adecuados hay que tener en cuenta que su cálculo lleva 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.

De este modo, en cada nodo tendremos:

- **Cota superior (CS)** y **Cota inferior (CI)** del beneficio (o coste) que podemos alcanzar a partir de ese nodo. Determinan cuándo se puede realizar una poda.
- **Estimación del beneficio** (o coste) que se puede encontrar a partir de ese nodo. En algunos casos se obtienen a partir de las cotas: puede ser la media de las dos cotas o una de ellas. Ayuda a decidir qué parte del árbol se evalúa primero.

¹Branch and bound, en inglés.

13.1.2. Estrategias de poda

Dependiendo de las características del problema, las cotas se utilizarán de diferente manera:

- En un problema de maximización, supongamos que hemos recorrido los nodos $1, \dots, n$, estimando para cada nodo j una cota superior $CS(j)$ y otra inferior $CI(j)$, respectivamente, para $1 \leq j \leq n$. Si a partir de un nodo siempre podemos obtener alguna solución válida, se poda un nodo i si $CS(i) < CI(j)$, para algún j generado. Por ejemplo, en el problema de la mochila, si $CS(a) = 4$ y $CI(b) = 5$, se puede podar el nodo a , sin perder ninguna solución óptima. Normalmente la condición de poda, $CS(i) < CI(j)$, se puede sustituir por otra más restrictiva, $CS(i) \leq CI(j)$, que puede producir una mayor cantidad de podas. El problema con esta segunda condición es que hay que controlar que un nodo no se podes a sí mismo o a alguno de sus descendientes que puedan llevar a la solución óptima a partir de él.
- Si a partir de un nodo puede que no lleguemos a una solución válida, se poda un nodo i si $CS(i) \leq Beneficio(j)$, para algún j que sea solución. Por ejemplo, en el problema de las reinas, a partir de una solución parcial no está garantizado que exista alguna solución, por lo que sólo se puede podar utilizando valores de nodos solución.

En problemas de minimización del coste, las podas se realizan invirtiendo las desigualdades anteriores: si $CI(a) > CS(b)$ se puede podar el nodo a .

13.1.3. Estrategias de ramificación

Para recorrer el árbol se utiliza lo que se llama una lista de nodos vivos (LNV), 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 hace un recorrido “a ciegas”, que puede consistir en generar los descendientes de los nodos en el mismo orden en que han sido generados (FIFO: first in first out). Por tanto, con la estrategia FIFO la lista de nodos vivos se implementa como una cola. En la figura 13.1 vemos cómo se recorre el árbol y cómo evoluciona la lista de nodos vivos con un ejemplo pequeño. Representamos la nueva lista de nodos vivos después de la generación de los hijos de cada nodo. Se puede apreciar que la estrategia FIFO da lugar a un recorrido del árbol en anchura.

- **Estrategia LIFO.**

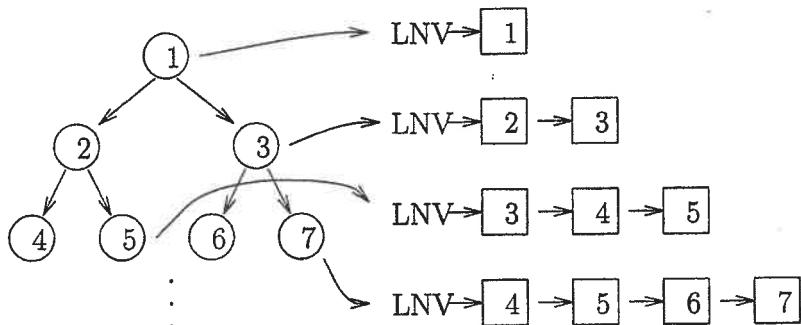


Figura 13.1: Recorrido de un árbol con estrategia FIFO.

En este caso la lista de nodos vivos se implementa como una pila (LIFO: last in first out). Se obtiene un árbol como el de la figura 13.2. En este caso, el árbol es recorrido en profundidad.

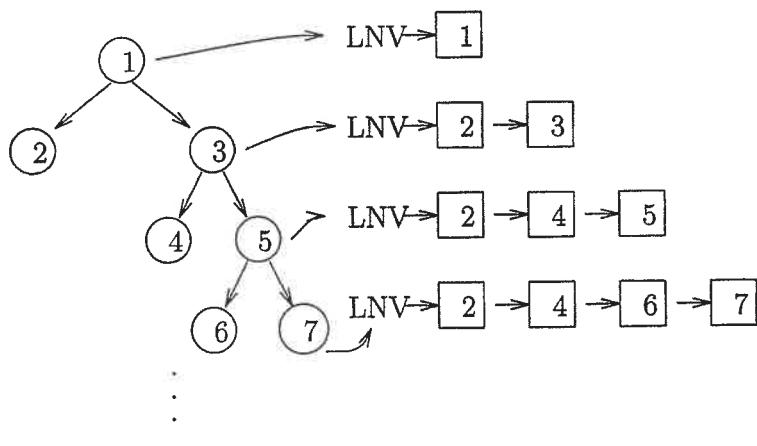


Figura 13.2: Recorrido de un árbol con estrategia LIFO.

■ 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. En la figura 13.3 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 C , global a todos los nodos, donde almacenamos la menor de las cotas superiores. Consideraremos que si $CI(i) \geq C$

podemos podar el nodo i . Estamos suponiendo que a partir de un nodo siempre hay solución.

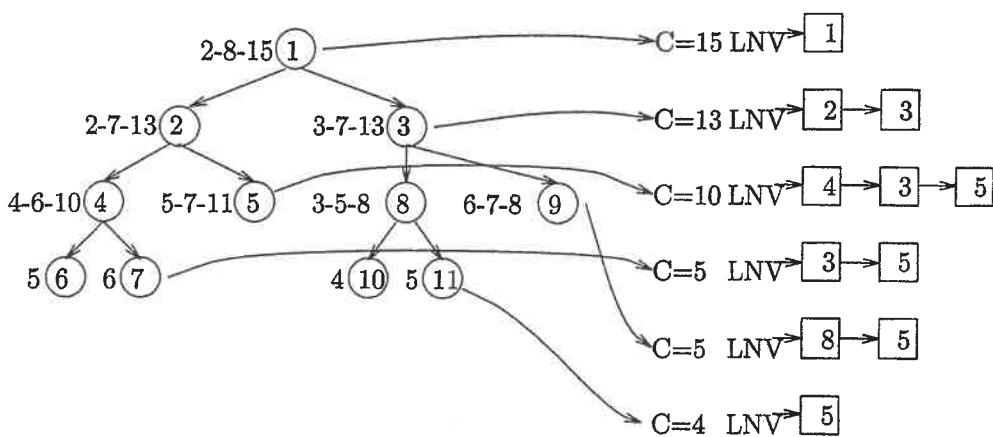


Figura 13.3: Recorrido de un árbol con estrategia LC-LIFO

En el recorrido del árbol se ve que hay nodos vivos que sólo se pueden podar tras sacarlos de la lista de nodos vivos, pues después de generarlos se ha obtenido una solución cuyo coste no pueden disminuir (nodo 5), y otros que se podan al generarlos (nodo 9). Se acaba el recorrido del árbol cuando no quedan nodos vivos.

■ Estrategia LC-LIFO.

En la figura 13.4 vemos el recorrido del árbol anterior con esta estrategia.

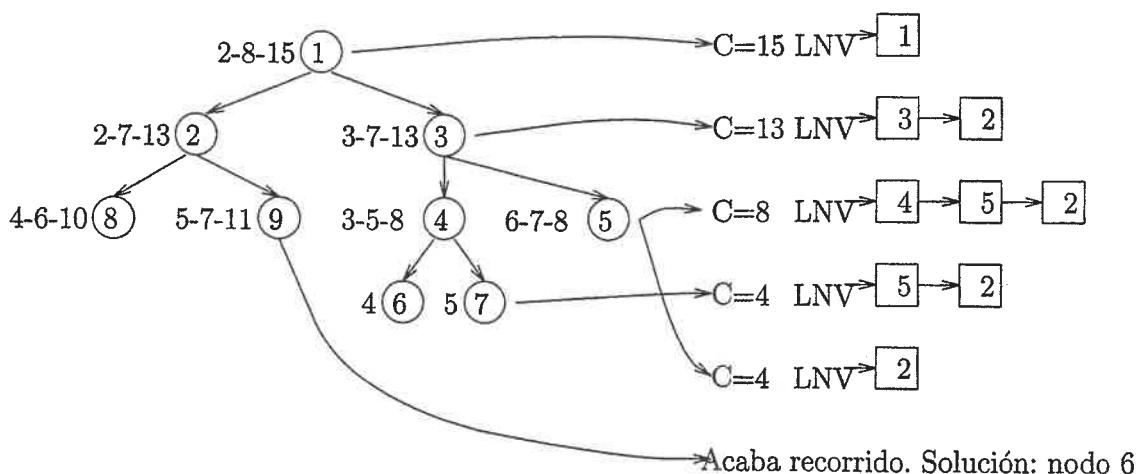


Figura 13.4: Recorrido de un árbol con estrategia LC-FIFO.

13.1.4. Esquema general

En un problema de minimización, suponiendo el caso en que existe solución a partir de cualquier nodo, los pasos a seguir para resolverlo serían:

- Tenemos la lista de nodos vivos inicializada con el nodo raíz.
- Se selecciona un nodo de la lista de nodos vivos con alguno de los criterios de ramificación anteriores.
- El nodo elegido se elimina de la lista.
- Si la cota inferior del nodo es mayor que la mínima cota superior de los nodos analizados (variable de poda C) no se generan sus hijos.
- Cuando se genera un nodo, si no se poda, se actualiza la variable C .

El esquema sería:

Algoritmo 13.1 Esquema de ramificación y poda para un problema de minimización de coste cuando existe solución a partir de cualquier nodo.

operación RamificacionYPoda(*NodoRaiz:tipo_nodo;var s:tipo_solucion*)

```

LNV:=NodoRaiz
C:=CS(NodoRaiz)
s:=∅
mientras LNV≠ ∅ hacer
    x:=Seleccionar(LNV) //Según un criterio FIFO, LIFO, LC-FIFO o LC-LIFO
    LNV:=LNV-{x}
    si CI(x)<C entonces //Si no se cumple se poda x
        para cada y hijo de x hacer
            si y es una solución final mejor que s entonces
                s:=y
                C:=min{C,Coste(y)}
            sino si y no es solución final Y CI(y)<C entonces
                LNV:=LNV+{y}
                C:=min{C,CS(y)}
        finpara
    finsi
finmientras

```

13.1.5. Análisis de algoritmos de ramificación y poda

El tiempo de ejecución de un algoritmo por ramificación y poda depende de:

- El número de nodos recorridos, que depende de lo efectiva que sea la ramificación y la poda, o lo que es lo mismo, de lo precisa que sea la estimación del beneficio y lo próximas que estén las cotas.

- El tiempo gastado en cada nodo, que es el tiempo de calcular la estimación del beneficio y las cotas y el de manejo de la lista de nodos vivos. Normalmente el coste es mayor para calcular estimaciones precisas y buenas cotas.

En el peor caso, el tiempo es igual que el de un algoritmo con backtracking (o peor si tenemos en cuenta el tiempo de gestión de la lista de nodos vivos). Pero normalmente se suelen obtener mejoras respecto al backtracking.

Para obtener un algoritmo de ramificación y poda eficiente se deben obtener estimaciones del beneficio muy precisas, lo que se hace con criterios heurísticos y cotas muy ajustadas. Esto permitirá podar un gran número de nodos aunque sea a costa de un mayor coste de generación de cada nodo.

13.2. Problema de la mochila 0/1

Consideramos el problema de la mochila 0/1, que ya vimos en el capítulo anterior cómo se resolvía por backtracking.

Para diseñar un algoritmo de ramificación y poda es necesario:

- Determinar el árbol de solución a utilizar, lo que conlleva definir una representación de la solución y determinar cómo, partir de un nodo, se obtienen sus descendientes.
- Dar una manera de calcular el valor de las cotas y la estimación del beneficio.
- Definir la estrategia de ramificación y de poda.

13.2.1. Árboles de solución

Como ya vimos en el capítulo dedicado al backtracking, el árbol de solución puede ser, entre otros:

- Un árbol binario, con lo que la solución se representa como una tupla (s_1, s_2, \dots, s_n) , con $s_i = 0$ o 1 , y los hijos de un nodo representado por la solución parcial (s_1, s_2, \dots, s_k) son $(s_1, \dots, s_k, 0)$ y $(s_1, \dots, s_k, 1)$.
- Un árbol combinatorio, donde las soluciones vienen dadas por (s_1, s_2, \dots, s_m) , con $m \leq n$ y $s_i \in \{1, 2, \dots, n\}$. Los hijos de un nodo (s_1, \dots, s_k) son $(s_1, \dots, s_k, s_k + 1)$, $(s_1, \dots, s_k, s_k + 2)$, ..., (s_1, \dots, s_k, n) .

13.2.2. Estimación del beneficio y las cotas

El siguiente paso en el diseño de la solución es la estimación del beneficio y la obtención de las cotas. Se pueden usar los siguientes valores:

- El beneficio obtenido hasta el nodo actual (*BA*). Con la representación de árbol binario será $\sum_{i=1}^k b_i s_i$, y con la de árbol combinatorio $\sum_{i=1}^k b_{s_i}$.

- El beneficio obtenido sumando al actual el resultado de aplicar avance rápido 0/1 a partir del nodo y con la capacidad de la mochila que hay disponible ($AR01$). En el árbol binario es: $AR01 = BA + Mochila01(k+1, M - \sum_{i=1}^k p_i s_i)$, y en el árbol combinatorio: $AR01 = BA + Mochila01(k+1, M - \sum_{i=1}^k p_{s_i})$.
- El beneficio obtenido sumando al actual el resultado de aplicar avance rápido no 0/1 (los objetos tendrían que estar ordenados de mayor a menor $\frac{b}{p}$) a partir del nodo y con la capacidad de la mochila que hay disponible ($ARno01$), y tomar la parte entera si los beneficios son enteros. En el árbol binario es: $ARno01 = BA + \lfloor MochilaNo01(k+1, M - \sum_{i=1}^k p_i s_i) \rfloor$, y en el árbol combinatorio: $AR01 = BA + \lfloor Mochila01(k+1, M - \sum_{i=1}^k p_{s_i}) \rfloor$.

Como Cota Inferior se puede usar BA y $AR01$, pero no $ARno01$. El uso de $AR01$ nos da un valor mayor y que, por lo tanto, puede contribuir a podar más nodos, pero requiere de un mayor tiempo de ejecución. La obtención de $AR01$ se puede hacer al mismo tiempo que la de $ARno01$, por lo que no supondría mucho coste adicional.

Como Cota Superior se puede usar $ARno01$, ya que el problema 0/1 es un caso particular del no 0/1 y por tanto la solución óptima del 0/1 está acotada superiormente por la del no 0/1, como vimos en el anterior capítulo.

La Estimación del Beneficio puede coincidir con la cota inferior o la superior, o ser la media de ellas. Cuál es la mejor estimación no lo sabemos, pero puede ser más apropiado tomar valores relacionados con $AR01$ y $ARno01$ (que suponen soluciones particulares que se obtienen intentando optimizar) que utilizar el BA , que no corresponde a ninguna solución obtenida intentando optimizar.

13.2.3. Estrategias de ramificación y poda

Para realizar la poda se utiliza una variable de poda C , donde se guarda el valor de la mayor cota inferior de los nodos generados hasta ese momento y que cumplen las restricciones del problema (no exceder la capacidad de la mochila). Dado que a partir de un nodo siempre hay solución, si para un nodo su cota superior es menor o igual que C , se puede podar ese nodo.

Para la estrategia de ramificación, puesto que tenemos una estimación del coste, se usará una estrategia LC (en este caso MB), con lo que la lista de nodos vivos se mantendrá ordenada por beneficio estimado. La gestión de esta lista supone un tiempo adicional pues, por cada nodo generado, hay que recorrerla hasta encontrar la posición donde se inserta el nodo. A igualdad de beneficio se puede decidir utilizar una estrategia FIFO o LIFO. Podemos pensar que es mejor usar una FIFO si los objetos están ordenados de mayor a menor beneficio partido por peso y el árbol es binario 1-0 o combinatorio, pues en ese caso se generan primero los nodos correspondientes a objetos de mayor $\frac{b}{p}$, que son los que parece que van a formar parte de la solución óptima. Por otro lado, utilizando una estrategia LIFO se exploran primero los nodos más profundos en el árbol, que están más cerca de nodos solución y por tanto dan valores que pueden producir podas

más productivas. No está claro cuál de las dos posibilidades es mejor, pero el uso de la estrategia LC debe hacer que no haya mucha diferencia entre usar FIFO o LIFO.

13.2.4. Algoritmo

La información asociada a cada nodo consiste en: el beneficio y el peso acumulados hasta ese nodo, las cotas y la estimación del beneficio, y la solución parcial, que viene dada por el nivel del árbol por el que vamos y el array solución. Todos estos valores se pueden agrupar en un registro:

```
tipo nodo=registro
    b_act,p_act:entero
    CI,BE,CS:entero
    s:array[1..n] de entero
    nivel:entero
finregistro
```

Y el esquema del algoritmo para árbol binario podría ser:

Algoritmo 13.2 Problema de la mochila 0/1 por ramificación y poda.

```
operación Mochila01BB(b,p:array[1..n] de entero;M:entero;var s:nodo)
    inic:=Nodoinicial(b,p,M)
    C:=inic.CI
    LNV:={inic}
    s.b_act:=-∞
    mientras LNV ≠ ∅ hacer
        x:=Seleccionar(LNV)
        LNV:= LNV - {x}
        si x.CS > C entonces //Si no se cumple se poda x
            para i=0,1 hacer
                y:=Generar(x,i,b,p,M)
                si (y.nivel=n) Y (y.b_act > s.b_act) entonces
                    s:=y
                    C:=max{C,s.b_act}
                sino si (y.nivel < n) Y (y.CS > C) entonces
                    LNV:=LNV+{y}
                    C:=max{C,y.CI}
            finsi
        finpara
    finsi
    finmientras
```

Donde las funciones Nodoinicial y Generar son:

```
operación Nodoinicial(b,p:array[1..n] de entero;M:entero):nodo
    res.CI:=0
    res.CS:=[MochilaVoraz(1,M,b,p)]
    res.BE:=Mochila01Voraz (1,M,b,p)
    res.nivel:=0
```

```

res.b_act:=0
res.p_act:=0
res.s← 0
devolver res
operación Generar(x:nodo;i:0,1;b,p:array[1..n] de entero;M:entero):nodo
    res.S:=x.s
    res.nivel:=x.nivel+1
    res.s[res.nivel]:=i
    si i=0 entonces
        res.b_act:=x.b_act
        res.b_act:=x.p_act
    sino
        res.b_act:=x.b_act+b[res.nivel]
        res.p_act:=x.p_act+p[res.nivel]
    finsi
    res.Cl:=res.b_act
    res.BE:=res.Cl+Mochila01Voraz(res.nivel+1,M-res.p_act,b,p)
    res.CS:=res.Cl+[MochilaVoraz(res.nivel+1,M-res.p_act,b,p)]
    si res.p_act>M entonces //Si sobrepasa la capacidad, descartar el nodo
        res.Cl:=res.CS:= res.BE:=-∞
    finsi
    devolver res

```

Podemos hacer algunas consideraciones sobre el esquema presentado:

- Como cota inferior hemos usado BA , pero podríamos usar $AR01$.
- Si en un nodo $res.Cl = res.CS$, el nodo se poda a sí mismo. Deberíamos evitarlo o bien comprobando que el valor de la variable de poda no lo ha actualizado el mismo nodo que estamos podando, o porque estemos en un problema donde no se puede dar la igualdad sino en nodos terminales.
- Cuando en un nodo se sobrepasa la capacidad de la mochila aseguramos que se pade el nodo haciendo $res.Cl := res.CS := res.BE := -\infty$, pero hay otras maneras de asegurar esa poda, como puede ser haciendo $res.nivel := n+1$.

Ejemplo 13.1 Consideramos el mismo problema del capítulo anterior: $n = 4$, $M = 7$, $b = (2, 3, 4, 5)$ y $p = (1, 2, 3, 4)$.

En cada nodo la cota inferior es el beneficio que se obtendría con los objetos que se han incluido hasta ese nodo, 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.

Con un árbol binario, generando los nodos en el orden 0,1 y con recorrido LC-FIFO, el árbol viene dado en la figura 13.5.

Vemos que se recorren 9 nodos mientras que con el método del backtracking se recorrían 15. Esto no quiere decir que ramificación y poda recorra siempre menos nodos

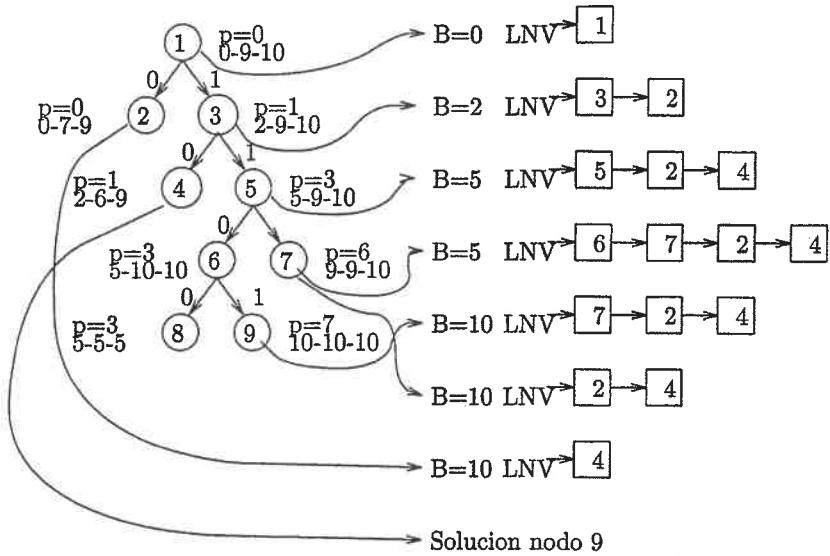


Figura 13.5: Recorrido del árbol de soluciones del ejemplo de la mochila 0/1, con árbol binario. p denota p_{act} , y los tres valores de abajo denotan la cota inferior, el beneficio estimado y la cota superior.

que el backtracking. Por ejemplo, si generamos las soluciones primero 1 y después 0 (recorremos el árbol de derecha a izquierda), con ramificación y poda se sigue recorriendo 9 nodos, pero con backtracking se recorrían 8. Además, si consideramos el árbol combinatorio, con backtracking se recorren 6 nodos y con ramificación y poda 11, tal como se muestra en la figura 13.6.

¿Qué quiere decir esto? No hay normas generales que nos digan cuándo es preferible 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 pade, y por lo tanto es preferible una técnica de ramificación y poda, siempre que se puedan encontrar unas cotas y una estimación del beneficio que parezcan apropiadas.

13.3. Secuenciamiento de trabajos con plazos

13.3.1. Definición del problema

Consideraremos 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 procesador, un plazo de ejecución por cada trabajo $d = (d_1, d_2, \dots, d_n)$, unas penalizaciones $p = (p_1, p_2, \dots, p_n)$, y unas duraciones de los trabajos $t = (t_1, t_2, \dots, t_n)$. Se tiene la penalización p_i si el trabajo i no empieza a ejecutarse dentro de su plazo d_i .

Se trata de hacer una planificación de las tareas de forma que se minimice la penalización de las tareas no ejecutadas.

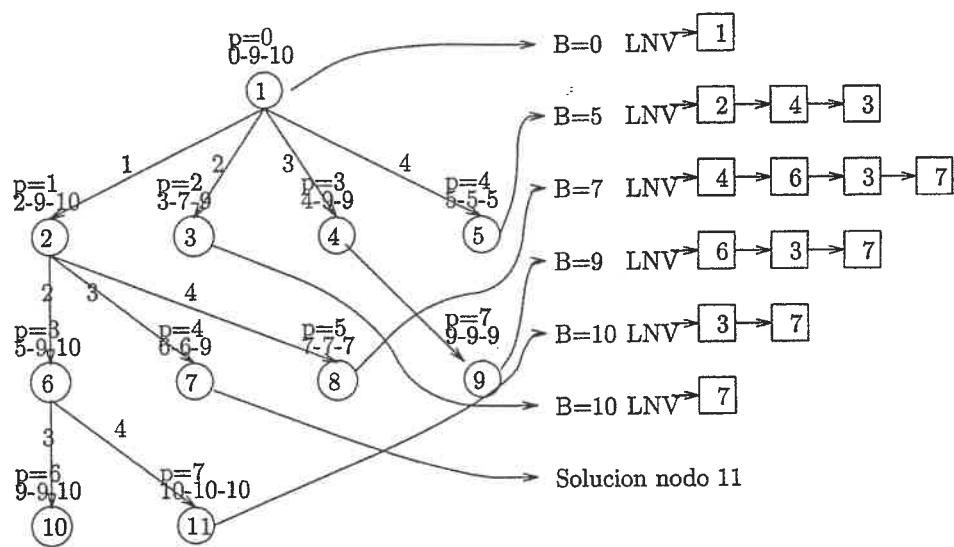


Figura 13.6: Recorrido del árbol de soluciones del ejemplo de la mochila 0/1, con árbol combinatorio. p denota p_{act} , y los tres valores de abajo denotan la cota inferior, el beneficio estimado y la cota superior.

13.3.2. Diseño de una solución

Igual que en el algoritmo voraz, dado un conjunto de tareas el orden de ejecución será en orden creciente de plazos, por lo que se preprocesará la entrada para ordenarla de menor a mayor d_i .

Se pueden utilizar árbol binario o combinatorio.

La cota inferior en cada nodo puede ser la penalización de los trabajos asignados con valor 0 hasta este nodo (son los trabajos que no se ejecutan).

La cota superior puede ser la inferior más la penalización de los trabajos no considerados hasta este momento, pues podría ocurrir que no se ejecute ninguno más dentro de su plazo.

La estimación del coste en cada nodo puede ser la media de las dos cotas anteriores.

Ejemplo 13.2 Con $n = 4$, $p = (5, 10, 6, 3)$, $d = (1, 3, 2, 1)$, y $t = (1, 2, 1, 1)$; para el nodo $(1, 0, 1)$, la cota inferior es $CI = p_2 = 10$, la cota superior es $CS = CI + p_4 = 10 + 3 = 13$, y la estimación del coste $CE = \frac{CI+CS}{2} = 11.5$.

Como el problema es de minimización y a partir de cada nodo existe al menos una solución, para la poda se lleva una variable C que contendrá el valor de la menor cota superior hasta ese punto, y se poda un nodo si su cota inferior es mayor que C . La estrategia de ramificación puede ser LC-FIFO.

Por cada nodo hay que comprobar si corresponde a una 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 plazos, pero teniendo en este caso en cuenta los tiempos de ejecución de cada trabajo.

Ejemplo 13.3 Consideramos el problema con $p = (5, 10, 6, 3)$, $d = (1, 3, 2, 1)$ y $t = (1, 2, 1, 1)$.

En la figura 13.7 se muestra el recorrido del árbol de soluciones con estrategia FIFO, sin usar estimación del coste.

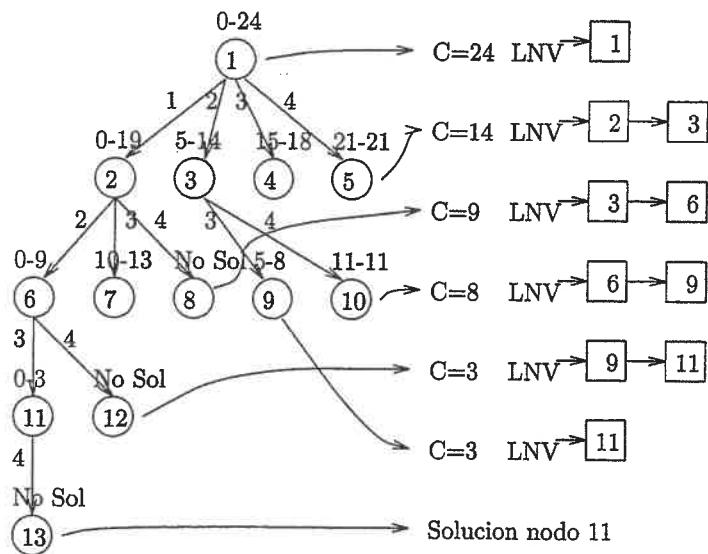


Figura 13.7: Recorrido del árbol de soluciones combinatorio del problema de secuencia de trabajos, con estrategia FIFO.

La solución será ejecutar los trabajos 1, 2 y 3 (nodo 11), pero en el orden 1,3,2.

Estimando el coste con la media entre las dos cotas, se puede hacer un recorrido con estrategia LC-FIFO, que se muestra en la figura 13.8, donde se ve que el utilizar una mejor estrategia de ramificación ha reducido un poco el número de nodos a generar. No se muestran en este caso los valores de las cotas y la estimación en los nodos que se podan.

El árbol binario tiene más nodos que el combinatorio, pero vemos en la figura 13.9 que el número de nodos que se genera es menor en este caso, debido a que en un árbol binario se generan menos hijos de cada nodo y la poda es más productiva. Los nodos 4 y 6 se insertarían en la lista de nodos vivos, pero hemos considerado que, al tener cada hijo solo dos nodos, para cada nodo que se extrae de la lista de nodos vivos, se generan todos sus hijos, se actualiza la variable de poda y a continuación se comprueba si se podan los nodos que se han generado y se insertan en la lista los no podados. Si no procedemos de este modo se generaría la misma cantidad de nodos, aunque con algunos de ellos (el 4 y el 6) se haría más trabajo al insertarlos en la lista para posteriormente sacarlos y podarlos.

13.4. Problema de las reinas

El problema de las reinas es un problema típico de backtracking que difícilmente puede adaptarse al esquema de ramificación y poda, que además es apropiado para pro-

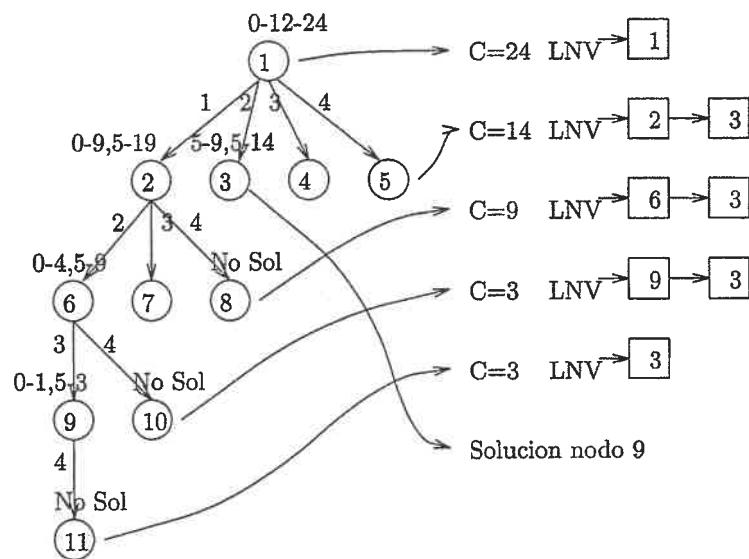


Figura 13.8: Recorrido del árbol de soluciones combinatorio del problema de secuenciamiento de trabajos, con estrategia LC-FIFO.

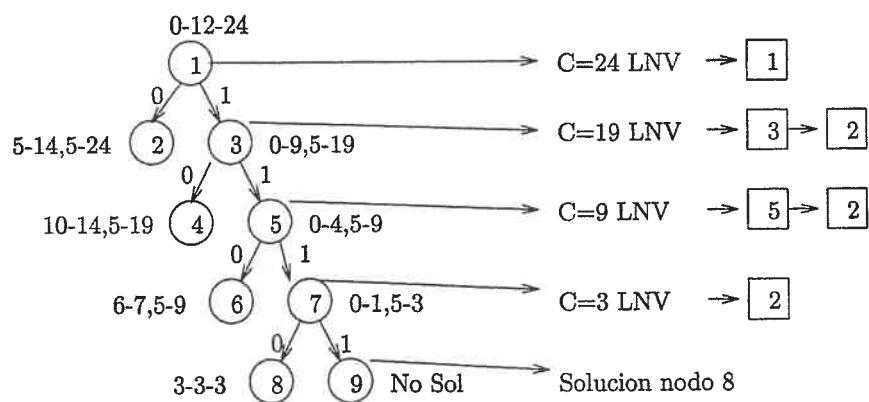


Figura 13.9: Recorrido del árbol de soluciones binario del problema de secuenciamiento de trabajos, con estrategia LC-FIFO.

blemas 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 ramificación y poda, asociando como 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 sigue buscando porque hemos alcanzado el valor $+\infty$.

La estimación del beneficio nos servirá para guiar la búsqueda. A cada configuración del tablero se asignará un valor que indique lo “prometedora” que es la configuración. Esta medida es heurística, y no nos asegura que las decisiones tomadas nos acerquen antes a una solución. Hay varias posibilidades:

- 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 13.10. 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 (12.4) 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 se pueda hacer una estimación del beneficio productiva, ¡sólo que no hemos sabido hacerla!
- El problema con la estimación anterior es que los nodos de los primeros niveles tienen normalmente más casillas libres, por lo que cuando estamos cerca de una solución (nodo 7) se retrocede a nodos de los niveles superiores. 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 13.11. 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.
- Con la segunda estimación seguimos teniendo el mismo problema que con la primera, a pesar de que se discrimina algo mejor (desde el principio se diferencia entre el nodo 3 y 2, siendo a partir del 3 donde está la solución). Podemos intentar mejorarlo teniendo 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 13.12. Se ve que ahorramos un nodo respecto a los métodos anteriores.

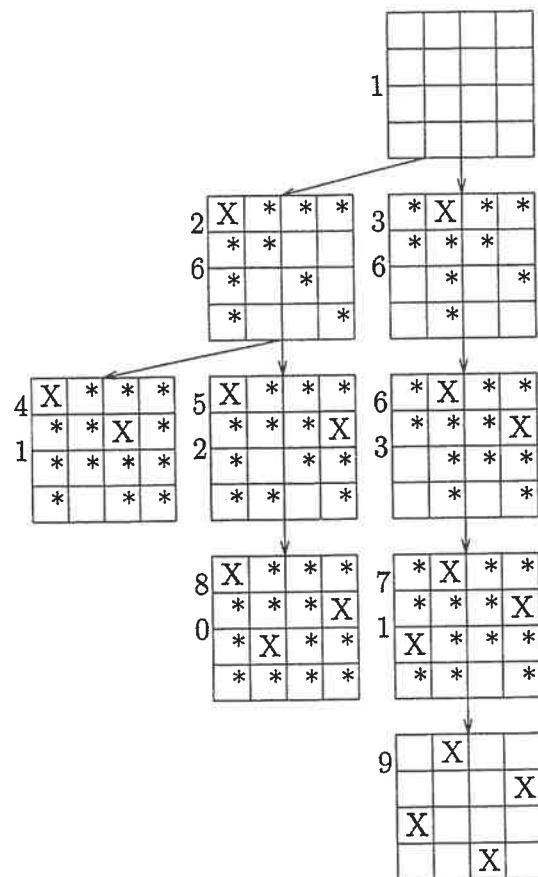


Figura 13.10: Recorrido del árbol en el problema de las 4 reinas cuando se estima el beneficio proporcionalmente al número de casillas no alcanzables.

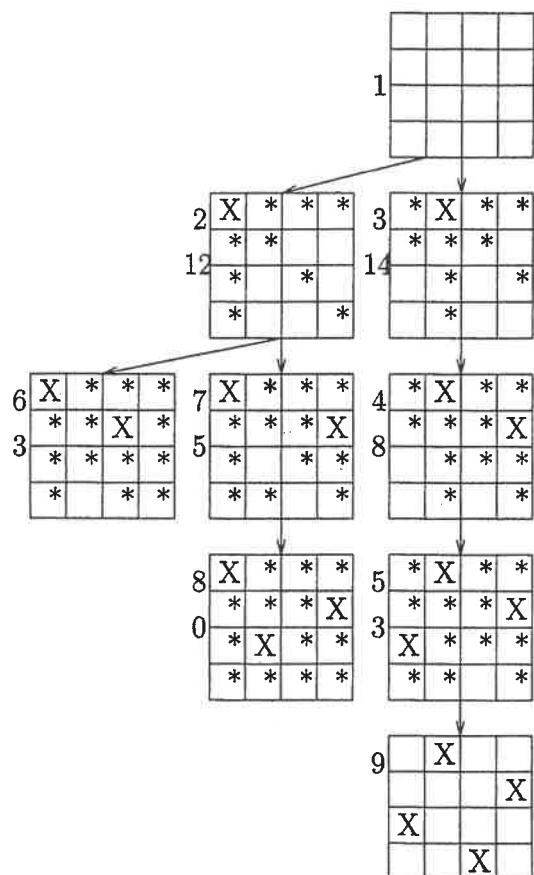


Figura 13.11: Recorrido del árbol en el problema de las 4 reinas cuando se estima el beneficio proporcionalmente al número de casillas no alcanzables teniendo en cuenta la fila en la que están las casillas.

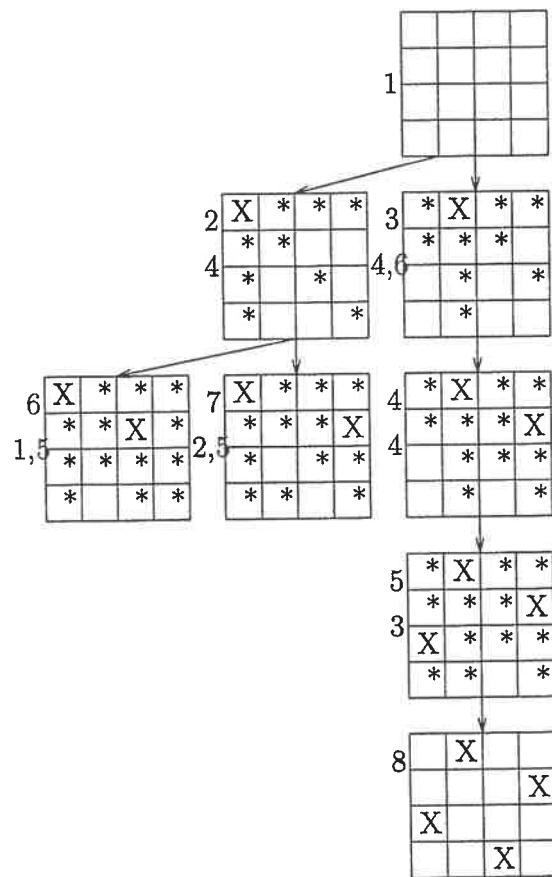


Figura 13.12: Recorrido del árbol en el problema de las 4 reinas cuando se estima el beneficio proporcionalmente al número de casillas no alcanzables teniendo en cuenta la fila en la que están las casillas y el nivel en que se encuentra el nodo.

La estimación del beneficio puede ser muy costosa en este caso, pues hay que evaluar las posiciones del tablero por debajo de la fila donde se coloca la reina, lo que tiene un orden cuadrático. Además, en el ejemplo sólo se ha ahorrado un nodo respecto a resolver el problema por backtracking. No sabemos si el utilizar ramificación y poda puede ser preferible a usar backtracking, pero en cualquier caso parece que lo sería para problemas de dimensión grande, donde la reducción en el número de nodos pueda ser significativa y compensar el coste de la estimación del beneficio.

La misma estimación se puede utilizar para hacer un backtracking guiado. La diferencia con ramificación y poda es que para cada nodo se evalúan sus descendientes para decidir por cuál continuar, pero no se utiliza una lista de nodos vivos para guardar la información asociada a esos nodos y decidir el siguiente a recorrer, sino que se sigue por el nodo seleccionado, generando sus descendientes hasta que se retrocede. Al retroceder a un nodo se vuelven a evaluar sus descendientes salvo los que ya se han generado. De esta manera, en la figura 13.12 se recorrerían los nodos en el orden 1, 3, 4, 5, 8. Vemos que el método lleva directamente a la solución, pero tampoco podemos asegurar que este backtracking guiado sea mejor que la estrategia de ramificación y poda.

Ejercicios resueltos

Ejercicio 13.1 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 ramificación y poda: 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 será 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 (figura 13.13). 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 $\sum_{i=1}^{niveles} T[i, s[i]]$. 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 $nivel$.

La cota superior de un nodo ($CS(nodo)$) puede ser la cota inferior más la suma de

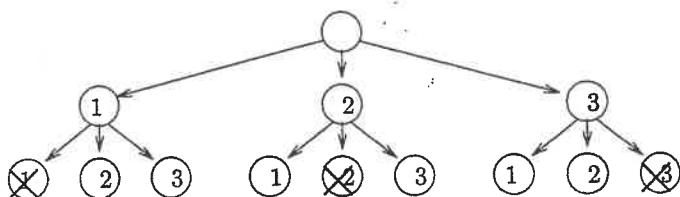


Figura 13.13: Árbol de soluciones para el problema de asignación de trabajos.

los beneficios máximos de los trabajos aún no asignados:

$$\text{CS}(\text{nodo}) = \text{CI}(\text{nodo}) + \sum_{i=nivel+1}^n \max_{j=1,\dots,n} \{T[i,j]\}$$

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 $j = 1, \dots, n$, 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. Ese valor también se podría usar como cota inferior.

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 $\text{CS}(\text{nodo}) \leq B$ 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.

Ejercicio 13.2 En un problema que se resuelve por la técnica de ramificación y poda 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. (Consideraremos 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: $0.875(n^2 + n)N$
- caso 3: $0.625(n^2 + n)N$
- 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 cuarto 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 $n = 1.4$.

Ejercicio 13.3 Dado el árbol de soluciones de la figura 13.14, 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, ramificación y poda FIFO, LIFO, LC-FIFO y LC-LIFO.

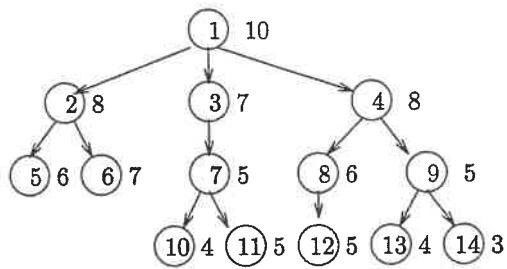


Figura 13.14: Árbol de soluciones del ejercicio 13.3.

Solución.

- En backtracking se recorre el árbol haciendo en cada nodo el recorrido: nodo, backtracking de hijo₁ de nodo, backtracking de hijo₂ de nodo, ..., backtracking de último hijo de nodo; pero en nuestro caso se pueden eliminar búsquedas que no 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 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.

- Ramificación y poda FIFO:

En la técnica ramificación y poda 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.

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).

- Ramificación y poda 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).

nodo generado	pila de nodos vivos
12	3, 2

(solución óptima la del nodo 12, beneficio 5).

nodo generado	pila de nodos vivos
7	2

(a partir del nodo 7 el máximo beneficio que podemos alcanzar es 5, que no mejora la solución de 12).

5, 6

- Ramificación y poda 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 6. Se elimina el nodo 3 de la lista pues no puede mejorar el beneficio del nodo 6).

8, 9

■ Ramificación y poda 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 podan y se acaba aquí).

Ejercicio 13.4 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 ramificación y poda. Poner ejemplos que ilustren los métodos indicados.

Solución.

Obviamente, este problema se puede resolver por el algoritmo de Dijkstra, estudiado en el capítulo 5, pero veremos cómo resolverlo por ramificación y poda.

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 $s:\text{array}[0..l+1] \text{ de } 1..n$ 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:

```

 $L:=0$ 
 $s[0]:=1$ 
 $s[l+1]:=n$ 
para  $i:=1,\dots,l$  hacer
   $min:=d[s[i-1],2+(n-2)(i-1)]/|$ 
   $s[i]:=2+(n-2)(i-1)/|$ 

```

```

para  $j:=3+(n-2)(i-1)/l, \dots, 2+(n-2)i/l$  hacer
  si  $d[s[i-1], j] < min$  entonces
     $min:=d[s[i-1], j]$ 
     $s[l]:=j$ 
  finsi
finpara
 $L:=L+d[s[i-1], s[l]]$ 
finpara
 $L:=L+d[s[l], n]$ 

```

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 **para** externo por un **mientras**. Al acabar el **para** 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 **seleccion** 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 si decidimos evitar el seguir generando caminos inexistentes consistiría en comprobar si el valor de min es ∞ . La función para 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 grafo de la figura 13.15, cuyo recorrido por backtracking se muestra en el árbol de la figura 13.16, donde dentro de cada nodo se pone el número de nodo a que corresponde en el grafo, en cada arista del árbol se pone el 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 ramificación y poda 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 figura 13.17

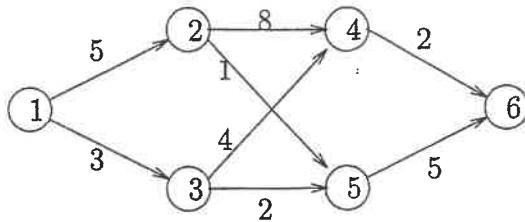


Figura 13.15: Grafo multietapa.

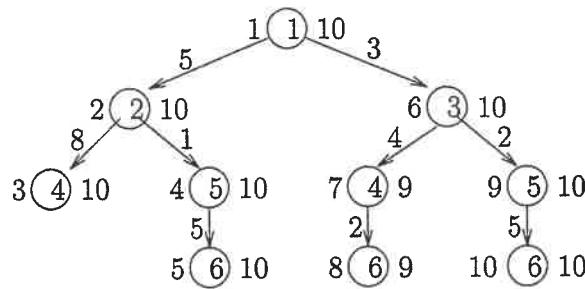


Figura 13.16: Recorrido por backtracking del árbol de soluciones para el grafo multietapa de la figura 13.15.

se muestra cómo funcionaría el método con el grafo ejemplo. Los números 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.

Ejercicio 13.5 Se quiere resolver el problema de, dado un damero rectangular de dimensiones $n \times m$ y una serie de p piezas rectangulares de dimensiones $n_1 \times m_1, \dots, n_p \times m_p$, 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.

- Resolver el problema por backtracking según un esquema no recursivo.
- Indicar alguna estrategia para resolver el problema por ramificación y poda.

Solución.

- 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 izquierdo 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

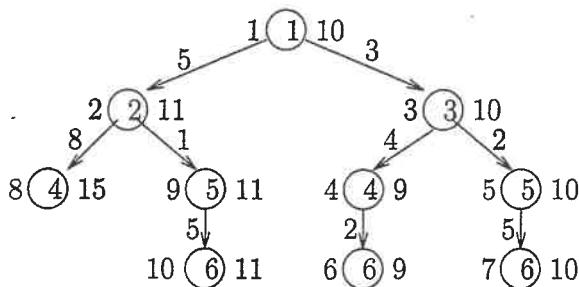


Figura 13.17: Recorrido por ramificación y poda del árbol de soluciones para el grafo multietapa de la figura 13.15.

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 $t:\text{array}[1..n,1..m]$ 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.

Con todas estas observaciones el esquema podría ser:

$t \leftarrow 0$

$s \leftarrow (-1, -1)$

```

 $so \leftarrow (0, 0)$ 
 $np := 0$ 
 $npo := 0$ 
 $puesta \leftarrow 0$ 
 $nivel := 1$ 
generar
repetir
    si solucion entonces
        si  $np > npo$  entonces
             $so \leftarrow s$ 
             $npo := n$ 
        finsi
    finsi
    si criterio entonces
         $nivel++$ 
        generar
    sino
        mientras  $nivel \neq 0$  Y NO mas hermanos hacer
            retroceder
        finmientras
        si  $nivel \neq 0$  entonces
            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:

operación solucion:

```

 $valor := \text{terminal Y cabe}$ 
si  $valor$  entonces
    poner
finsi
devolver  $valor$ 

```

Donde:

operación terminal:

```
devolver  $nivel = p$ 
```

operación cabe:

```
 $vale := \text{verdadero}$ 
```

```
 $i := 0$ 
```

```
 $j := 0$ 
```

```
mientras  $vale$  Y  $i \neq piezas[nivel].x$  hacer
```

```
si  $s[nivel].x + i > n$  O  $s[nivel].y + j > m$  entonces
```

```
 $vale := \text{falso} \quad // \text{se sale del tablero}$ 
```

```

finsi
si vale entonces //si no se sale comprueba si solapa
  si t[s[nivel].x+i,s[nivel].y+j]=1 entonces
    vale:=falso
  finsi //pasa a la siguiente posición de la pieza
  j++
  si j=piezas[nivel].y entonces
    j:=0
    i++
  finsi
finsi
finmientras
devolver vale

```

y:

operación poner:

```

//pone las piezas que previamente ha comprobado que no se salen del tablero
//ni solapan con otra ya en el tablero
para i:=s[nivel].x,..., s[nivel].x+piezas[nivel].x-1 hacer
  para j:=s[nivel].y,...,s[nivel].y+piezas[nivel].y-1 hacer
    t[i,j]:=1
  finpara
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:

operación generar:

```

puesta[nivel]:=0
si s[nivel].x=-1 entonces //no se ha generado todavía ningún hijo
  s[nivel].x:=1
  s[nivel].y:=1
sino si s[nivel].x=n Y s[nivel].y=m entonces
//se han comprobado todas la posiciones en el tablero
  s[nivel].x:=0 //se indica que no se pone la pieza
sino
  si s[nivel].y≠m entonces //si no es la última columna
    s[nivel].y:=s[nivel].y+1
  sino
    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:

operación 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:

operación retroceder:
si $puesta[nivel]=1$ **entonces**
 quitar
finsi
 $s[nivel].x:=-1$
 $nivel--$

Donde:

operación quitar:
para $i:=s[nivel].x, \dots, s[nivel].x+piezas[nivel].x-1$ **hacer**
para $j:=s[nivel].y, \dots, s[nivel].y+piezas[nivel].y-1$ **hacer**
 $t[i,j]:=0$
finpara
finpara
 $np--$

b) Para resolver el problema por ramificación y poda, 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 la 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 ramificación 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.

Ejercicio 13.6 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 $i_1 \times j_1, i_2 \times j_2, \dots, i_n \times j_n$, cada una de ellas con beneficio b_1, b_2, \dots, 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 $2 \times 3, 1 \times 4, 3 \times 1$ 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 ramificación y poda del mismo problema pero sin poder dividirse en cuadrados las piezas.

Solución.

a) El problema es similar al 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:

```

operación avance_rapido:
  ordenar piezas de mayor a menor  $\frac{b}{ij}$ 
  libres:= $M_1 M_2$ 
  benef:=0
  pieza:=1
  mientras pieza≤n Y  $i_{\text{pieza}} * j_{\text{pieza}} \leq \text{libres}$  hacer
    incluir(pieza, $i_{\text{pieza}} * j_{\text{pieza}}$ )
    libres:=libres -  $i_{\text{pieza}} * j_{\text{pieza}}$ 
    benef:=benef+ $b_{\text{pieza}}$ 
    pieza++
  finmientras
  si pieza≤n entonces
    incluir(pieza,libres)
    benef:=benef+ $\frac{\text{libres}}{i_{\text{pieza}} * j_{\text{pieza}}} b_{\text{pieza}}$ 
  finsi
```

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

Ejercicios resueltos

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 tabla, 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 el de la figura 13.18. 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.

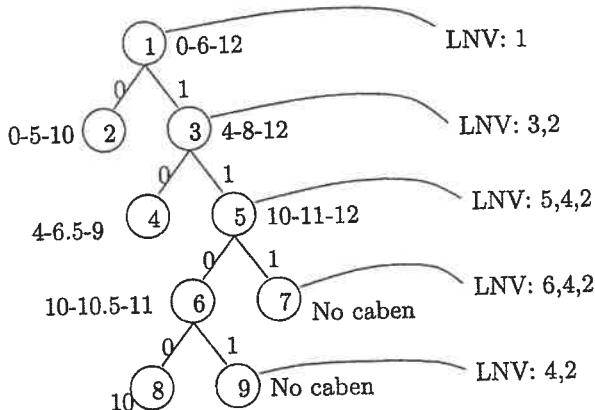


Figura 13.18: Árbol de solución por ramificación y poda del ejercicio 13.6.

Ejercicio 13.7 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 ($media = \frac{\sum_{i=1}^n t[i, s[i]]}{n}$).

- Resolver el problema por avance rápido, intentando minimizar la desviación.
- Indicar cómo se puede utilizar el método de avance rápido anterior para resolver el problema por ramificación y poda. Indicar cómo se calcularían las cotas inferior y superior y la estimación del beneficio en cada nodo, y cómo se haría la poda.

Solución.

- La solución se almacenará en un array $s:array[1..n]$ de $1..n$; 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 $asignado:array[1..n]$ de booleano donde verdadero indica que el trabajo correspondiente se ha asignado, por lo que estará inicializado a falso.

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 **para**. 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 preprocessamiento 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
para i:=1,...,n hacer
    para j:=1,...,n hacer
        m:=m+t[i,j]
    finpara
finpara
m:= $\frac{m}{n^2}$ 
para i:=1,...,n hacer
    min:=∞
    para j:=1,...,n hacer
        si NO asignado[j] Y |m - t[i,j]| < min entonces
            k:=j
            min:=|m - t[i,j]|
        finsi
    finpara
    asignado[k]:=verdadero
    s[i]:=k
finsi

```

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.

- Ya que la media que hemos calculado en el preprocessamiento 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: $media = \frac{t[1,s[1]] + \dots + t[n,s[n]]}{n}$). 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.

Ejercicios propuestos

Ejercicio 13.8 Resolver por backtracking y ramificación y poda el problema de obtener de una serie de números los que suman S con una cantidad mínima de números.

Ejercicio 13.9 Diseñar un algoritmo por ramificación y poda 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.

Ejercicio 13.10 Dados n números naturales $S = \{x_1, x_2, \dots, x_n\}$ y un número natural N . Queremos encontrar el subconjunto de S ($T \subset S$, $T = \{y_1, y_2, \dots, y_m\}$) que minimice $|N - y_1 - y_2 - \dots - y_m|$. Indicar cómo se podría resolver por ramificación y poda, indicando cómo se calcularían la cota inferior y superior y la estimación del beneficio en cada nodo, y cómo se haría la ramificación y la poda.

Ejercicio 13.11 Resolver por ramificación y poda el problema de la mochila 0-0.5-1. 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.

Ejercicio 13.12 Dados n programas de longitudes l_1, l_2, \dots, 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 ramificación y poda 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).

Ejercicio 13.13 Aplicar la técnica de ramificación y poda al problema del viajante: dado un grafo no dirigido y ponderado, encontrar un ciclo simple de coste mínimo que pase

por todos los nodos una sola vez. Definir la representación de la solución y una forma de obtener la cota superior, inferior y el beneficio estimado a partir de una solución parcial. Escribir el algoritmo para resolver el problema. ¿Cuál es el orden de complejidad del algoritmo?

Ejercicio 13.14 En Windows, cuando se quieren copiar varios archivos en disquetes, se empiezan a copiar por el orden seleccionado. Cuando un disquete se llena, se mete otro y se sigue copiando. Con este método, puede que necesitemos más disquetes de los necesarios. Por ejemplo, suponiendo que en un disquete caben 1.4 Mbytes, y los archivos son de tamaño: 400 Kb, 400 Kb, 800 Kb, 800 Kb, necesitaríamos 3 discos, cuando podríamos hacerlo con sólo 2. Se supone que los archivos no se pueden partir y que son de menor tamaño que la capacidad del disquete. Diseñar una solución para el problema anterior utilizando ramificación y poda. Dado un array $T[1..n]$, con los tamaños de los archivos, y una cantidad M que indica el espacio libre de los disquetes, el problema consiste en encontrar el número mínimo de disquetes necesarios para copiar los archivos, y la asignación $S[1..n]$, indicando el número de disquete en el que se debe copiar cada archivo.

a) Exponer qué forma tendrá el árbol de búsqueda, con un ejemplo sencillo de árbol. ¿Qué representa cada nivel del árbol? ¿Qué descendientes son generados para cada nodo? ¿Cuándo un nodo es una solución final?

b) Escribir el esquema del algoritmo de ramificación y poda. Especificar cómo son las funciones para: generar los hijos de un nodo, comprobar si un nodo es solución final. Dar fórmulas (lo más ajustadas posible) para calcular cotas y el beneficio estimado para cada nodo.

c) Establecer una estrategia de ramificación y de poda adecuadas, y mostrar la ejecución del algoritmo para el ejemplo anterior.

Cuestiones de autoevaluación

Ejercicio 13.15 Explicar cómo se podría resolver el problema de la devolución de monedas por ramificación y poda. ¿Se puede seguir algún criterio para asignar pesos a los nodos del árbol? Considerar los casos: a) disponemos de n monedas cuyos valores están almacenados en un array de 1 a n , b) disponemos de una cantidad ilimitada de monedas de cada tipo.

Ejercicio 13.16 Dado el árbol de búsqueda de la figura 13.19, 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 ramificación y poda 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.

Ejercicio 13.17 Indicar si se podría resolver el problema del paseo del caballo por ramificación y poda: 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.

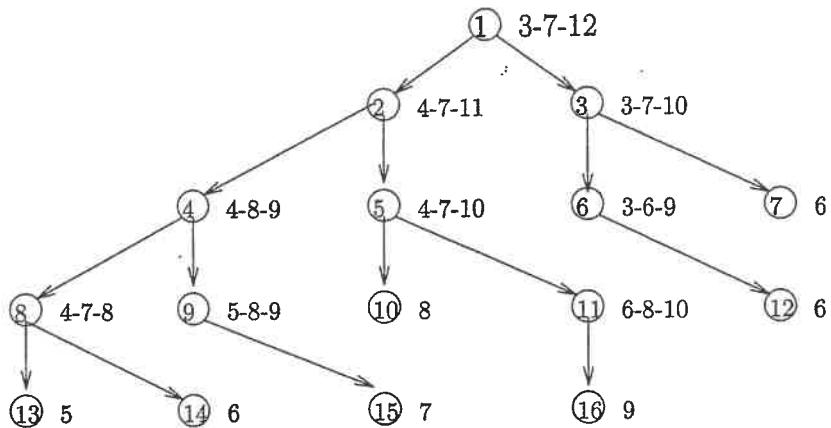


Figura 13.19: Árbol del ejercicio 13.16.

Ejercicio 13.18 Se quiere resolver por ramificación y poda un problema representado por el árbol de la figura 13.20, 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:

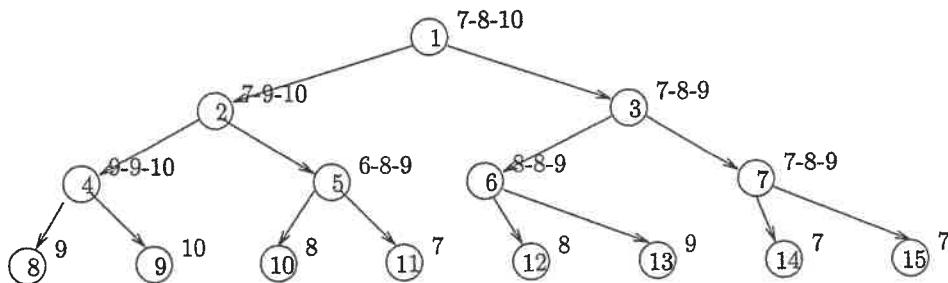


Figura 13.20: Árbol del ejercicio 13.18.

- 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.

Ejercicio 13.19 En un problema determinado, una solución está dada por una tupla de n elementos (x_1, x_2, \dots, x_n) . Para cada elemento existen en total m posibles valores. Comparar el número de nodos generados para resolver el problema usando un algoritmo de

backtracking (suponiendo que no se realiza ninguna poda), un algoritmo de avance rápido (se supondrá que cada posibilidad probada es un nodo), y un algoritmo de ramificación y poda. En el último caso, ¿podemos predecir el número exacto de nodos generados o debemos dar un mejor y peor caso?

Ejercicio 13.20 En el problema de las n reinas queremos obtener todas las posibles soluciones para un n dado. En esta situación, ¿se obtiene algún beneficio utilizando ramificación y poda en lugar de backtracking? Justifica la respuesta. En general, para un problema cualquiera de optimización donde queramos obtener todas las soluciones óptimas, ¿tendrá sentido utilizar ramificación y poda?

Ejercicio 13.21 En muchos problemas, un procedimiento de avance rápido es usado como una manera de estimar una cota o el beneficio que se puede alcanzar a partir de un nodo, para usarlo después en un algoritmo de ramificación y poda. Del mismo modo, podríamos pensar en usar un algoritmo de programación dinámica para hacer una estimación más precisa de esos mismos valores. Usando el ejemplo del problema de la mochila, muestra por lo menos dos razones por las que esta forma de realizar las estimaciones no tiene ninguna utilidad.

Referencias bibliográficas

Hay muchas menos referencias de la técnica ramificación y poda que del resto de las técnicas estudiadas. En la mayoría de los casos se encuentran dentro de capítulos sobre recorrido de grafos y de árboles de juegos. En [Brassard90] se explica sucintamente el método y en la siguiente edición ([Brassard97]) se incluye algún ejemplo más, como el problema de la mochila. En [Troya84] se estudia esta técnica y se ilustra con el problema del secuenciamiento de trabajos con plazos.

Capítulo 14

Árboles de juegos

En este capítulo vamos a estudiar la resolución de juegos donde participan dos jugadores que mueven alternativamente. En este tipo de problemas, los distintos estados del juego se pueden representar mediante un árbol de juegos. Recorriendo este árbol, o parte del mismo, se decide qué movimiento realizar. El recorrido de árboles de juegos está relacionado con las dos técnicas (backtracking y ramificación y poda) estudiadas en los dos capítulos anteriores. También en este caso se trata de recorrer el árbol de forma sistemática y evitando recorrer nodos de manera innecesaria.

Objetivos del capítulo:

- Comprender la representación de los estados de un juego por medio de árboles.
- Entender la técnica básica de recorrido de un árbol de juegos para tomar decisiones apropiadas, conocida como estrategia minimax.
- Conocer la técnica de poda alfa-beta, para la eliminación de nodos en un árbol de juegos posibilitando la exploración de mayor parte del árbol.
- Concienciarse de la importancia de la heurística en la toma de decisiones, en particular cuando éstas se toman dentro de juegos de tamaño grande o infinitos.
- Saber aplicar las técnicas de estrategia minimax y poda alfa-beta a la resolución de problemas de juegos nuevos.

Contenido del capítulo:

14.1. Árboles de juegos	225
14.2. Estrategia minimax	227
14.2.1. Algoritmo	228
14.3. Poda alfa-beta	229
14.3.1. Algoritmo	230
Ejercicios resueltos	231
Ejercicios propuestos	235
Cuestiones de autoevaluación	237
Referencias bibliográficas	238

14.1. Árboles de juegos

En este capítulo consideramos la solución de juegos con dos jugadores, A y B, que mueven alternativamente intentando ganar. El resultado del juego puede ser que gana A, gana B o hay empate. En cada movimiento un jugador podrá elegir entre un número finito, y normalmente reducido, de posibilidades. Además suponemos exclusivamente juegos en los que no influye el azar.

Las características anteriores se cumplen en muchos juegos de tablero: ajedrez, tres en raya, tic-tac-toe, las damas, cuatro en raya, el juego de los palillos, etc. Así que las técnicas que estudiaremos se podrán aplicar sobre todos ellos. Lo que analizaremos es cómo programar la toma de decisiones en cada paso para intentar ganar el juego.

El desarrollo de un juego se puede representar por una secuencia de configuraciones C_1, C_2, \dots, C_m , donde cada configuración C_i contiene la posición del juego y el jugador al que corresponde mover, $C_i = (p_i, j_i)$. Una secuencia de este tipo cumple:

- C_1 es la configuración inicial. Por ejemplo, si empieza moviendo A, será $C_1 = (p_1, A)$.
- Cada C_i con $0 < i < m$ es una situación no terminal (no ha acabado el juego).
- 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 (el juego ha terminado con victoria de A, de B o con empate).

Un juego se llama finito cuando no hay secuencias válidas de longitud ∞ . Los juegos finitos se pueden representar mediante **árboles de juegos**. Básicamente, un árbol de juegos es un árbol que representa todas las posibles evoluciones del juego, partiendo de una situación inicial. Si el árbol del juego es muy grande o infinito (por ejemplo, en el ajedrez), se generará el árbol hasta un nivel determinado y se representará sólo parte del árbol, la parte que se está explorando.

En concreto, los árboles de juegos tienen las siguientes características:

- Cada nodo representa una situación del juego en un momento dado.
- El nodo raíz representa el comienzo de una partida.
- Los descendientes de un nodo dado son los movimientos posibles de un jugador. Por lo tanto, puesto que los jugadores mueven alternativamente, los niveles impares del árbol representan los movimientos de un jugador A, y los pares los movimientos de otro jugador B.
- Un nodo hoja representa una situación donde acaba el juego. Cada nodo hoja está etiquetado con un número, que indica el resultado del juego. Por ejemplo, 1 puede indicar que acaba con victoria de A, -1 con victoria de B, y 0 que se produce un empate.

- El objetivo de cada jugador es ganar, y por lo tanto encontrar un camino en el árbol que le lleve hasta un nodo hoja con valor 1 o -1, según sea el jugador A o el B.

Lo veremos con un ejemplo:

Ejemplo 14.1 Juego del NIM.

Tenemos n palos en un montón. Los jugadores A y B mueven alternativamente, consistiendo un movimiento en quitar 1, 2 o 3 palos del montón, pero no más de los que hay. Pierde el jugador que quita el último palo, es decir, el que mueve último.

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. Llegaremos a una configuración terminal cuando no queden palos en el montón.

Una secuencia C_1, C_2, \dots, C_m que representa el desarrollo de un juego cumple:

- C_1 es la configuración inicial. Por ejemplo, si mueve A y tenemos 4 palos, será $C_1 = (4, A)$.
- C_m es una configuración terminal. Por ejemplo, $C_m = (0, X)$ siendo X el jugador ganador.

El árbol para este juego para este problema, empezando con 4 palos, es el mostrado en la figura 14.1. 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.

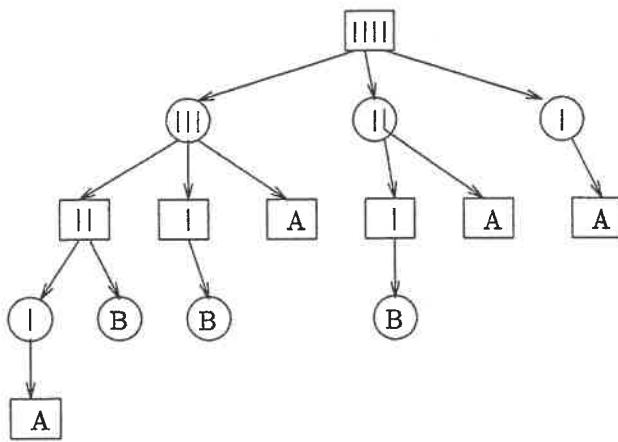


Figura 14.1: Árbol para el juego del NIM, empezando con 4 palos.

A cada configuración del juego (o nodo del árbol de juegos) se puede asociar una función de utilidad, que indica cómo de buena es la situación para un determinado jugador. Por ejemplo, si la definimos desde el punto de vista del jugador A, en los nodos

terminales donde gana A el valor puede ser 1, en los que gana B puede ser -1, y donde hay empate es cero. De esta forma la toma de decisiones se convierte en un problema de maximización (si mueve A) o minimización (si mueve B).

En el ejemplo anterior los nodos etiquetados con A tienen valor 1, los etiquetados con B valor -1 o no hay posibilidad de empate. En los juegos infinitos no se llega a nodos terminales, y la asignación de un valor a un nodo puede no estar tan clara como en este caso. Por ejemplo, en el ajedrez podría consistir en asignar un valor dependiendo de una serie de factores: piezas de cada bando en el tablero, dominio del centro o de las columnas, etc. Además, también es necesario asignar valores a los nodos no terminales para que un jugador pueda decidir, a la vista de los valores de los nodos descendientes, qué decisión tomar.

14.2. Estrategia minimax

Un jugador hará el movimiento que maximice sus posibilidades de ganar independientemente de lo que haga el otro jugador. Ya hemos visto que esto se traduce en moverse hacia una hoja con valor 1, para el jugador A, o -1, para el B. Pero ¿qué ocurre si a partir de la situación inicial no se llega directamente a esa hoja ganadora? Tendremos que *propagar* los valores de la función de utilidad, desde las hojas hacia los nodos intermedios, hasta llegar a la raíz. La propagación de los valores de utilidad en nodos no terminales se realiza de la siguiente forma:

- En los movimientos de A, el valor del nodo padre será el máximo de los valores de utilidad de los nodos hijos, ya que A intenta maximizar su beneficio.
- En los movimientos de B, el valor del nodo padre será el mínimo de los valores de utilidad de los nodos hijos, ya que B intenta maximizar su beneficio o, lo que es lo mismo, minimizar el de A.
- Este proceso se repite hacia arriba en el árbol, hasta llegar al nodo raíz (situación de partida).

En definitiva, la propagación de valores en el árbol se realiza aplicando pasos de maximizar en los niveles pares de árbol y de minimizar en los niveles impares; de ahí el nombre de *minimax*. El resultado final del problema –la decisión “qué movimiento realizar a continuación”– será tomar el hijo de la raíz para el cual se obtenga el máximo beneficio.

Si vemos el juego desde el punto de vista del jugador A, se trata de obtener el movimiento que maximice sus posibilidades de ganar, suponiendo que el jugador B juega de manera perfecta. El jugador A considera positivo ganar (valor MAX) y negativo perder (valor MIN), e intenta maximizar, tomando de todos sus posibles hijos la decisión que le lleva a un valor mayor. Sin embargo, B suponemos que juega de manera perfecta e intenta minimizar el beneficio de A suponiendo que éste no comete errores.

Ejemplo 14.2 Juego del NIM. Estrategia minimax.

En la figura 14.2 consideramos el mismo árbol de la figura 14.1, asociando a cada nodo el valor de utilidad calculado como se ha indicado, con nodos terminales con valor

1 cuando gana A y -1 si gana B. Para calcular los valores en los nodos no terminales, 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 consigue quitando 3 palos en el primer movimiento. Pero en cualquiera de las otras dos posibilidades gana B si juega bien.

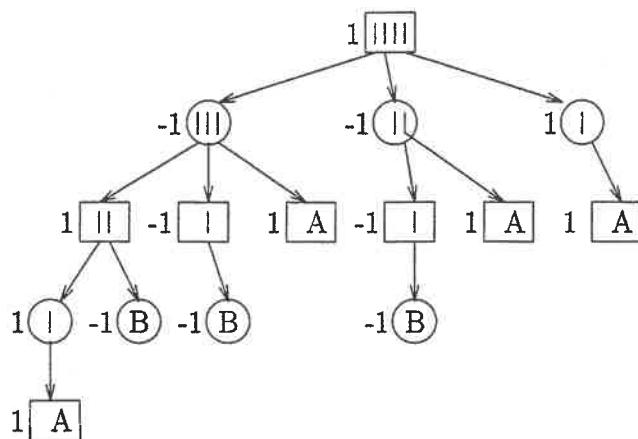


Figura 14.2: Árbol para el juego del NIM. Procedimiento minimax.

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 desendientes de ese nodo. Esto produce que el tiempo de ejecución sea muy elevado.

14.2.1. Algoritmo

Como hemos visto, para escoger el primer movimiento del jugador A hay que recorrer todo el árbol propagando los valores de los nodos hoja hasta llegar al raíz. Este recorrido se puede realizar con un backtracking. Puesto que hay propagación de los valores de los nodos hijos a los padres, usaremos una implementación recursiva.

Algoritmo 14.1 Algoritmo minimax recursivo.

```

operación Minimax(B:tipo_tablero;modo:(MAX,MIN)):real
  si B es una hoja entonces
    devolver Utilidad(B)
  sino
    si modo=MAX entonces
      valor_act:=−∞
    
```

```

sino
    valor_act:=  $\infty$ 
finsi
para cada hijo C del tablero B hacer
    si modo=MAX entonces
        valor_act:=max{valor_act, Minimax(C,MIN)}
    sino
        valor_act:=min{valor_act, Minimax(C,MAX)}
    finsi
finpara
devolver valor_act
finsi

```

14.3. Poda alfa-beta

La poda alfa-beta es una manera de mejorar la estrategia minimax, evitando el recorrido de nodos que no aportan información adicional. Es decir, los nodos podados son aquellos cuyo recorrido y evaluación podemos garantizar que no va a modificar el valor del máximo o el mínimo que se está calculando.

A cada nodo no terminal donde mueve el jugador A se le asocia un valor alfa, y en los que mueve el jugador B un valor beta. El alfa valor de una posición MAX (un nivel donde se calcula el máximo) está definido como el mínimo valor posible de esa posición, y el valor beta de una posición MIN es el máximo valor posible para esa posición. Si consideramos el algoritmo recursivo 14.1, el alfa y beta valor de un nodo es simplemente el *valor_act* de ese nodo en un momento dado de la ejecución del algoritmo. Por ejemplo, si en un nodo MAX tenemos que *valor_act* = 7, podemos asegurar que el valor final de ese nodo será como mínimo 7.

Consideremos el árbol de juegos de la figura 14.3. Un alfa valor es indicado con $\geq X$, y un beta valor con $\leq X$. En el árbol mostrado tenemos ya recorrida cierta parte y nos queda por evaluar el subárbol D. Ahora bien, valga lo que valga D el valor de la raíz será siempre 50 y el movimiento óptimo será hacia el hijo izquierdo¹. En consecuencia, podemos podar el nodo D sin eliminar el movimiento óptimo. La situación que posibilita esta poda es que el beta valor de un nodo MIN (≤ 30) es menor que el alfa valor MAX del padre (≥ 50). Esto es lo que se conoce como **poda alfa**.

De forma similar, podemos podar los hijos de un nodo MAX cuando su alfa valor sea mayor que el beta valor del nodo MIN padre, como ocurre en la figura 14.4. Esta es la **poda beta**.

En definitiva, la poda alfa-beta consiste en lo siguiente:

- **Poda alfa:** si el beta valor de una posición MIN es menor o igual que el alfa valor de su padre, no hace falta generar los hijos de esa posición (figura 14.3).

¹Para convencerse de esto, se puede probar aplicando la estrategia minimax suponiendo que D vale 10, 30, 40, 900 o cualquier cosa. Se obtendrá siempre el mismo resultado.

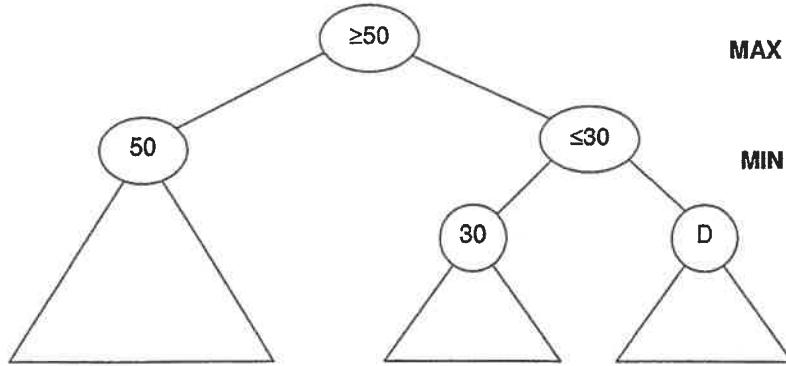


Figura 14.3: Poda alfa en árbol de juegos: podar los hijos de un nodo MIN si su beta valor (30) es menor que el alfa valor del padre MAX (50).

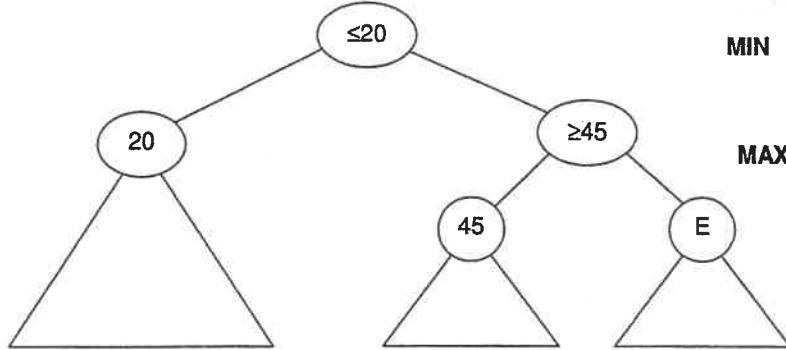


Figura 14.4: Poda beta en árbol de juegos: podar los hijos de un nodo MAX si su alfa valor (45) es mayor que el beta valor del padre MIN (20).

- **Poda beta:** si el alfa valor de una posición MAX es mayor o igual que el beta de su nodo padre, no hace falta generar los hijos de esa posición (figura 14.4).

14.3.1. Algoritmo

Vamos a ver un esquema recursivo de la poda alfa-beta. El algoritmo es básicamente el mismo que el del apartado 14.2.1, pero añadiendo dos comprobaciones: una para la poda alfa y otra para la beta. La primera llamada, para decidir el movimiento a realizar por el jugador A, se haría con `AlfaBeta(TableroIncial, ∞ , MAX)`.

Algoritmo 14.2 Procedimiento minimax recursivo con poda alfa-beta.

```

operación AlfaBeta(B:tipo_tablero;valor_padre:real;modo:(MAX,MIN)):real
    si B es una hoja entonces
        devolver Utilidad(B)
    sino
        si modo=MAX

```

```

valor_act:=-∞
sino
    valor_act:=∞
finsi
para cada hijo C del tablero B hacer
    si modo=MAX entonces
        valor_act:=max{valor_act,AlfaBeta(C,valor_act,MIN)}
        si valor_act≥valor_padre entonces //Poda beta
            Salir del bucle, descartando los demás hijos de B
        finsi
    sino //modo=MIN
        valor_act:=min{valor_act,AlfaBeta(C,valor_act,MAX)}
        si valor_act≤valor_padre entonces //Poda alfa
            Salir del bucle, descartando los demás hijos de B
        finsi
    finsi
finpara
devolver valor_act
finsi

```

Ejercicios resueltos

Ejercicio 14.1 Consideramos un juego del tipo de los estudiados en este capítulo. Si tenemos en los nodos terminales los valores que se indican en la figura 14.5, indicar el recorrido del árbol a partir del nodo 2 utilizando la poda alfa-beta, dando los alfa y los beta valores en cada nodo por cada paso del recorrido, y determinar qué nodos se podan.

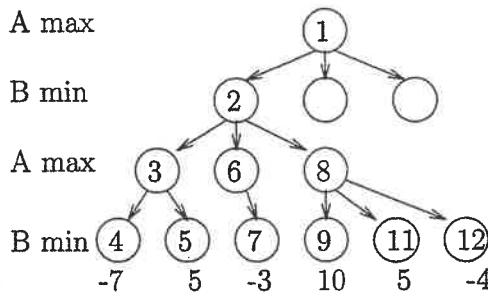


Figura 14.5: Árbol del juego del ejercicio 14.1.

Solución:

Analizamos el recorrido en la tabla 14.1, 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.

	1α	2β	3α	4	5	6α	7	8α	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

Tabla 14.1: Recorrido del árbol de la figura 14.5 con la estrategia alfa-beta.

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 . 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 , con lo que sabemos que el mínimo es -3 sin necesidad de evaluar los nodos 11 y 12 (que son podados aplicando una poda beta).

Ejercicio 14.2 Analizar el recorrido del árbol del juego del NIM, con la condición inicial y los movimientos indicados en la figura 14.1, cuando se utiliza la poda alfa-beta.

Solución.

Los nodos que se recorren y el orden en que se hace el recorrido se muestran en la figura 14.6. El valor de los alfa y beta valores en cada uno de los nodos a lo largo del recorrido, y las podas realizadas se muestran en la tabla 14.2. Vemos que algunos nodos se han podado. 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.

Ejercicio 14.3 Dar un esquema de la poda alfa-beta utilizando dos funciones, una para evaluar los alfa valores de nodos donde mueve el jugador A, y otra para evaluar los beta valores cuando mueve B. Se supondrá 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.

Solución.

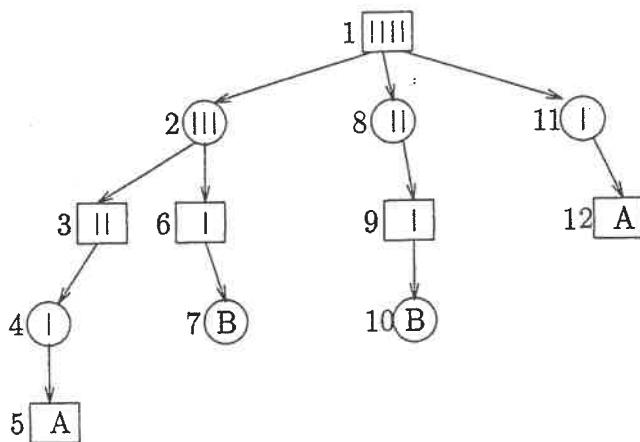


Figura 14.6: Nodos recorridos en el árbol del ejercicio 14.2.

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 alfa valor del padre.

La evaluación del valor en un nodo donde mueve A puede ser:

```
operación A(x:nodo;l:nivel;β:valor): valor
  var v,α:valor;i:entero
  α:=-∞
  si terminal(x) O l=0 entonces
    α:=valor(x)
  sino
    i:=1
    mientras existehijo(x,i) Y α < β hacer
      v:=B(hijo(x,i),l-1,α)
      si α < v entonces
        α:=v
      finsi
      i:=i+1
    finmientras
  finsi
  devolver α
```

Y en un nodo donde mueve B:

```
operación B(x:nodo;l:nivel;α:valor): valor
  var v,β:valor;i:entero
  β:=∞
  si terminal(x) O l=0 entonces
    β:=valor(x)
  sino
    i:=1
```

	1α	2β	3α	4β	5	6α	7	8β	9α	10	11β	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

Tabla 14.2: Alfa y beta valores en el recorrido del árbol del ejercicio 14.2.

mientras existehijo(x, i) $\vee \beta > \alpha$ **hacer**

$v := A(\text{hijo}(x, i), i+1, \beta)$

si $\beta > v$ **entonces**

$\beta := v$

finsi

$i := i + 1$

finmientras

finsi

devolver β

Las funciones tienen los siguientes significados:

- **terminal**, devuelve verdadero 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 verdadero si el nodo tiene hijo i -ésimo.
- **hijo**, devuelve el hijo i -ésimo del nodo actual.

Habrá que hacer una primera llamada con $A(1,l,\infty)$ para decidir la esperanza de ganar empezando a mover. Si lo que queremos es decidir qué movimiento hacer sería:

```

mover:=0
i:=1
mientras existehijo(x,i) hacer
    v:=B(hijo(x,i),l, $\alpha$ )
    si  $\alpha < v$  entonces
         $\alpha := v$ 
        mover:=i
    finsi
    i:=i+1
finmientras
```

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 habrá que volver a ejecutar el mismo código anterior para decidir el siguiente movimiento.

Ejercicios propuestos

Ejercicio 14.4 Dado el árbol de juegos de la figura 14.7, donde se muestra los valores de utilidad en los nodos terminales:

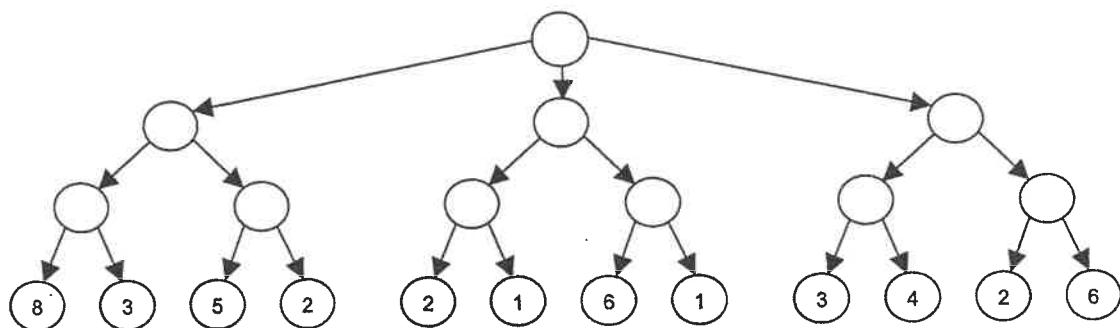


Figura 14.7: Árbol de juego del ejercicio 14.4.

Los valores de utilidad están dados desde el punto de vista del jugador A, que es el que empieza a mover.

- Indicar los valores que se obtienen en cada uno de los nodos utilizando el método minimax.

- b) Analizar el recorrido, incluyendo los valores alfa y beta en cada nodo y los nodos que se podan, cuando se utiliza el método alfa-beta realizando el recorrido de izquierda a derecha.
- c) Analizar el recorrido, incluyendo los valores alfa y beta en cada nodo y los nodos que se podan, cuando se utiliza el método alfa-beta realizando el recorrido de derecha a izquierda.

Ejercicio 14.5 Repetir el problema anterior suponiendo que los beneficios están dados desde el punto de vista de A pero empieza moviendo el jugador B. ¿Cuál es la diferencia respecto al caso anterior?

Ejercicio 14.6 Consideramos el juego del tres en raya, consistente en dos jugadores que ponen alternativamente una cruz y un círculo en un tablero 3×3 (el jugador A pone una cruz y el B un círculo, por lo que el A puede llegar a poner 5 cruces y el B un máximo de 4 círculos). Gana el primer jugador que sitúa tres de sus símbolos en la misma fila, columna o diagonal, y el juego puede acabar en empate. Calcular el número de nodos del árbol de este juego.

- a) Indicar alguna estrategia para reducir el número de nodos que se recorren.
- b) Suponiendo que se decide no recorrer todo el árbol, sino llegar hasta un cierto nivel, indicar alguna función de utilidad que se pudiera usar en los nodos no terminales. Justificar la elección realizada.

Ejercicio 14.7 Diseñar un procedimiento minimax no recursivo.

Ejercicio 14.8 Considerar el siguiente juego. Inicialmente tenemos varios montones de palillos, cada uno de los cuales tiene un número conocido de palillos. Por ejemplo, si llamamos m al número de montones, el número inicial de palillos en cada uno sería (p_1, p_2, \dots, p_m) . En el juego participan dos jugadores, A y B, que mueven de forma alternativa. En cada movimiento un jugador puede eliminar uno o más palillos, pero siempre del mismo montón. Pierde el jugador que quite el último palillo (o, equivalentemente, el que se quede con un solo palillo). Se pide:

- a) Resolver el juego utilizando la estrategia minimax y la poda alfa-beta. Estudiar el problema y diseñar un algoritmo para resolverlo, especificando de forma precisa su funcionamiento. Se entiende que el objetivo del problema es: dada una situación inicial del juego, decidir el movimiento que debe hacer el jugador que empieza a jugar para ganar.
- b) Aplicar el algoritmo diseñado sobre los casos mostrados en la figura 14.8. Decir quién gana el juego de los dos jugadores y los movimientos que debe hacer para lograrlo. ¿Cómo se puede aprovechar el resultado obtenido en un caso para ahorrar cálculos en los otros?

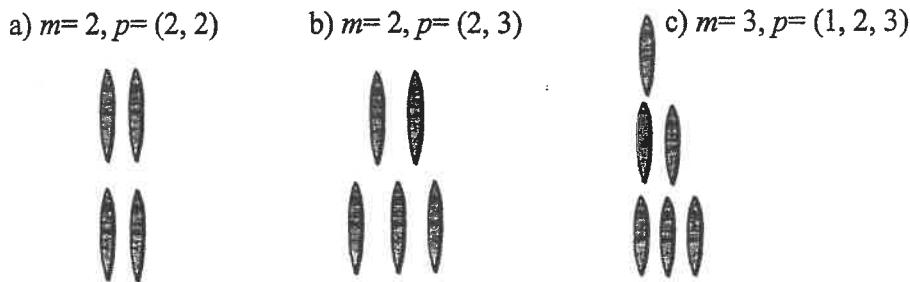


Figura 14.8: Posibles situaciones de partida del juego de los palillos, del ejercicio 14.8.

Ejercicio 14.9 El juego de las 4 en raya se juega sobre un tablero de tamaño 8×8 . A diferencia de las 3 en raya, existe un efecto de *gravedad*: las fichas se colocan en una columna y caen hasta la primera fila libre de esa columna. Gana el jugador que antes consigue colocar 4 de sus fichas en línea. Programar un algoritmo para resolver el problema, utilizando las técnicas de resolución de juegos. ¿Cuántos nodos tendrá el árbol de juegos si es generado hasta nivel k ? En función del resultado, ¿cuánto será el orden de complejidad del algoritmo? Estimar una cota superior del número de nodos del árbol completo.

Ejercicio 14.10 Modificar el procedimiento alfa-beta del algoritmo 14.2 para restringir la exploración hasta un cierto nivel de profundidad.

Ejercicio 14.11 Suponer la versión del juego de las tres en raya en la que cada jugador sólo tiene tres fichas. Si ya tiene colocadas las tres fichas en el tablero, entonces en el siguiente movimiento deberá quitar una ficha para ponerla en otra posición distinta. ¿Qué inconvenientes presenta esta variante del juego? Mostrar la forma de resolver el problema utilizando la estrategia minimax con poda alfa-beta.

Cuestiones de autoevaluación

Ejercicio 14.12 Dar alguna idea para la modificación de los procedimientos minimax y alfa-beta para juegos con tres jugadores, que mueven alternativamente y que intentan ganar. Analizar su funcionamiento con el juego del tres en raya, en un tablero 4×4 , donde los jugadores A, B y C escriben alternativamente una cruz, un círculo y una raya, y gana el jugador que primero coloca tres de sus símbolos en posiciones adyacentes en la misma fila, columna o diagonal.

Ejercicio 14.13 Dar alguna idea para la modificación de los procedimientos minimax y alfa-beta para juegos con dos jugadores, que mueven alternativamente y que intentan ganar, pero que incluyen azar. Analizar su funcionamiento con el juego del tres en raya en el que, cuando un jugador tiene que poner, por cada casilla libre del tablero se decide con probabilidad $\frac{1}{2}$ si se puede poner (puede darse el caso de que un jugador no pueda poner habiendo casillas libres).

Ejercicio 14.14 En el esquema de algoritmo minimax con poda alfa-beta, puesto que existe propagación de valores de los hijos a los padres, se utilizó un procedimiento recursivo. Debido a esto, ¿cómo es, implícitamente, la estrategia de ramificación en el árbol?

Propón una estructura de representación que nos permita utilizar un algoritmo no recursivo. Determinar qué datos se deben almacenar para cada nodo, qué nodos se encontrarán en la lista de nodos vivos, cómo se realiza la propagación de valores y cuándo se saca un nodo de la lista.

Ejercicio 14.15 En un árbol de juegos tenemos valores de utilidad 1, -1 o 0, en caso de victoria de A, derrota o empate, respectivamente. Suponiendo que el jugador aplica la estrategia minimax sobre su situación actual y obtiene un -1 en la raíz, ¿significa que el jugador A siempre pierde la partida? ¿Por qué? Suponiendo que obtiene un valor 1, ¿tiene garantizada la victoria? ¿Por qué?

Ejercicio 14.16 Un juego de tablero es *injusto* si un jugador tiene garantizada la victoria o la derrota por el hecho de empezar a mover o no. Por ejemplo, el juego de los palos no es justo, porque con 5 palos siempre pierde el que empieza moviendo. En un juego injusto con derrota, la estrategia minimax produciría un empate entre valores -1 en la raíz. ¿Cómo se podría decidir en este caso por un movimiento u otro, suponiendo que el humano no siempre jugará de forma óptima?

Referencias bibliográficas

El tema de los árboles de juegos se estudia en muchos libros de algoritmos o estructuras de datos, algunas veces junto con la técnica de backtracking ([Aho88], [Campos95], [Weis95]), otras dentro de técnicas de recorrido de árboles ([Horowitz82], [Tenenbaum88], [Kruse89], [Langsam97]) o de grafos ([Brassard90]), y en pocos casos en un capítulo exclusivo sobre juegos ([Weis00]).

Los esquemas utilizados en este capítulo coinciden básicamente con los de [Aho88].

Capítulo 15

Complejidad algorítmica

En los capítulos anteriores hemos estudiado el coste de los algoritmos propuestos y comparado sus prestaciones, y hemos visto ejemplos de problemas para los que teníamos algoritmos de distintos costes. De esta manera, para un problema concreto podemos tener un algoritmo del cual podemos estudiar su tiempo, pero no sabemos cuál es la dificultad intrínseca del problema que resuelve. Es decir, aunque tengamos un algoritmo de coste $\Theta(n^3)$, ¿es posible que haya un algoritmo de coste menor para resolver el problema? En este capítulo estudiaremos algunas nociones básicas de la complejidad de problemas.

Objetivos del capítulo:

- Comprender la diferencia entre complejidad de algoritmos y de problemas.
- Conocer algunas técnicas básicas de estudio de complejidad y de equivalencia de problemas.
- Conocer la existencia de una jerarquía de clases de problemas.
- Entender la importancia de la heurística y las soluciones aproximadas en la solución de problemas de alto coste.

Contenido del capítulo:

15.1. Complejidad de problemas	241
15.1.1. Concepto de cota inferior de un problema	241
15.1.2. Métodos para la obtención de la cota inferior de un problema	242
15.2. Equivalencia de problemas	245
15.2.1. Reducción entre problemas matriciales	246
15.3. Clasificación de problemas	247
15.3.1. Problemas P y NP	248
15.3.2. Soluciones heurísticas y aproximadas	248
Ejercicios resueltos	249
Ejercicios propuestos	250
Cuestiones de autoevaluación	251
Referencias bibliográficas	251

15.1. Complejidad de problemas

15.1.1. Concepto de cota inferior de un problema

En el análisis de algoritmos nos hemos encontrado con algoritmos de distinto coste incluso para solucionar un mismo problema. Por ejemplo, tenemos algoritmos de ordenación con coste $\Theta(n^2)$ (métodos directos) y con coste $\Theta(n \log n)$ (ordenación por mezcla y ordenación rápida en el caso promedio); y para la multiplicación de matrices tenemos coste $\Theta(n^3)$ en el método directo y $\Theta(n^{2,81})$ en el divide y vencerás. Se nos plantea la pregunta: ¿podemos encontrar algoritmos más rápidos asintóticamente? Es decir, ¿se puede diseñar un algoritmo de coste menor a $n \log n$ para la ordenación o menor a $n^{2,81}$ para la multiplicación de matrices? O incluso aunque no seamos capaces de diseñar esos algoritmos, ¿existen? De estas cuestiones se ocupa la disciplina de la **Complejidad Computacional**.

Dado un problema, nos interesa saber la dificultad intrínseca de resolverlo, que es el coste mínimo de cualquier algoritmo que lo resuelva. Se trata de encontrar una función $f(n)$ de manera que, para cualquier algoritmo con coste $g(n)$ que resuelva el problema, sea $g(n) \in \Omega(f(n))$. Esta función $f(n)$ será una **cota inferior de la complejidad del problema**. Nos interesa obtener la mayor de entre todas las cotas inferiores, con lo que sabríamos la complejidad del problema. Que una de esas cotas inferiores es la mayor se puede obtener si se encuentra un algoritmo cuyo coste coincide con la cota inferior del problema. Es decir, si el problema tiene cota $\Omega(f(n))$ y hay un algoritmo con orden $\Theta(f(n))$, podemos concluir que la complejidad del problema es $f(n)$.

El problema de ordenación por comparaciones se sabe que tiene complejidad $n \log n$, pues se puede demostrar que esta función es cota inferior de la complejidad del problema, y al mismo tiempo se conocen algoritmos con ese coste.

Sin embargo, de la multiplicación de matrices no se sabe su complejidad, pues se conocen cotas inferiores (por ejemplo n^2), pero no coinciden con el coste de ningún algoritmo conocido. Lo más que se puede saber en este caso es que la complejidad del problema está entre la mayor cota inferior obtenida y el coste del algoritmo más rápido conocido. Aunque hemos visto un algoritmo de coste $n^{2,81}$, hay algoritmos de menor coste¹. Por tanto, hablamos de complejidad de problemas en términos asintóticos, pues la dificultad de resolución de un problema de tamaño reducido es siempre reducida, aunque se utilice un algoritmo de coste exponencial.

Como nos podemos imaginar de lo comentado, es mucho más complicado estudiar la complejidad de un problema que la de un algoritmo. En el caso de un algoritmo tenemos algo concreto que estudiar, el algoritmo, pero en el caso de un problema puede haber muchos algoritmos para resolverlo, conocidos y no conocidos. Entonces, ¿cómo se estudia la complejidad de un problema? No hay métodos generales y, de hecho, no se conoce la complejidad de un gran número de problemas muy comunes (por ejemplo, la multiplicación de matrices).

¹Aunque las constantes que los afectan y las necesidades de memoria hacen que el método de Strassen sea preferible para los tamaños de matrices que normalmente se quiere resolver y los sistemas de que disponemos en la actualidad para resolverlos.

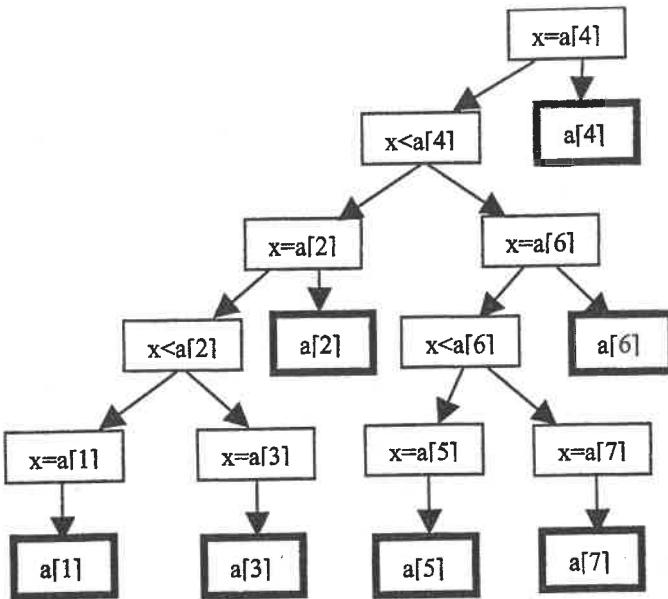


Figura 15.1: Árbol de decisión para la búsqueda binaria en array, con 7 datos.

15.1.2. Métodos para la obtención de la cota inferior de un problema

Aunque no hay métodos generales para obtener la complejidad de un problema, vamos a ver algunas de las ideas básicas para obtener cotas inferiores.

Árboles de decisión Esta técnica se puede utilizar para problemas de comparación cuya solución se puede representar por un **árbol de decisión**, que es un árbol binario donde cada nodo hoja representa una solución y cada nodo interno una decisión de dos posibles, al ser el árbol binario². La profundidad del árbol de decisión asociado al problema es una cota inferior de la complejidad del problema, pues una solución puede llegar a tener tiempo de ejecución (número de comparaciones) igual a la profundidad del árbol. Vemos dos ejemplos de aplicación de esta técnica.

Ejemplo 15.1 Complejidad del problema de búsqueda binaria en array ordenado.

En la figura 15.1 se muestra un árbol de decisión para el problema de búsqueda binaria de un dato que se encuentra en un array ordenado, con $n = 7$. En cada paso se compara con el elemento central del trozo de array donde se está buscando. Si coincide se ha encontrado el elemento, y si no se busca en la zona izquierda o derecha según el resultado de otra nueva comparación.

Por simplicidad consideramos $n = 2^k - 1$, y una vez obtenida la complejidad del problema se puede quitar la restricción. Puede que con una única comparación encontremos

²Podrían considerarse árboles de decisión no binarios, pero esto no modificaría el coste de los problemas.

el elemento ($x = a[4]$), pero lo que nos da la complejidad del problema es el caso más desfavorable del tiempo de ejecución de cualquier algoritmo que lo resuelve, que viene dado por la altura del árbol. En los niveles impares (el nodo raíz está en el nivel 1) no tenemos nodos terminales, y en los pares tenemos 2^{i-1} nodos terminales en el nivel $2i$, por lo que la altura del árbol es $2k$, y el número de comparaciones $2k - 1$. Por tanto, el problema tiene cota inferior $\Omega(\log n)$. Como la búsqueda binaria tiene cota superior $O(\log n)$, concluimos que el coste del problema es $\log n$.

Ejemplo 15.2 Complejidad de la ordenación por comparación.

La ordenación de una serie de datos se puede hacer por métodos basados en transformaciones (por ejemplo, almacenando los datos en una tabla con alguna técnica hashing y después recorriendo la tabla para obtener los datos ordenados) o basados en comparaciones. En el caso de los métodos basados en comparaciones se puede obtener el coste del problema utilizando árboles de decisiones.

Dados n datos, pueden estar ordenados de $n!$ maneras distintas, por lo que el árbol de decisión asociado tendrá $n!$ nodos terminales. Sea k tal que $2^{k-1} < n! \leq 2^k < (n+1)!$, con lo que $k-1 < \log n! \leq k < \log(n+1)! = \log(n+1) + \log n!$. Como $\Theta(\log n!) = \Theta(\log(n+1) + \log n!)$, tenemos que $\Theta(\log n!) = \Theta(k)$, y la altura mínima de un árbol de decisión con 2^k nodos terminales es $k+1$, con lo que la complejidad del problema es $\log n!$.

Comprobamos que $\Theta(\log n!) = \Theta(n \log n)$. Por un lado

$$\log n! = \sum_{i=1}^n \log i \leq \sum_{i=1}^n \log n = n \log n$$

con lo que $\log n! \in O(n \log n)$. Por otro lado,

$$\log n! = \sum_{i=1}^n \log i \geq \sum_{i=\frac{n}{2}}^n \log i \geq \sum_{i=\frac{n}{2}}^n \log \frac{n}{2} = \frac{n}{2} \log \frac{n}{2} \in \Theta(n \log n)$$

con lo que $\log n! \in \Omega(n \log n)$. Al cumplirse las dos inclusiones tenemos que $\Theta(\log n!) = \Theta(n \log n)$. Por tanto, tenemos una cota inferior del problema, $n \log n$. Pero además sabemos que hay algoritmos con ese coste, por lo que $n \log n$ es la complejidad del problema.

Método del oráculo Este método también se llama del **adversario**. Consiste en suponer que tenemos un **oráculo** (o un adversario) que, ante una pregunta dada, responde siempre con la respuesta que nos lleva a un subproblema de mayor coste, pero siendo la respuesta posible para alguna entrada del problema. Como para una ejecución particular podríamos tener la entrada que nos produce esa serie de respuestas, la complejidad del problema está acotada inferiormente por el número de respuestas generadas.

Ejemplo 15.3 Consideramos de nuevo el problema de encontrar un dato en un array ordenado.

Ante una pregunta del tipo $x = a[i]$ la respuesta sería “no”, y ante una del tipo $x < a[i]$ sería la que nos llevara a buscar a un subarray de mayor tamaño. En el caso

de $n = 2^k - 1$, cualquiera de las dos posibles respuestas valdría pues los dos subarrays a izquierda y derecha tienen el mismo tamaño. Cada dos comparaciones pasamos de un problema de tamaño $2^s - 1$ a otro de tamaño $2^{s-1} - 1$, por lo que tenemos la ecuación $t(2^k - 1) = 2 + t(2^{k-1} - 1) = 2(k - 1) + t(1) = 2k - 1 \in \Theta(\log n)$.

Otra posibilidad es utilizar el oráculo para encontrar una entrada que constituya un contraejemplo para una resolución determinada. Es decir, si suponemos que se da una cierta respuesta en un número determinado de pasos, encontrar una entrada con la que en ese número de pasos la respuesta es errónea.

Ejemplo 15.4 Obtención del elemento mayor en un array.

Sabemos que se puede obtener el mayor elemento de un array con n datos haciendo $n - 1$ comparaciones. Si conjeturamos que se puede hacer con menos de $n - 1$ comparaciones, digamos con $n - 2$, en cada comparación $a[i] < a[j]$ habrá un “perdedor” (el índice correspondiente al menor valor). Dado que hacemos $n - 2$ comparaciones no puede haber más de $n - 2$ perdedores. Uno de ellos es el índice que se daría como respuesta (k), pero habrá otro índice l que no ha perdido nunca. El oráculo puede producir la entrada $a[i] = i, \forall i \neq k, i \neq l, a[k] = n + 1, a[l] = n + 2$, que con $n - 2$ comparaciones puede dar efectivamente respuesta k , pero es errónea al ser el máximo $a[l]$.

El número mínimo de comparaciones es $n - 1$, y tenemos un algoritmo con ese número de comparaciones, por lo que la complejidad del problema es n .

Conteo de entradas y salidas Para resolver un problema hay que leer una serie de datos de entrada y producir unos datos de salida. El volumen de estos datos nos da una cota inferior de la complejidad del problema, pues al menos hay que leerlos y escribirlos. Vemos algunos ejemplos.

Ejemplo 15.5 Obtención del mayor elemento en un array.

En la entrada hay que leer los n elementos del array, y la salida consiste en decir un único elemento. El coste es por lo menos n , y al haber un algoritmo de coste n esa es la complejidad del problema.

Ejemplo 15.6 Producto de dos matrices.

Para multiplicar dos matrices de tamaño $n \times n$ hay que leer $2n^2$ datos, por lo que una cota inferior de la complejidad es n^2 .

Conocemos un algoritmo de coste $n^{2.81}$, por lo que no sabemos la complejidad del problema, pero sí que va a estar entre n^2 y $n^{2.81}$.

Para este problema se conocen algoritmos de coste asintótico menor que el del algoritmo de Strassen. En particular hay algoritmos de orden $n^{2.376}$. Por tanto, se aproximan más las cotas de la complejidad del problema, pero en la actualidad no se conoce esta complejidad, por lo que está abierta la posibilidad de encontrar algoritmos más rápidos (asintóticamente) que los conocidos.

Ejemplo 15.7 Producto de dos enteros largos.

Para multiplicar dos enteros largos de longitud n hay que leer $2n$ datos, por lo que una cota inferior de la complejidad es n .

Conocemos un algoritmo de coste $n^{1.59}$, por lo que no sabemos la complejidad del problema, pero sí que está entre n y $n^{1.59}$.

Hay otros algoritmos de menor coste, como uno de Schöngaehe y Strassen, de coste $n(\log n)^3$.

Ejemplo 15.8 Producto exterior de vectores.

El producto exterior de dos vectores u y v , de tamaño n , consiste en obtener la matriz A de dimensión $n \times n$, donde $A = (u_i v_j)$. En este caso la entrada consiste en $2n$ datos y la salida en n^2 . Tenemos que n^2 es una cota inferior de la complejidad. Además el algoritmo obvio de multiplicar cada elemento de u por cada uno de los de v (es la definición de producto exterior) tiene coste n^2 . Por tanto, la complejidad del problema es n^2 .

15.2. Equivalencia de problemas

Para la mayoría de los problemas no se conoce su complejidad, pero es posible en algunos casos demostrar que varios problemas tienen la misma complejidad, con lo que una mejora en cuanto al conocimiento de la complejidad de uno de los problemas conlleva la mejora correspondiente en la complejidad del problema equivalente.

Dados dos problemas, P_1 y P_2 , se dice que P_1 es **linealmente reducible** a P_2 si la existencia de un algoritmo para P_2 con coste $O(f(n))$ implica la existencia de otro algoritmo para P_1 con el mismo orden. Se denota $P_1 \leq^l P_2$.

Decimos que dos problemas, P_1 y P_2 , son linealmente equivalentes cuando $P_1 \leq^l P_2$ y $P_2 \leq^l P_1$. Se denota $P_1 \equiv P_2$.

Ejemplo 15.9 No conocemos la complejidad de la multiplicación de enteros largos (MEL), pero se puede demostrar que este problema es linealmente equivalente al cálculo del cuadrado de un entero largo (CEL).

Un algoritmo de multiplicación se puede utilizar para calcular el cuadrado, por lo que la complejidad de calcular el cuadrado no puede ser mayor que la de la multiplicación, y $CEL \leq^l MEL$.

Por otro lado, de la fórmula:

$$xy = \frac{(x+y)^2 - (x-y)^2}{4}$$

se deduce que la multiplicación de dos enteros se puede calcular con sumas y restas de enteros largos, que tienen coste lineal, menor que la multiplicación de enteros (ejemplo 15.7), división por 4, que tiene coste entre constante y lineal³, y elevando al cuadrado dos enteros largos.

El entero $x + y$ puede tener tamaño $n + 1$, con n el tamaño de los enteros x e y . Por eso, sólo podríamos concluir que la existencia de un algoritmo de coste $O(f(n))$ para CEL implica la de uno de coste $O(f(n+1))$ para MEL.

En este caso es fácil demostrar la equivalencia, pues si consideramos que $x + y = a10^n + z$, con a de un dígito y z de tamaño n , se puede obtener $(x + y)^2 = a^210^{2n} +$

³Dependiendo de cómo se implemente.

$2a10^n z + z^2$. Como todas las operaciones, salvo la z^2 , tienen coste constante o lineal, se deduce que $(x+y)^2$ se obtiene con el mismo coste que z^2 . Concluimos que $MEL \leq^l CEL$, y por tanto $CEL \equiv MEL$.

En el ejemplo anterior hemos visto que algunas veces aparece el coste de un algoritmo que resuelve cierto problema de tamaño n como coste de un algoritmo para otro problema, pero en ese caso no de tamaño n sino de otro tamaño mayor (en el ejemplo $n+1$). En estos casos no se puede deducir directamente que los problemas sean equivalentes.

Ejemplo 15.10 Consideramos los problemas de resolver un sistema de ecuaciones lineales (SEL), $Ax = b$, con A una matriz invertible de tamaño $n \times n$, y de invertir una matriz cuadrada (IMC) también de tamaño $n \times n$.

$x = A^{-1}b$, y la multiplicación matriz vector tiene coste n^2 , que es una cota inferior de SEL porque hay que leer al menor los n^2 elementos de A . Por tanto $SEL \leq^l IMC$.

La matriz inversa de A se puede obtener resolviendo la ecuación matricial $AX = I$, con X una matriz $n \times n$, e I la matriz identidad. Si tenemos un algoritmo de coste $f(n)$ para SEL, como A^{-1} se puede obtener resolviendo n ecuaciones lineales, tenemos un algoritmo de coste $O(nf(n))$ para IMC. Esto no quiere decir que los problemas no sean equivalentes, sólo que no hemos sido capaces de probarlo. Lo que sí concluimos es que el coste de IMC no excede en más de n veces el de SEL.

En realidad se sabe mucho más de IMC, pues es equivalente a la multiplicación de matrices cuadradas.

En algunos casos se puede concluir que los problemas son equivalentes haciendo alguna suposición en la forma que tienen las funciones. Por ejemplo, se puede suponer que la función $f(n)$ es eventualmente no decreciente y b -armónica para todo entero $b \geq 2$, lo que es bastante común en las funciones que representan tiempos de ejecución⁴.

15.2.1. Reducción entre problemas matriciales

Como ejemplo de reducciones entre problemas vamos a ver algunos problemas matriciales. Los siguientes problemas son equivalentes bajo ciertas condiciones:

- La multiplicación de matrices cuadradas (MMC).
- La multiplicación de matrices triangulares (MMT). Una matriz $A = (a_{ij})$ es triangular superior si $a_{ij} = 0$, $\forall j < i$, y triangular inferior si es $a_{ij} = 0$, $\forall i < j$.
- La multiplicación de matrices simétricas (MMS). Una matriz $A = (a_{ij})$ es simétrica si $a_{ij} = a_{ji}$, $\forall 1 \leq i, j \leq n$.
- El cálculo de la inversa de una matriz cuadrada no singular (IMC). Que una matriz sea no singular es que tiene inversa, pues no todas las matrices cuadradas la tienen.
- El cálculo de la inversa de una matriz cuadrada triangular no singular (IMT).

⁴Recuerda el teorema 7.1.

Algunas de las reducciones son triviales y no requieren de condiciones adicionales en los problemas. Por ejemplo, $\text{MMT} \leq^l \text{MMC}$ y $\text{MMS} \leq^l \text{MMC}$ sin más que considerar que un algoritmo para multiplicar matrices cuadradas se puede usar para multiplicar matrices triangulares o simétricas.

Las dependencias recíprocas requieren de suposiciones adicionales en los problemas. Definimos los conceptos de "suavidad" en las funciones, algoritmos y problemas.

Definición 15.1 Decimos que una función $f : N \rightarrow R^n$ es **suave** si es eventualmente no decreciente y $f(bn) \in O(f(n))$, \forall entero $b \geq 2$.

Definición 15.2 Un **algoritmo** es suave si su coste es $\Theta(f(n))$, con $f(n)$ una función suave.

Definición 15.3 Un **problema** es suave si el mejor algoritmo que lo resuelve es suave.

Se puede demostrar que $\text{MMC} \leq^l \text{MMT}$ y $\text{MMC} \leq^l \text{MMS}$, si MMT y MMS son problemas suaves.

Para demostrar que $\text{MMC} \leq^l \text{MMT}$ basta con calcular AB en la forma:

$$\begin{pmatrix} 0 & A & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & B \\ 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 & AB \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Con lo que si tenemos un algoritmo de coste $f(n)$ para MMT, hemos obtenido uno de coste $f(3n)$ para MMC. Al ser $f(n)$ suave es $O(f(3n)) \subseteq O(f(n))$, y $\text{MMC} \leq^l \text{MMT}$. De manera similar se obtiene $\text{MMC} \leq^l \text{MMS}$ si MMS es suave.

Suponiendo otras condiciones distintas en los problemas, se pueden demostrar las equivalencias con IMT e IMC. Remitimos a las referencias bibliográficas para el análisis de estas reducciones.

A partir de estas equivalencias se pueden demostrar otras, como la que vemos en el ejemplo siguiente.

Ejemplo 15.11 El problema de resolución de un sistema matricial lineal (SML) $AX = B$ de matrices cuadradas es equivalente al de IMC y MMC en las mismas condiciones en que estos dos son equivalentes.

Si tenemos un algoritmo para resolver el sistema matricial $AX = B$, aplicándolo a $AX = I$ se obtiene A^{-1} , por lo que $\text{IMC} \leq^l \text{SML}$.

Por otro lado, como se puede obtener $X = A^{-1}B$, en las condiciones en que IMC y MMC sean equivalentes, se obtiene un algoritmo de coste $f(n)$ para IMC y MMC, aplicando primero el de IMC a A y después el de MMC a A^{-1} y B , se resuelve $AX = B$. El coste de aplicar los dos algoritmos uno tras otro coincide con el máximo de los dos costes, por lo que el algoritmo para resolver $AX = B$ tiene el mismo coste $f(n)$ y $\text{SML} \leq^l \text{IMC}$.

15.3. Clasificación de problemas

Al estudiar un problema, obteniendo una cota inferior de su complejidad o encontrando otro problema al que es equivalente, no sabemos todavía la complejidad intrínseca del problema, y por lo tanto no sabemos la eficiencia del mejor algoritmo que lo resuelve.

Existen multitud de problemas para los que no se conoce un algoritmo “eficiente” que los resuelva, pero de los que tampoco se conoce la dificultad de su resolución. Algunos de los problemas vistos en el libro son de este tipo: el problema del viajante, el de la coloración de grafos, el de la mochila, el del camino hamiltoniano, etc. Que no se conozcan algoritmos eficientes para resolverlos, ¿quiere decir que no existen esos algoritmos? Primero deberíamos determinar a qué llamamos un algoritmo eficiente. Diremos que **un algoritmo es eficiente**, o polinómico, si tiene tiempo de ejecución polinómico.

El decir que un algoritmo es eficiente según la definición anterior no dice mucho sobre su “eficiencia” práctica, pues no es lo mismo que se pueda resolver en tiempo n^{100} que en tiempo n^2 . También hay que tener en cuenta las constantes, que pueden hacer que un algoritmo de coste exponencial sea preferible a uno de coste polinómico para tamaños de problema no muy grandes y que quizás sean los tamaños de problema que se necesita resolver.

En el conjunto de todos los problemas se definen clases para determinar la dificultad de resolución según la eficiencia del mejor algoritmo que los resuelve. En el caso de los problemas nombrados antes no se sabe si hay algoritmos eficientes para resolverlos, pero sí se sabe que si existe un algoritmo de coste polinómico para uno de ellos existe para los demás. Por otro lado, debido a la gran cantidad de trabajo dedicado a la resolución de estos problemas, y a no haberse encontrado algoritmos eficientes para ellos, se piensa que no existen esos algoritmos y que los problemas tienen un coste intrínseco mayor al polinómico.

15.3.1. Problemas P y NP

Para la clasificación de problemas se estudian problemas de decisión, que son aquellos en los que la respuesta es sí o no. Se obtienen dos clases básicas, la P y la NP.

Definición 15.4 Se llama P a la clase de los problemas de decisión que se pueden resolver con un algoritmo de tiempo polinómico.

Definición 15.5 La clase NP es la de los problemas para los que comprobar si una determinada posible solución es en realidad solución se puede hacer en tiempo polinómico.

Por lo tanto, para los problemas NP verificar una solución es rápido, aunque encontrarla puede que sea muy costoso.

También se suele definir NP como la clase de los problemas que se pueden resolver en tiempo polinómico por un algoritmo no determinista.

Algunas veces se confunde la clase NP con la de los problemas cuyo coste es “No Polinomial”. Como hemos indicado, no es ese el significado de “NP”, sino el de “Polinomial No determinista”.

Se puede comprobar que $P \subseteq NP$, pero no se sabe si $P = NP$ o no, aunque se piensa que $P \neq NP$. Esta es la gran cuestión pendiente de la informática teórica.

15.3.2. Soluciones heurísticas y aproximadas

Para los problemas que tienen un alto coste computacional, o no se conocen algoritmos para resolverlos con un coste razonable, puede ser preferible obtener una solución

aproximada o cercana a la óptima utilizando un algoritmo eficiente.

Una posibilidad es utilizar **algoritmos heurísticos**. De este tipo vimos los algoritmos de avance rápido, que son normalmente rápidos (en tiempo de ejecución) y en los que se toma una serie de decisiones basándonos en el conocimiento heurístico que tenemos del problema, lo que puede llevarnos a una solución óptima o a una muy alejada de la óptima.

En algunos casos en que se tiene un algoritmo polinómico para resolver un problema se puede preferir resolverlo por un método aproximado más rápido.

Ejemplo 15.12 El problema de la mochila 0/1, con capacidad de la mochila M y n objetos se puede resolver en un tiempo $\Theta(nM)$ por programación dinámica. Si M es muy grande puede ser preferible (por motivos de tiempo de ejecución o de ocupación de memoria) resolverlo por un método aproximado. Por ejemplo, se podría aplicar el esquema de avance rápido para el problema no 0/1, pero no incluyendo el trozo del último objeto que se estudia si este no cabe entero.

En los algoritmos aproximados nos interesa tener una cota de la diferencia entre el valor de la solución que se obtiene y la óptima. Si la cota es conocida y no demasiado grande, el algoritmo aproximado puede tener interés, pero en caso contrario no.

Ejemplo 15.13 En el problema de la mochila 0/1 se puede obtener una cota, pues sabemos que una cota inferior se obtiene realizando un avance rápido 0/1, y una superior con un avance rápido no 0/1 y tomando la parte entera.

Pero esta cota puede ser tan grande como queramos. Si consideramos capacidad de la mochila M , y dos objetos de beneficios $b = (2, M)$ y pesos $p = (1, M)$, la cota inferior es 2 (se cogería sólo el primer objeto), y la superior $M + 1$ (cogiendo el objeto 1 y $\frac{M}{M-1}$ partes del objeto 2).

También se usa la heurística en algoritmos tipo ramificación y poda, y se puede usar en los de backtracking para guiar la búsqueda. Esto no haría que redujéramos el orden de complejidad, ya que el número de nodos del árbol de búsqueda sigue siendo el mismo, aunque puede que sí redujera el tiempo de ejecución. Dado que estos algoritmos pueden tener un coste muy alto, puede ser preferible dar una solución después de un cierto tiempo aunque ésta no sea óptima. Lo mismo ocurre con el avance rápido, donde se puede empezar una búsqueda local (más o menos amplia según el tiempo de que dispongamos) después de la solución obtenida por avance rápido.

Ejercicios resueltos

Ejercicio 15.1 Llamamos MMTD al problema de multiplicar dos matrices tridiagonales, donde una matriz $T = (t_{ij})$ es tridiagonal si $t_{ij} = 0 \forall i, j$ tal que $|j - i| > 1$. Y llamamos MMCTD al problema de multiplicar una matriz cuadrada $n \times n$ por otra tridiagonal.

- ¿Cuál es la complejidad de MMTD?
- ¿Cuál es la complejidad de MMCTD?

Solución.

- Tenemos una cota inferior de orden n pues hay que leer los n datos de la diagonal y los $2n - 2$ de la primera super y subdiagonal. Por otra parte, el algoritmo de multiplicación

tiene en este caso coste n , pues la matriz resultante de la multiplicación es pentadiagonal, y cada elemento se obtiene con un máximo de tres multiplicaciones y dos sumas. La complejidad del problema es, por tanto, n .

b) La cota inferior es n^2 , ya que hay que leer los datos de la matriz cuadrada no tridiagonal. Y cada uno de los n^2 elementos de la matriz resultante se obtiene con un máximo de tres multiplicaciones y dos sumas. La complejidad del problema es por lo tanto n^2 .

Ejercicio 15.2 Dar cotas inferior y superior de la complejidad espacial y temporal del problema de elevar una matriz $n \times n$ a n .

Solución.

Si representamos n en forma binaria $n = n_k \dots n_1 n_0$ (lo que se puede obtener en orden $\log n$), es $A^n = \prod_{i=0}^k (A^{2^i} n_i + I_n (1 - n_i))$. Cada $A^{2^i} = A^{2^{i-1}} A^{2^{i-1}}$, con lo que para calcular A^{2^i} hay que hacer $\log i$ multiplicaciones de orden el que tenga la multiplicación de matrices. Además, para obtener los productos (\prod) hay que hacer como mucho $\log n$ multiplicaciones. Así, $n^{2,376} \log n$ es una cota superior, pero no sabemos si $n^2 \log n$ es inferior pues podría ser que se pudiera obtener el resultado con otra fórmula distinta de la que hemos usado.

La complejidad espacial sería en este caso de $n^2 \log n$, si se guardan las potencias que se van calculando, pero esto no es necesario pues sólo hay que guardar la última potencia calculada y la potencia hasta ella. De esta manera la complejidad espacial es n^2 .

Ejercicios propuestos

Ejercicio 15.3 En el ejemplo de la búsqueda binaria en array ordenado (ejemplo 15.1) hemos considerado que se hacía una primera comparación de igualdad para ver si era el elemento buscado y, en caso de no haber encontrado el elemento, una segunda comparación para ver si era menor o mayor. Comprobar que si se hacen comparaciones del tipo menor o igual también se obtiene la misma complejidad.

Ejercicio 15.4 ¿Cuál es la complejidad de obtener ordenados los $\frac{n}{2}$ datos menores de un array de n datos? Suponemos que se utiliza un método basado en comparaciones. Y en general, ¿cuál es la complejidad de obtener ordenados los $\frac{n}{k}$ datos menores de un array de n datos, con k constante?

Ejercicio 15.5 Consideramos el problema de, dado un damero de dimensión $n \times n$ y una serie de piezas formadas por un número fijo de cuadrículas, p , encontrar la mayor cantidad de piezas que se pueden poner en el damero sin que ninguno de sus cuadrados quede fuera del tablero y sin que se solapen dos cuadrados de piezas distintas. Acotar la complejidad de este problema. Indicar algún método heurístico y aproximado para su resolución.

Ejercicio 15.6 Considerando árboles ternarios de decisión (en cada nodo no terminal se hace una comparación que puede dar tres resultados, por ejemplo, si el resultado de la comparación es menor, igual o mayor), demostrar que la complejidad del problema de búsqueda binaria es $n \log n$.

Cuestiones de autoevaluación

Ejercicio 15.7 Contestar si son ciertas o no, justificando la respuesta, cada una de las siguientes afirmaciones:

- a) Todos los problemas para los que existe un algoritmo de programación dinámica para resolverlos pertenecen a P.
- b) Todos los problemas de optimización para los que existe un algoritmo voraz para resolverlos obteniendo el óptimo pertenecen a P.
- c) Los problemas para los que el mejor algoritmo para resolverlos sea de backtracking pertenecen a NP.
- d) Los problemas para los que el mejor algoritmo para resolverlos sea de backtracking no pertenecen a P.

Ejercicio 15.8 Dar cotas inferior y superior de la complejidad espacial y temporal del problema de elevar a n un entero largo de longitud n .

Ejercicio 15.9 Razonar sobre si es linealmente reducible el problema de la mochila 0-0.5-1 al de la mochila 0-1.

Referencias bibliográficas

En este capítulo se dan unas ideas muy generales sobre la complejidad de problemas. Se han seguido principalmente las referencias [Baase83], [Brassard90], [Brassard97] y [Baase00], aunque en estos libros se desarrolla el tema bastante más de lo que se ha hecho en este capítulo. En [Troya84] también se pueden encontrar algunas ideas básicas sobre la clasificación de problemas. En todos estos libros y en el presente capítulo se sigue un enfoque algorítmico. En otros libros ([Aho74], [Cormen90] y [Johnson90]) se trata el tema desde un punto de vista menos algorítmico y más centrado en la teoría de la computación, y se trata la clasificación de problemas a partir de máquinas de Turing y reconocimiento de lenguajes. Ese mismo enfoque se sigue en libros completamente dedicados a Computación, Teoría de Autómatas o Lenguajes Formales, pero ese enfoque correspondería más a otras asignaturas de la carrera. Algunos de estos libros son [Garey79], [Hopcroft79], [Lewis81], [Davis83], [Wood87] y [Hopcroft01].

Apéndices

Apéndice D

Práctica 2: Análisis de algoritmos

En esta práctica se hace hincapié en el uso de las notaciones asintóticas estudiadas en teoría para realizar análisis teórico de algoritmos. En particular se aplicará a algoritmos de ordenación, entre los cuales se incluyen dos algoritmos divide y vencerás, de modo que también se practica esta técnica algorítmica. Tras implementar los distintos algoritmos se procede a un estudio experimental de los mismos y a la comparación de los estudios teórico y experimental. En este capítulo se expone un enunciado real (correspondiente a las prácticas del curso 2002/2003), incluyendo la puntuación de cada apartado, y se dan algunas ideas generales para guiar su realización. En ningún caso ha de tomarse este apéndice como un ejemplo de resolución completa de la práctica.

D.1. Enunciado

Se estudiarán distintos métodos de ordenación por comparación: algún método directo a decidir por el alumno (inserción, selección, burbuja, ...) y los método divide y vencerás *mergesort* y *quicksort* vistos en clase, con tamaño del caso base variable, usándose como solución para el caso base el método directo anterior. Los datos a ordenar serán cadenas de caracteres, que se tomarán de un fichero ASCII que contendrá una cadena por línea. La salida será un fichero del mismo tipo que el de entrada con las cadenas ordenadas en orden alfabético. El estudio debe contener todos y cada uno de los siguientes puntos:

- 1) ESTUDIO TEÓRICO:

- 1.a) (1 punto) Se estudiarán teóricamente los tres métodos de ordenación, estudiándose por separado el número de comparaciones y de asignaciones de elementos del tipo que se está ordenando, así como el orden de la ocupación de memoria. El estudio de comparaciones y asignaciones debe comprender caso más favorable, más desfavorable y promedio: para cada uno de ellos habrá que obtener las notaciones asintóticas estudiadas, Ω , O , Θ y o . El estudio se hará en función del número de cadenas a ordenar, sin tener en cuenta la longitud de las cadenas, aunque en la práctica esta longitud puede influir en el coste de las operaciones básicas de asignación y comparación. En los métodos divide y vencerás se considerará tamaño del caso base uno.

- 1.b) (1 punto) De forma similar al apartado anterior se estudiarán teóricamente los métodos divide y vencerás con tamaño del caso base variable, y se estudiará teóricamente el tamaño óptimo del caso base. Estos estudios se realizarán solamente para el caso promedio del tiempo de ejecución medido como combinación de comparaciones y asignaciones.
- 2) POSIBLES OPTIMIZACIONES: Se considerarán posibles optimizaciones sobre los métodos divide y vencerás estudiados en clase:
 - 2.a) (1 punto) En la ordenación por mezcla se estudiará la posibilidad de evitar copias de datos utilizando una estructura auxiliar de manera que en un paso se mezclen datos de la estructura origen a la auxiliar y en el siguiente paso de la estructura auxiliar a la origen. Se estudiará de manera teórica cómo esta modificación puede influir en el tiempo de ejecución, la ocupación de memoria y el tamaño óptimo del caso base.
 - 2.b) (1 punto) La ordenación rápida tiene un caso más desfavorable de coste cuadrático. Para evitar este caso más desfavorable se puede elegir el pivote tomando una muestra pequeña de los datos, ordenándola y utilizando el elemento central de la muestra ordenada como pivote. ¿Cómo influirá esto en el tiempo de ejecución?
- 3) IMPLEMENTACIONES: (2.5 puntos) Se programará una función de ordenación por cada uno de los tres métodos del apartado 1 y cada una de las optimizaciones del apartado 2, con parámetro de entrada la estructura donde se almacenen los datos leídos del fichero, y con parámetro de salida en la que se guardan los datos antes de escribirlos en un fichero. En los métodos por divide y vencerás un parámetro será el tamaño del caso base. Se explicarán (sin incluir en la documentación los códigos) los detalles de implementación: estructura utilizada para almacenar los datos, función de comparación y de asignación utilizada o programada, funciones del esquema divide y vencerás, etc.
- 4) VALIDACIÓN DE LOS PROGRAMAS: Para validar los programas hay que hacer una serie de pruebas. Para esto habrá que hacer:
 - 4.a) (1 punto) Se hará un programa para permitir comparar los resultados de la ordenación de un mismo fichero mediante dos métodos diferentes a elegir, por ejemplo mediante un programa que nos pregunte por los dos métodos que se quiere comparar y una función que nos compare los resultados obtenidos con los dos métodos. Se trata de comparar las ordenaciones resultantes, no los tiempos empleados en efectuar la ordenación. El profesor realizará pruebas con los ficheros de entrada que considere oportunos, por lo que ni la longitud de las cadenas de entrada ni el número de cadenas a ordenar debe estar limitado por el algoritmo, ni debe aparecer explícitamente como parámetro de entrada del programa.
 - 4.b) (0.5 puntos) El alumno explicará las pruebas que ha realizado para estar convencido de que sus funciones trabajan correctamente.

- 1.b) (1 punto) De forma similar al apartado anterior se estudiarán teóricamente los métodos divide y vencerás con tamaño del caso base variable, y se estudiará teóricamente el tamaño óptimo del caso base. Estos estudios se realizarán solamente para el caso promedio del tiempo de ejecución medido como combinación de comparaciones y asignaciones.
- 2) POSIBLES OPTIMIZACIONES: Se considerarán posibles optimizaciones sobre los métodos divide y vencerás estudiados en clase:
 - 2.a) (1 punto) En la ordenación por mezcla se estudiará la posibilidad de evitar copias de datos utilizando una estructura auxiliar de manera que en un paso se mezclen datos de la estructura origen a la auxiliar y en el siguiente paso de la estructura auxiliar a la origen. Se estudiará de manera teórica cómo esta modificación puede influir en el tiempo de ejecución, la ocupación de memoria y el tamaño óptimo del caso base.
 - 2.b) (1 punto) La ordenación rápida tiene un caso más desfavorable de coste cuadrático. Para evitar este caso más desfavorable se puede elegir el pivote tomando una muestra pequeña de los datos, ordenándola y utilizando el elemento central de la muestra ordenada como pivote. ¿Cómo influirá esto en el tiempo de ejecución?
- 3) IMPLEMENTACIONES: (2.5 puntos) Se programará una función de ordenación por cada uno de los tres métodos del apartado 1 y cada una de las optimizaciones del apartado 2, con parámetro de entrada la estructura donde se almacenen los datos leídos del fichero, y con parámetro de salida en la que se guardan los datos antes de escribirlos en un fichero. En los métodos por divide y vencerás un parámetro será el tamaño del caso base. Se explicarán (sin incluir en la documentación los códigos) los detalles de implementación: estructura utilizada para almacenar los datos, función de comparación y de asignación utilizada o programada, funciones del esquema divide y vencerás, etc.
- 4) VALIDACIÓN DE LOS PROGRAMAS: Para validar los programas hay que hacer una serie de pruebas. Para esto habrá que hacer:
 - 4.a) (1 punto) Se hará un programa para permitir comparar los resultados de la ordenación de un mismo fichero mediante dos métodos diferentes a elegir, por ejemplo mediante un programa que nos pregunte por los dos métodos que se quiere comparar y una función que nos compare los resultados obtenidos con los dos métodos. Se trata de comparar las ordenaciones resultantes, no los tiempos empleados en efectuar la ordenación. El profesor realizará pruebas con los ficheros de entrada que considere oportunos, por lo que ni la longitud de las cadenas de entrada ni el número de cadenas a ordenar debe estar limitado por el algoritmo, ni debe aparecer explícitamente como parámetro de entrada del programa.
 - 4.b) (0.5 puntos) El alumno explicará las pruebas que ha realizado para estar convencido de que sus funciones trabajan correctamente.

■ 5) ESTUDIO EXPERIMENTAL Y COMPARACIÓN CON TEÓRICO:

- 5.a) (1 punto) (estudio experimental) Para estudiar experimentalmente los distintos métodos se realizará un programa para obtener tiempos de ejecución (se puede usar la función `gettimeofday`) de los distintos métodos implementados, variando el tamaño de la entrada, y del caso base en los métodos divide y vencerás, generando las entradas de manera aleatoria o cercanas al caso más desfavorable, y distintos costes de la comparación, lo que se puede hacer variando la longitud de las cadenas de caracteres. Se obtendrán resultados experimentales para valores significativos (los que son valores significativos lo tiene que decidir y justificar el alumno) y se compararán los distintos métodos en los diferentes casos analizados.
- 5.b) (1 punto) (comparación con teórico) Se contrastarán los resultados teóricos y los experimentales, comprobando si los experimentales confirman los teóricos, para cada uno de los resultados teóricos obtenidos en el apartado 1). El alumno justificará los experimentos realizados para contrastarlos con la teoría, y en caso de discrepancia entre la teoría y los experimentos debe intentar buscar una explicación. En particular hay que confirmar la forma en que crecen los tiempos de ejecución, qué método es mejor para una determinada entrada, cómo el coste de las comparaciones influye en el tiempo de ejecución, si las mejoras teóricas lo son en la práctica, si el tamaño óptimo del caso base es cercano al obtenido teóricamente, etc.

D.2. Cómo resolver la práctica

1) Estudio teórico

La idea de este apartado es que el alumno se familiarice con las notaciones asintóticas y su uso para analizar un problema real. No es tan importante conseguir hallarlo TODO como llegar a utilizar con cierta familiaridad las herramientas de las que disponemos. Para cada uno de los casos lo ideal es que seamos capaces de obtener la Θ pequeña, pero a veces no podremos o será demasiado complicado. Entonces intentaremos calcular Θ , y si esto tampoco es posible intentaremos obtener al menos Ω y O . Si los obtuviésemos y coinciden, entonces habríamos encontrado Θ de forma indirecta. Si obtenemos Θ pero no la Θ pequeña, quizás podamos aún dar una buena aproximación de la constante de la Θ pequeña. Esto es mejor que quedarnos solo con Θ . Por último, si hemos obtenido Θ para los casos peor y mejor entonces nos sirven como Ω y O del caso promedio.

Ten presente que no es necesario repetir cada vez todos los cálculos, que a menudo son muy parecidos de unos casos a otros. Basta con indicar dónde están las diferencias respecto al caso parecido anterior y cómo afecta eso al resultado. Y no pierdas de vista la puntuación que tiene asignada este apartado, tres puntos. Esa es su importancia relativa – sobre un total de diez – en el conjunto de la práctica. La experiencia dice que se dedica mucho tiempo a este apartado y luego poco al resto, sobre todo al estudio experimental. Pero por perfecto que se haga valdrá, como mucho, esos tres puntos, no lo olvides.

En cuanto al estudio de la ocupación de memoria, recuerda que lo importante es cuál es el máximo de memoria necesitado por un algoritmo para un cierto tamaño de entrada, ya que ese máximo es el que limitará el tamaño de la entrada procesable.

1.a) Con caso base 1

Método directo: selección. Hemos elegido este método porque su análisis teórico es sencillo. Si se elige otro puede resultar mucho más complicado, sobre todo el del caso promedio. El esquema es el siguiente. Dada una secuencia de n cadenas $c_1 \dots c_n$:

```

para  $i := 1, \dots, n - 1$  hacer
   $min := i$ 
  para  $j := (i + 1), \dots, n$  hacer
    si  $c[i] > c[j]$  entonces
       $min := j$ 
    finsi
  finpara
  si  $min \neq i$  entonces
     $tmp := c[i]$ 
     $c[i] := c[j]$ 
     $c[j] := tmp$ 
  finsi
finpara

```

El bucle exterior se ejecuta siempre $n - 1$ veces. En su ejecución i -ésima el bucle interior se ejecuta $n - i$ veces, y en cada una de ellas se hace una comparación de cadenas. Esa es la única comparación, de modo que el número de comparaciones es:

$$C(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^n 1 = \sum_{i=1}^{n-1} (n - i) = n(n - 1) - \frac{n}{2}(n - 1) = \frac{1}{2}n(n - 1) \quad (\text{D.1})$$

Como $C(n)$ no depende de la entrada particular, tenemos que todos los casos, incluyendo mejor, peor y promedio, coinciden:

$$C_m(n) = C_M(n) = C_p(n) = C(n)$$

Dado que conocemos $C(n)$ y no depende del caso, el cálculo de las notaciones asintóticas es directo:

$$C(n) \in \Theta(n^2) \quad C(n) \in o\left(\frac{1}{2}n^2\right)$$

Dado que conocemos que el orden exacto de $C(n)$ es ese, también sabemos que $C(n) \in O(n^2)$ y $C(n) \in \Omega(n^2)$.

En cuanto a las asignaciones de cadenas, en cada ejecución i del bucle externo tendremos tres o ninguna, según x_i sea o no el mínimo de los elementos $x_i \dots x_n$. En el peor caso, cuando nunca lo sea, tendremos $A_M(n) = 3n$. Un ejemplo de entrada peor caso es la siguiente:

2	3	4	...	n-1	n	1
---	---	---	-----	-----	---	---

La evolución de la ordenación sería (en negrita los últimos números intercambiados):

1	3	4	...	n-1	n	2
1	2	4	...	n-1	n	3
...						
1	2	4	...	n-2	n	n-1
1	2	4	...	n-2	n-1	n

En el mejor caso tendríamos $A_m(n) = 0$. Este caso corresponde a una entrada totalmente ordenada.

Ya conocemos $A_m(n)$ y $A_M(n)$, y por tanto conocemos sus correspondientes σ pequeña, O , Ω y Θ .

Para calcular el caso promedio suponemos que la probabilidad de que cualquier cadena en $x_{i+1} \dots x_n$ sea el mínimo es $\frac{1}{n-i}$, y por tanto la probabilidad de intercambio en cada ejecución del bucle es $\frac{n-i-1}{n-i}$. De ese modo:

$$A_p(n) = \sum_{i=1}^{n-1} 3 \frac{n-i-1}{n-i} = 3 \sum_{i=1}^{n-1} \left(1 - \frac{1}{n-i}\right) = 3(n-1) - 3 \sum_{i=1}^{n-1} \frac{1}{n-i} \quad (\text{D.2})$$

Tenemos un término positivo y otro negativo. El negativo podríamos calcularlo mediante la siguiente identidad, obtenida por desarrollo del sumatorio:

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

y aproximando mediante integración:

$$\sum_{i=1}^{n-1} \frac{1}{i} \approx \int_1^n \frac{1}{x} dx = \ln n$$

La σ pequeña resultante es $3n$.

Ordenación por mezcla (mergesort). Puedes recordar su esquema en la página 68 del capítulo 9. El número de comparaciones para un tamaño n es dos veces las del tamaño $\frac{n}{2}$ (siempre se divide el problema por la mitad) más las de la mezcla, que son $\frac{n}{2}$ en el mejor caso y $n-1$ en el peor. Es decir, tenemos, según se dedujo en la página 69,

$$C(n) = \begin{cases} 0 & n = 1 \\ 2C\left(\frac{n}{2}\right) + \frac{n}{2} + l & n > 1 \end{cases} \quad (\text{D.3})$$

con $l \in [0, \frac{n}{2} - 1]$. Resolvamos la ecuación de recurrencia para el mejor caso ($l = 0$). Utilizamos el cambio de variable $n = 2^k$ (con lo que $k = \log n$):

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

La ecuación característica es:

$$(x - 2)(x - 2) = 0$$

Con lo que, deshaciendo el cambio de variable, obtenemos:

$$C_m(n) = c_1 n + c_2 n \log n \quad (\text{D.4})$$

Para calcular las constantes c_1 y c_2 planteamos un sistema con dos ecuaciones y las constantes como incógnitas. Las ecuaciones resultan de igualar la ecuación de recurrencia (D.3) con la expresión en (D.4) para el caso base (uno) y el “siguiente caso base”, es decir, el tamaño de problema doble al del caso base (dado que partimos nuestro problema por la mitad). Así obtenemos:

$$\begin{cases} C_m(1) = 0 = c_1 \\ C_m(2) = 2C_m(1) + \frac{2}{2} = c_1 2 + c_2 2 \log 2 \end{cases} \equiv \begin{cases} c_1 = 0 \\ 1 = 2c_1 + 2c_2 \Rightarrow c_2 = \frac{1}{2} \end{cases}$$

Con lo que finalmente

$$C_m(n) = \frac{1}{2}n \log n$$

De forma similar calcularíamos el peor caso, $C_M(n)$, que tendrá el mismo orden exacto, por lo que conocemos el orden exacto del caso general y del promedio.

En cuanto a la σ pequeña del caso promedio, calcularla puede ser algo complicado: hay que hacer un análisis de la probabilidad de cada valor posible de l . Esto impide que podamos conocer con exactitud la σ pequeña. Pero podemos encontrar una buena aproximación de la constante. En primer lugar conocemos sus cotas inferior y superior a partir de la constante de la σ pequeña de $C_m(n)$ y $C_M(n)$, $\frac{1}{2}$ y 1 respectivamente. Pero podemos saber algo más: en promedio, ¿esta constante estará más cerca de la cota superior o de la inferior? Pensemos en qué significan el mejor y el peor caso. El mejor caso significa que todos los números de una de las mitades son menores que todos los números de la otra mitad. Si la distribución inicial de los números es aleatoria esto es algo muy poco probable. En cambio el peor caso, o casos muy próximos al peor, son bastante probables: lo normal es que haya un reparto equitativo de números entre ambas mitades, de modo que los mayores números de una, digamos los dos o tres mayores, sean mayores que todos los de la otra, pero no que todos lo sean. De este modo, en promedio, la constante estará mucho más cerca de 1 que de $\frac{1}{2}$, y podemos afirmar que la σ pequeña del caso promedio estará próxima a $n \log n$.

En cuanto al cálculo de las asignaciones, son siempre las mismas, dado que el procedimiento de mezcla hace siempre $2n$ asignaciones (n para hacer la mezcla y n para la copia de vuelta a la localización original de la cadenas). Sus órdenes se calculan de la misma forma que los de las comparaciones.

Ordenación rápida (quicksort). Puedes recordar su esquema en la página 70. En este caso el problema no siempre se divide por la mitad: el punto de división depende del pivote elegido. En la página 71 vimos que el tiempo de ejecución $t(n)$ tenía un mejor caso de orden $n \log n$ cuando siempre se dividía por la mitad y un peor caso cuadrático cuando la partición dejaba por un lado a un único elemento y por otro los $n - 1$ restantes. El tiempo de ejecución es la suma de asignaciones y comparaciones. A la hora de analizarlas por separado no tiene por qué ocurrir que estos casos mejor y peor para $t(n)$ coincidan con los de $A(n)$ o $C(n)$, como veremos enseguida que en efecto ocurre. De hecho no tiene siquiera por qué ocurrir, en principio, que los casos mejor y peor se encuentren en esas

particiones extremas, pero hacer un análisis exhaustivo sobre este punto sería demasiado complicado, de modo que asumiremos que sí ocurre.

Estudiemos primero las asignaciones. Nos planteamos qué puede pasar en los casos extremos en cuanto a por dónde ocurre la partición. Si suponemos que siempre se divide por la mitad (lo denotaremos con superíndice a) tendríamos una ecuación de recurrencia:

$$A^a(n) = \begin{cases} 0 & n = 1 \\ 2A^a\left(\frac{n}{2}\right) + 3i & n > 1 \end{cases}$$

donde i es el número de intercambios realizados, que en el mejor caso es 0 y en el peor $n/2$. Para el peor caso obtenemos $A_M^a(n) = \frac{3}{2}n \log n$. Para el mejor caso obtenemos $A_m^a(n) = 0$. Si en cambio consideramos que en cada división se deja en una parte un elemento y en la otra el resto tenemos (superíndice b):

$$A^b(n) = \begin{cases} 0 & n = 1 \\ A^b(1) + A^b(n-1) + 3i & n > 1 \end{cases}$$

donde de nuevo i es el número de intercambios realizados, siendo el mejor caso $i = 0$ y el peor $i = 1$ en todos los niveles. Para el mejor obtenemos $A_m^b(n) = 0$, y para el peor, resolviendo la recurrencia – esta vez sin cambio de variable – obtenemos $A_M^b(n) = 3(n-1)$. Resumiendo: no conocemos el valor exacto de $A_M(n)$, así que tomaremos la mejor aproximación que hemos podido encontrar, $A_M^a(n) = \frac{3}{2}n \log n$; por otra parte, el mejor caso es $A_m(n) = 0$, valor independiente del lugar por el que se particione (en particular $A_m^a(n) = A_m^b(n) = 0$).

Veamos ahora qué ocurre con las comparaciones. El número de comparaciones depende directamente del punto de división. Cuando dividimos por la mitad tenemos:

$$C^a(n) = \begin{cases} 0 & n = 1 \\ 2C^a\left(\frac{n}{2}\right) + n - 1 & n > 1 \end{cases}$$

Resolviendo obtenemos $C^a(n) = n \log n$. Si dividimos en un elemento y resto:

$$C^b(n) = \begin{cases} 0 & n = 1 \\ C^b(1) + C^b(n-1) + n - 1 & n > 1 \end{cases}$$

Resolviendo obtenemos $C^b(n) = n^2 - n$. Concluyendo, $C_M(n) = C^b(n) = n^2 - n$ y $C_m(n) = C^a(n) = n \log n$.

Ni para asignaciones ni para comparaciones coinciden los órdenes de mejor y peor caso, de modo que no podemos encontrar el orden exacto, tan solo O y Ω :

$$\begin{aligned} C(n) &\in \Omega(n \log n) & C(n) &\in O(n^2) \\ A(n) &\in \Omega(1) & A(n) &\in O(n \log n) \end{aligned}$$

Respecto al caso promedio, el cálculo para las comparaciones se haría de forma parecida al cálculo del tiempo de ejecución efectuado en la página 72 (donde se han usado todas las constantes igual a uno). El cálculo de las asignaciones es muy complejo, ya que para cada punto de partición hay distintos números de asignaciones posibles, de modo que no intentaremos efectuarlo de forma exacta, sino que nos conformaremos con una

aproximación. Dos posibles aproximaciones sencillas serían aplicar el enfoque utilizado para las comparaciones en cuanto a promediar el punto de partición y luego suponer el mejor o peor caso, para cada partición, en cuanto a asignaciones. Pero el mejor caso es 0 asignaciones en cada partición, que nos llevaría a 0 asignaciones promedio, algo totalmente alejado del promedio real. Por tanto utilizaremos la segunda posibilidad, el caso peor en cada partición, para calcular la aproximación del promedio de asignaciones.

1.b) Con caso base variable

Las expresiones a obtener aquí son parecidas a las conseguidas para caso base uno pero teniendo en cuenta que el caso base pasa a ser una variable, podemos llamarla n_0 . La obtención de un caso base óptimo sólo tiene sentido para el caso promedio y para el tiempo de ejecución (debe ser óptimo en promedio), medido, según se dice en el enunciado, como combinación de asignaciones y comparaciones (debe ser óptimo para el algoritmo en su conjunto, no solo para comparaciones o solo para asignaciones). Puede que no conozcamos con exactitud A_p o C_p . En ese caso utilizaremos lo más próximo a ellos que conozcamos.

La expresión del tiempo promedio será:

$$t_p(n, n_0) = k_1 A_p(n, n_0) + k_2 C_p(n, n_0) + O_p(n, n_0) \quad (\text{D.5})$$

donde las constantes k_1 y k_2 expresan el coste de asignaciones y comparaciones y el término O_p se refiere a otras operaciones, las que no sean asignaciones o comparaciones del tipo ordenado. En nuestro estudio ignoraremos estas otras operaciones. En cuanto a las constantes, habrá que hacer una estimación. Lo más importante es la relación entre k_1 y k_2 : podemos expresar k_2 como $k_2 = xk_1$, de modo que (D.5) pasa a:

$$t_p(n, n_0) = k_1(A_p(n, n_0) + xC_p(n, n_0))$$

k_1 simplemente escala los tiempos, pero el valor de n_0 en que se encuentra el mínimo depende sólo de x , no del k_1 o k_2 concretos. La elección de x depende de muchos factores. Por ejemplo, si las asignaciones se ejecutan en tiempo constante, como es el caso si utilizamos una estructura como la comentada en el volumen 1, en la sección sobre gestión de memoria dinámica del tutorial de C, pero en cambio el tiempo empleado en una comparación depende de la longitud de las cadenas, l , entonces x será kl , donde k es una constante aún por determinar. Si ignoramos esta dependencia de l , según se dice en el enunciado para el estudio teórico, simplemente habrá que estimar cuánto cuesta una comparación en relación a una asignación, ambas de tiempo constante. Seguiremos nuestro estudio considerando $x = 1$.

El caso base óptimo que obtengamos será una aproximación del óptimo real, pero siempre será mejor que usar un caso base cualquiera, sin ninguna información previa. La idea es “tomar decisiones que parezcan lógicas según la información de que disponemos”. Por ejemplo, para calcular el tiempo promedio en el quicksort, para las asignaciones podemos tomar la decisión de utilizar la aproximación del promedio descrita en el apartado anterior, o bien considerar el caso más favorable en cuanto a la partición (dividir por la mitad), y el más desfavorable al hacer las asignaciones en cada partición (se producen todos los intercambios). Podemos probar ambas opciones y ver si el caso base óptimo varía mucho o no.

Para calcular el caso base óptimo obtendremos una expresión para t_p que sea función tanto del tamaño de la entrada como del caso base, es decir, $t_p(n, n_0)$. Una vez conseguida se puede localizar el caso base óptimo según se indica en la página 70.

Veamos un ejemplo de obtención de caso base óptimo. Considera el cálculo de $t_p(n, n_0)$ para la ordenación por mezcla. Fuimos capaces de calcular $A_p(n)$, pero no conocemos $C_p(n)$, sino sólo una aproximación de su σ pequeña. Por los mismos argumentos utilizados para razonar sobre la constante de la σ pequeña de $C_p(n)$ podemos utilizar el peor caso como una aproximación del caso promedio: $C_p(n) \approx C_M(n)$.

Quitaremos el subíndice p para simplificar la notación. Según estas consideraciones podemos definir t como:

$$t(n, n_0) = \begin{cases} t_d(n) & n \leq n_0 \\ 2t\left(\frac{n}{2}\right) + 3n - 1 & n > n_0 \end{cases}$$

Donde $t_d(n)$ es el tiempo del método directo que hayamos elegido y $3n - 1$ viene de $2n$ asignaciones más $n - 1$ comparaciones en el peor caso, con $x = 1$ y $k_1 = 1$. La ecuación de recurrencia se obtiene de la misma forma que si tuviésemos caso base 1, obteniendo:

$$t(n, n_0) = c_1 n + c_2 n \log n + c_3$$

Los principales cambios aparecen al calcular las constantes c_1 , c_2 y c_3 . Ahora el caso base no es 1, sino n_0 . Y los "siguientes casos base" son $2n_0$ y $4n_0$:

$$\begin{cases} t(n_0, n_0) = t_d(n_0) = c_1 n_0 + c_2 n_0 \log n_0 + c_3 \\ t(2n_0, n_0) = 2t(n_0, n_0) + 6n_0 - 1 = c_1 2n_0 + c_2 2n_0 \log 2n_0 + c_3 \\ t(4n_0, n_0) = 2t(2n_0, n_0) + 12n_0 - 1 = c_1 4n_0 + c_2 4n_0 \log 4n_0 + c_3 \end{cases}$$

Resolvemos el sistema y obtenemos:

$$c_1 = \frac{t_d(n_0) - 1}{n_0} - 3 \log n_0 \quad c_2 = 3 \quad c_3 = 1 \quad (\text{D.6})$$

Con lo que

$$t(n, n_0) = \left(\frac{t_d(n_0) - 1}{n_0} - 3 \log n_0 \right) n + 3n \log n + 1$$

El valor de $t_d(n)$ se obtiene a partir de las comparaciones y asignaciones promedio para el método directo ((D.1) y (D.2)). Teniendo en cuenta que hemos elegido $x = 1$ y $k_1 = 1$ obtenemos

$$t_d(n) = \frac{1}{2}n(n-1) + 3(n-1) - 3 \ln n = \frac{1}{2}n^2 + \frac{5}{2}n - 3 \ln n - 3$$

Calculamos la derivada parcial y la igualamos a 0:

$$\frac{\partial t(n, n_0)}{\partial n_0} = \left(\frac{\frac{1}{2}n_0^2 + 3 \ln n_0 + 1}{n_0^2} - \frac{3}{n_0} \log_2 e \right) n = 0$$

Los valores de n_0 para los que se satisface la ecuación son: $n_0 \approx -0,395$ y $n_0 \approx 6,645$. Cuando tenemos varios candidatos a caso base óptimo discriminamos entre ellos calculando la segunda derivada respecto a n_0 y comprobando cuáles de ellos son mínimos relativos (segunda derivada positiva) y, de ellos, cuál es el que da un valor de tiempo menor –mínimo absoluto. En nuestro caso lo único que conseguimos con esto es comprobar que 6,645 es un mínimo.

2) Optimizaciones

2.a) Ordenación por mezcla

La forma de implementar esta optimización pasa por tener una tabla idéntica a la tabla en que se almacenan los elementos a ordenar. Llamemos A a la tabla original y B a la auxiliar. En vez de en cada nivel de la recursión hacer una mezcla de dos porciones de A sobre B y luego copiar el resultado de vuelta sobre A lo que haremos es dejar la mezcla en B y la “próxima vez” mezclar de B en A. La cuestión fundamental es: ¿qué significa exactamente eso de *la próxima vez*? Tenemos que estudiar cómo se decide si hay que mezclar de un array al otro o viceversa. La clave está en considerar los niveles de recursión. Ante la tabla completa de n elementos (supongamos n potencia de 2) dividimos en dos partes de $\frac{n}{2}$ elementos. Si queremos que el resultado final quede ordenado en A mezclaremos desde B hacia A. Este nivel es impar (nivel 1). En el siguiente nivel (nivel 2) tenemos para cada una de las dos mitades iniciales dos submitades de $\frac{n}{4}$ elementos. Si en el nivel 1 necesitábamos los elementos a mezclar en B en este nivel mezclaremos sobre B. Y se trata de un nivel par. Si seguimos la recursión veremos enseguida que la clave es mezclar de la misma forma que para los niveles 1 y 2 para todos los niveles impares y pares respectivamente. Teniendo presente además que cuando lleguemos en un nivel al caso base aún no habremos hecho ningún movimiento, y si lo que necesitamos es que ya haya elementos en B habrá que copiarlos una primera vez desde A. Evitamos tener en cuenta esto si directamente hacemos una copia completa de A sobre B.

Respecto a la mejora conseguida en el tiempo de ejecución, fíjate en que la única diferencia respecto a la ordenación por mezcla normal es que en cada nivel de recursión se hacen n asignaciones en vez de las $2n$ consideradas en la página 260, de modo que se divide por 2 el número de asignaciones pero no se cambia su orden, y se mantiene el número de comparaciones.

2.b) Ordenación rápida

La clave de esta optimización reside en la elección del pivote. Si lo elegimos de forma aleatoria la posibilidad de un caso peor se desvanece automáticamente: dado que la secuencia de pivotes elegidos es arbitraria no existe forma de plantear una entrada que sea un peor caso. Con elegir un único pivote de forma aleatoria ya evitamos el peor caso cuadrático, pero podemos hacer algo incluso un poco mejor: intentar elegirlo de forma que nos acercarnos a los resultados del mejor caso. Según se estudió en 71 el mejor caso era aquella entrada que hacía que el pivote elegido partiese por la mitad en cada ocasión la tabla de elementos a ordenar. Podríamos conseguir esto para cualquier entrada si siempre eligiésemos la *mediana*, que se define precisamente de ese modo, como aquel elemento que

queda en la mitad si ordenamos la tabla. Pero claro, para obtenerla por el método obvio tendríamos que ordenar la tabla, y eso es precisamente nuestro objetivo. Por tanto no podemos aspirar a obtener el resultado del mejor caso. Pero sí a aproximarnos. Podemos hacer una buena estimación de la mediana si tomamos aleatoriamente un cierto número de elementos m , por ejemplo $m = 5$, los ordenamos y nos quedamos con el de en medio. Esta operación tiene un coste constante, de forma que no se incrementa el orden. O podemos incluso mejorar esta idea haciendo depender m del tamaño de la entrada, n , pero de forma que no se incremente el orden. Podemos obtener experimentalmente el valor de m para varios valores de n y así obtener una regla general para cualquier n . Puede que encontremos que efectivamente el m óptimo depende de n o, al contrario, que es constante.

3) Implementaciones

El objetivo de este apartado de la documentación es que se expliquen las características más sobresalientes de los programas que implementan los algoritmos de ordenación bajo estudio. Se trata de aportar alguna explicación que permita hacerse una idea rápida de qué es lo que se ha hecho, qué estructuras de datos se han manejado, qué limitaciones tiene el programa, si hay algún aspecto que consideras especialmente interesante en cuanto a su originalidad o funcionalidad, etc. En particular, cómo se leen los datos de fichero y se organizan en memoria, cómo se han implementado las operaciones de comparación y asignación (o cuáles se han reutilizado), cómo se hace la reserva de memoria. Habría que explicar aquí, especialmente, los detalles sobre la implementación de los métodos optimizados, dado que sus implementaciones están más abiertas a las decisiones del alumno.

Lo que no hay que hacer aquí es mostrar el listado del código (como dice expresamente el enunciado) o un listado de cabeceras de funciones y parámetros. Obviamente la puntuación de este apartado tiene en cuenta las implementaciones en sí, qué se ha programado, no solo lo que se escriba en la documentación. En ese sentido, que los distintos programas realizados funcionen es un requisito imprescindible para considerar aprobada la práctica en su conjunto.

Un ejemplo de lo que habría que decir en este apartado podría ser el siguiente:

Para todos los métodos implementados hago una lectura secuencial inicial del fichero a ordenar, en la que determino el número de palabras y caracteres del fichero. Con esta información hago la reserva de memoria para una estructura como la descrita como ejemplo en la sección de gestión dinámica de la memoria de los apuntes. A continuación hago una segunda lectura secuencial del fichero en la que voy llenando la estructura creada. Utilizo caracteres '\0' como fin de cadena en el array letras para poder utilizar las funciones de cadena de caracteres de C.

Esta estructuración de la información permite que la asignación de cadenas se reduzca a la asignación de un puntero, por tanto la operación de asignación tendrá un coste constante. En cuanto a las comparaciones, he implementado una función compcad que recibe dos cadenas como parámetros de entrada y devuelve: 0 si son iguales, 1 si la primera es mayor y 2 si lo es la segunda. Para determinar el resultado, esta función recorre ambas cadenas en paralelo, desde el primer carácter hasta que encuentra uno que no coincida en ambas o llegue al final de una de ellas.

(...)

(...) en el algoritmo de mergesort mejorado se realiza una copia inicial del array A sobre B, lo cual evita comprobar, cada vez que se llega al caso base, si estamos en un nivel par o impar, y en el primer caso hacer una copia previa de elementos de A sobre B. Se utiliza una variable booleana (entero de C) `nivel-impar`, que se inicializa a 1, para llevar cuenta de si el nivel de recursión en que nos encontramos es par o impar. Se trata de un parámetro de la función mergesort, pasado por copia. Su actualización ocurre en cada nueva llamada recursiva de mergesort a sí mismo: mergesort (`!nivel-impar, resto-parametros`). La información sobre el nivel es utilizada por la función mezcla, que tomará como arrays origen y destino A y B, respectivamente, o viceversa, según el nivel (...)

(...)

4) Validación

Este apartado se centra en comprobar que las implementaciones de los algoritmos funcionan correctamente.

4.a) Programa de validación

En este apartado hay que hacer un programa que permita dicha comprobación. El programa que se sugiere en el enunciado permitiría ordenar un fichero mediante dos métodos y comparar entre sí las salidas. Si todos los métodos obtienen el mismo resultado, presumiblemente todos funcionan. Para que el programa sea realmente automático debería permitir la comparación automática de ambas salidas. Es decir, debería ser capaz de dar como resultado una respuesta: los ficheros de salida son iguales o no lo son. También es interesante incluir, bien en el propio programa bien como un programa independiente, la posibilidad de generar ficheros de forma automática, por ejemplo ficheros aleatorios (ver función `rand` de C), para probar los algoritmos. Ante el problema de ordenar datos otra posible comprobación de corrección es ver si cada elemento de la tabla de salida es menor o igual que el anterior, suponiendo que no se ha perdido ni añadido ningún elemento en el proceso.

4.b) Pruebas de validación

En este apartado se debe explicar qué conjunto de pruebas se ha utilizado para hacer la validación. Se trata de explicar qué ficheros concretos se han elegido y por qué. No entraña ninguna dificultad, la idea es asegurar la correcta validación del programa mediante ficheros suficientemente significativos. ¿Y qué son ficheros suficientemente significativos? Pues muchos ficheros de tamaños razonablemente grandes, tanto en el número de cadenas como en la longitud de las cadenas. Hablamos de miles de líneas. Obviamente estos ficheros han de ser generados de forma automática, no manual. Para eso está el programa de la subsección anterior. La idea es asegurarnos de que nuestro programa no tiene ningún fallo, ni siguiera de esos que aparecen sólo de vez en cuando, especialmente inoportunos en una entrevista de prácticas. El objetivo fundamental de este apartado es cazar cualquier fallo. Es responsabilidad del alumno el conseguir un programa a prueba

de fallos y en este apartado se puntúa el llevar a cabo y describir una batería de pruebas convincente.

Un ejemplo podría ser: *he ejecutado el programa de validación 100 veces, con generación aleatoria del fichero de entrada para tamaños de cadena entre 500 y 1.000 y para un número de cadenas entre 10.000 y 100.000, y no se ha colgado ni generado resultados erróneos en ninguna de las ocasiones, aunque sí ocurría que para ejecuciones con más de 80.000 cadenas, aproximadamente, a veces el programa era abortado por falta de memoria.*

5) Estudio experimental y comparación con teórico

5.a) Estudio experimental

Para hacer el estudio experimental es necesario contar con un programa que se encargue de realizar **baterías de tests** con los parámetros que le digamos y devuelva los resultados en un formato con el que podamos trabajar fácilmente para interpretarlos, ayudándonos incluso de una representación gráfica, generada a partir de la salida del programa, cuando lo consideremos oportuno. Construir un programa de este tipo debería ser el **primer paso** en este apartado. El programa nos debe permitir hacer cualquier tipo de prueba que queramos. Por ejemplo, probar un cierto algoritmo para un mismo caso base y un rango de tamaños de entrada, o generar una tabla de tiempos para un rango de tamaños de entrada y casos base, para un cierto caso base y un rango de tamaños de entrada y longitud de cadenas, etc. Este programa también nos debería permitir la obtención de resultados para las mismas entradas y distintos algoritmos. Incluso para las mismas entradas y distintos casos base ya que, si queremos comparar los algoritmos entre sí, deberíamos probar cada uno de ellos con su caso base óptimo.

En definitiva, el tiempo de ejecución para un cierto algoritmo va a depender de tres parámetros: el número de cadenas a ordenar, n , el caso base utilizado (para los métodos divide y vencerás), n_0 , y la longitud de las cadenas, l . Es decir, tenemos $t(n, n_0, l)$. Adicionalmente, en el quicksort mejorado habrá un cuarto parámetro, m , el tamaño de la muestra aleatoria tomada para determinar el pivote. Nuestro programa nos debe permitir la generación automática de experimentos para determinar valores de esta función para valores fijos o rangos en cada uno de los tres parámetros. Es interesante también que se permita que los rangos se generen con incrementos lineales (10, 20, 30, ..., 100) o exponenciales (2, 4, 8, 16, ..., 1024). Un programa de generación automática de tests requiere una generación automática de entradas para nuestros algoritmos. Generaremos entradas aleatorias cuando queramos estudiar tiempos promedio. Cuando queramos estudiar casos mejores o peores construiremos entradas que constituyan para cada algoritmo esos casos extremos, por ejemplo entradas ordenadas (o inversamente ordenadas) para la ordenación rápida para el peor caso. Para estudiar el efecto de la longitud de las cadenas, es interesante utilizar cadenas que fuercen comparaciones que analicen la cadena más corta hasta el final, es decir, cadenas que sean iguales en todos los caracteres menos el último.

Una vez que tenemos el programa de generación de tests nos planteamos qué cosas queremos estudiar, en consecuencia qué tests habremos de realizar, y analizaremos los resultados obtenidos. Según estos resultados, podremos concluir que ya tenemos suficiente información para describir el comportamiento de aquello que estábamos estudiando, o

bien decidiremos que necesitamos más información, es decir, que necesitamos repetir el test para otros parámetros, por ejemplo, con otro rango, o con otro grado de detalle, para un parámetro concreto. Veamos cómo estudiariamos algunas de las cosas que se nos piden en el enunciado.

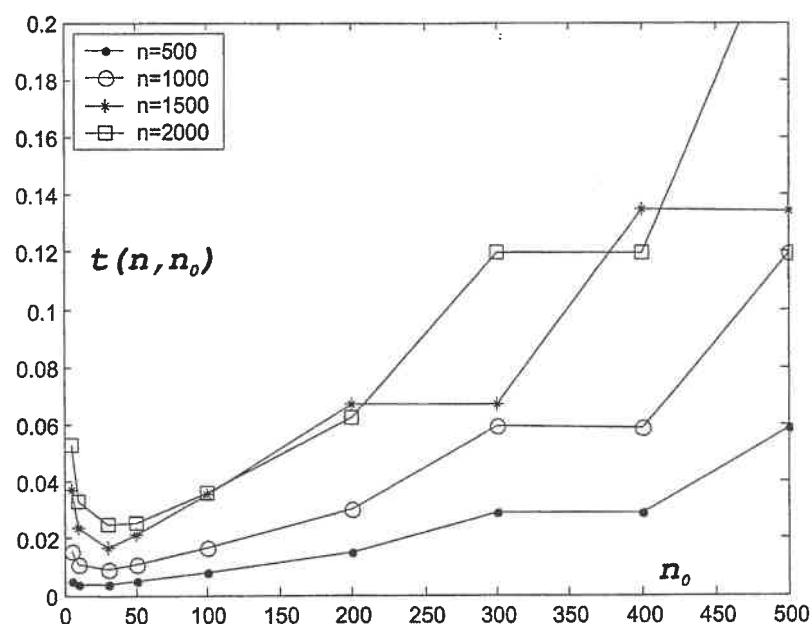
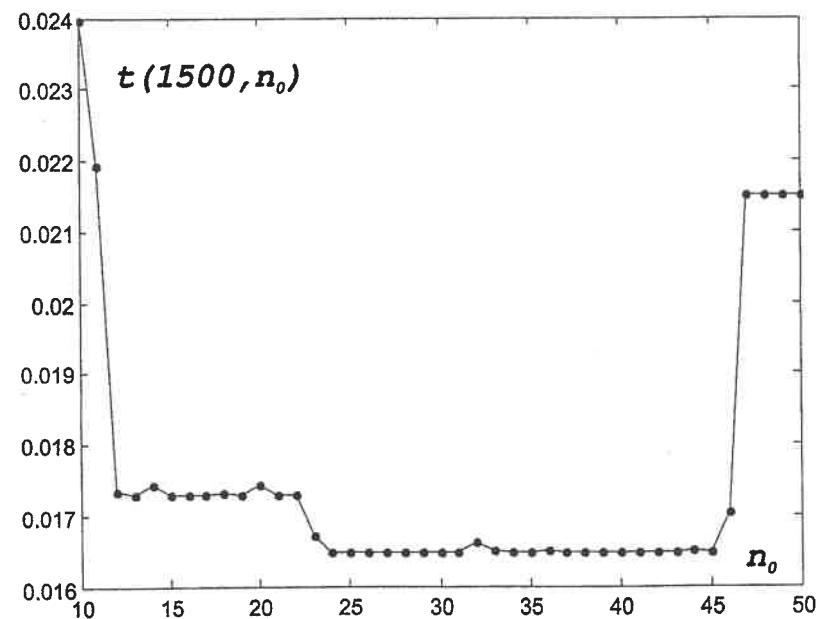
Vamos a estudiar por ejemplo el tiempo de ejecución promedio, t_p , de la ordenación por mezcla. Tenemos tres parámetros cuya influencia queremos determinar – n , n_0 y l . Formalmente tenemos una función de tres parámetros, $t(n, n_0; l)$, que ni siquiera es representable gráficamente. Necesitamos abordar este estudio por partes:

- Por ejemplo, podemos asumir por el momento una longitud de cadena fija, digamos $l = 100$, ya que sabemos que esto sólo va a afectar a los tiempos de ejecución multiplicando algunos términos por una constante. Es decir, estudiaremos $t(n, n_0, 100)$, que ya sí es representable gráficamente (mediante una superficie tridimensional). Pero aún así no trazaremos simplemente una gráfica de t frente a esos dos parámetros: de nuevo abordaremos el estudio por partes.
- Podemos por ejemplo estudiar primero cómo evoluciona t para algunos tamaños de entrada fijos según el caso base elegido, y buscar el óptimo, que podrá ser el mismo para todo tamaño de entrada o depender de ésta. Por ejemplo elegimos $n \in \{500, 1000, 1500, 2000, 4000\}$. Para cada uno de ellos obtenemos el tiempo de ejecución para unos primeros valores orientativos de n_0 . El resultado es la siguiente tabla (con tiempos expresados en microsegundos):

$n_0 \setminus n$	500	1000	1500	2000	4000
5	5079.8	15602.2	37166.4	52834.2	187989.6
10	3817.8	10619.0	23947.4	32884.8	108155.4
30	3704.8	9146.0	16465.0	24928.2	72590.8
50	4789.0	10723.0	21482.2	25650.8	68263.6
100	7821.6	16545.8	35584.8	35837.8	79502.2
200	14643.0	30020.2	67163.8	62088.6	130074.4
300	28995.4	59119.8	67090.4	119691.6	244483.4
400	28953.6	59060.8	134865.2	119664.6	244668.2
500	58819.6	119497.4	134049.4	240530.8	487404.6

A la vista de esta tabla podríamos decir que el caso base óptimo está en torno a 30 para $n \in \{500, 1000, 1500, 2000\}$ y en torno a 50 para $n = 4000$ (es decir, el caso base óptimo varía con la entrada). En la gráfica D.1 hemos representado (en segundos) las cuatro primeras columnas de la tabla. La quinta necesita de una escala mayor, por lo que ha sido omitida.

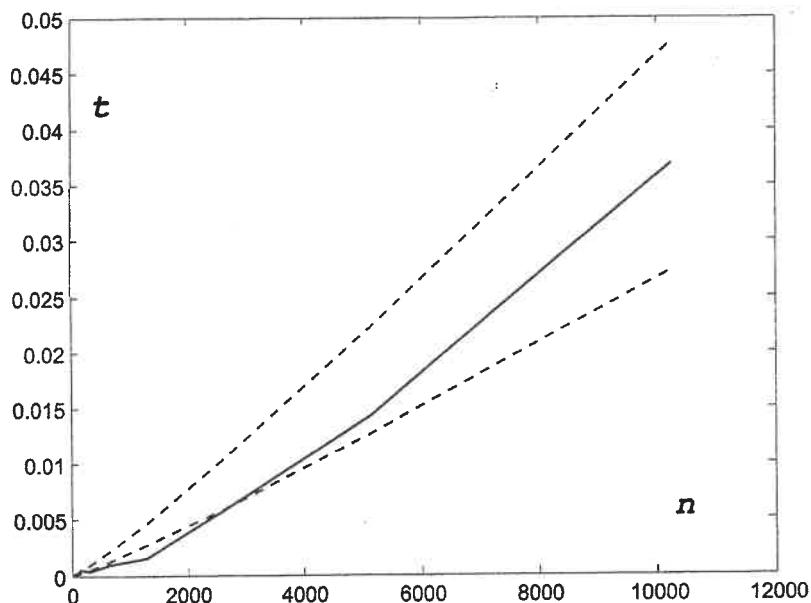
Para determinar el óptimo concreto para un cierto n habrá que hacer un estudio en un rango de n_0 más específico. Por ejemplo, si nos centramos en $n = 1500$, hacemos un estudio en el rango $n_0 \in [10, 50]$ con muestras espaciadas una unidad. El resultado que obtenemos es el que se muestra en la figura D.2. Podemos ver que para n_0 en el rango $[24, 45]$, aproximadamente, tenemos el mínimo tiempo de ejecución, luego cualquier valor en ese rango podría ser caso base óptimo. El mínimo tiempo se alcanza realmente para 24, pero la diferencia

Figura D.1: t frente a n_0 , primera aproximación.Figura D.2: t frente a n_0 , $n = 1500$, rango en torno a $n_0 = 30$.

con el resto de valores en ese rango es mínima. En este sentido, es conveniente darse cuenta de que el caso base óptimo experimental se sitúa en un valle en que cualquier valor sería una buena aproximación al óptimo. Y también ocurre que la subida desde ese valle al desplazarnos a la izquierda, acercándonos al caso base uno, tiene mayor pendiente que cuando nos desplazamos hacia la derecha (observa la columna para $n_0 = 1500$ de la tabla anterior). Ambas cosas suceden en general. Esto nos lleva a concluir que es conveniente tomar un caso base real mayor al óptimo teórico obtenido, especialmente cuando este es pequeño, como en nuestro caso, ≈ 7 , ya que lo más probable es que estemos consiguiendo acercarnos más a los resultados del óptimo experimental, y en el peor de los casos no nos alejaremos nada de ellos – si ya estábamos en el valle – o nos alejaremos poco – si hemos entrado en la zona de subida por la derecha, con poca pendiente–. En cambio si estábamos en la zona a la izquierda del valle, con pendiente elevada, estaremos lejos de los resultados del óptimo experimental. Por ejemplo, vemos que nuestro óptimo teórico se encontraba en la zona de pendiente elevada, con un tiempo mucho mayor al del óptimo experimental. A la vista del valor del óptimo teórico podríamos haber elegido un óptimo real de 16, aproximadamente el doble del teórico. El resultado en este caso habría sido que aún no conseguiríamos resultados tan buenos como los del óptimo experimental, pero hubiesen sido bastante mejores que los del calculado teóricamente.

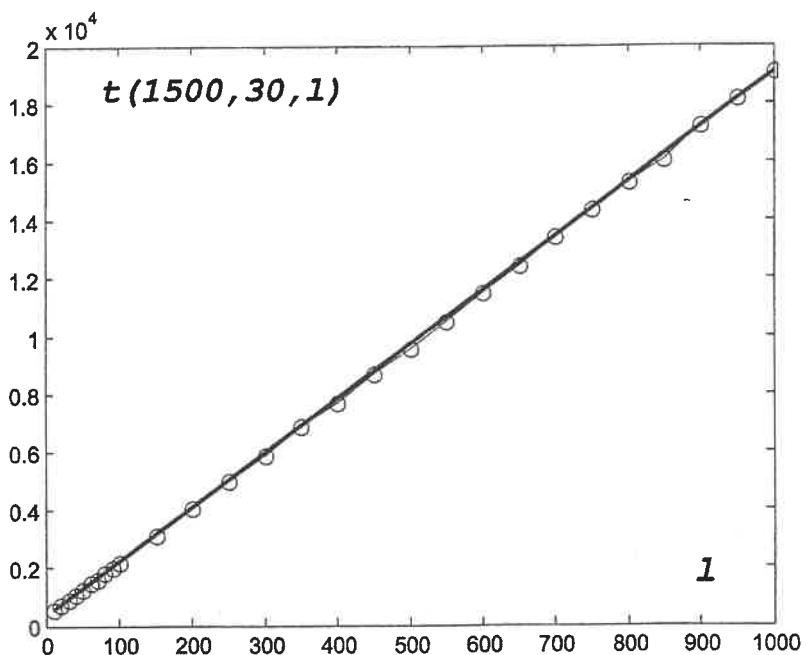
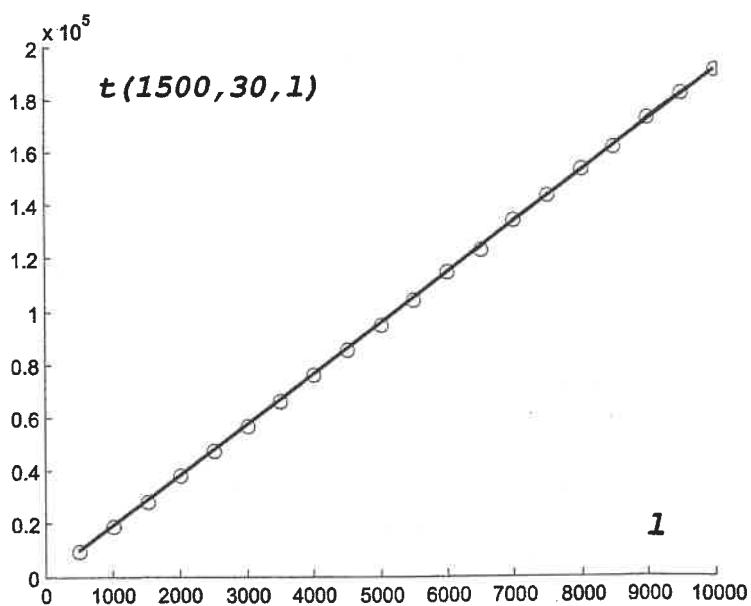
A veces podemos obtener curvas tipo “diente de sierra”, con muchas irregularidades. Uno de los motivos por los que puede ocurrir esto es porque una misma entrada con parámetros idénticos puede tardar tiempos diferentes en ejecuciones diferentes. Estamos hablando de tiempos tan pequeños que ya influyen cuestiones como un acceso eventual a disco, el reparto de tiempo de CPU entre procesos, el acceso a la memoria caché, la posible paginación, etc. Para evitar la interferencia de estos efectos conviene hacer varias mediciones para los mismos parámetros de entrada y promediar. Así lo hemos hecho aquí. Otra cuestión a tener en cuenta es que el tiempo consumido por una ejecución no tiene por qué evolucionar suavemente con las variaciones en el parámetro estudiado. Una pequeña variación en el parámetro puede producir un gran efecto en el tiempo. Por ejemplo puede ocurrir que aumentar en una unidad el caso base lleve a una división menos por divide y vencerás, resolviendo un problema de tamaño doble por el método directo, cuadrático, en vez de dos problemas de tamaño la mitad más la mezcla lineal. Fíjate por ejemplo en el salto que hay de $n_0 = 45$ a $n_0 = 47$ en la figura D.2.

- Ahora podemos estudiar la variación del tiempo de ejecución según el tamaño de la entrada. Lo haremos para uno o varios casos base fijados. Una buena elección de caso base es el óptimo si es común a todos los tamaños de entrada. Si no, algunos representativos. Para $n_0 = 36$ obtenemos la gráfica de la figura D.3 (eje horizontal logarítmico, base 10, muestras de n : 10, 20, 40, 80, 160, 320, 640, 1280, 2560, 5120 y 10240). La línea sólida corresponde a los tiempos medidos. Las líneas discontinuas corresponden a la función $n \log n$ multiplicada

Figura D.3: t frente a n para $n_0 = 36$.

por dos constantes ($2e-7$ y $35e-8$). Actúan como cotas de la línea de los tiempos medidos a partir de cierto n . Esta es una prueba gráfica de que el orden del algoritmo es $n \log n$. Fíjate en que la parte inicial de la gráfica experimental no está entre las cotas. Esto es debido en parte a que para tamaños menores ciertos factores influyen positivamente, por ejemplo, la gestión de la memoria será más eficiente porque una proporción mayor de los datos necesarios estarán en la memoria caché. En cualquier caso, recuerda que en la definición de orden se habla de que la función quede acotada *a partir de un n suficientemente grande*.

- Respecto al efecto de la longitud de las cadenas, podemos fijar por ejemplo $n = 500$ y $n_0 = 30$ y estudiar cómo varía el tiempo según esta longitud. Hacemos un primer estudio con $l \in [10, 100]$, con incrementos de 10 en 10, y $l \in [100, 1000]$, con incrementos de 50 en 50. Obtenemos el resultado que se muestra en la figura D.4. Los círculos representan los valores medidos. Hemos dibujado un segmento en trazo grueso entre el primer punto y el último. Vemos que, salvo algunas pequeñas variaciones, todos los puntos coinciden aproximadamente con el segmento, lo que demuestra que el tiempo depende linealmente de l . Se han utilizado cadenas iguales en todos sus elementos salvo en el último para forzar una mayor influencia del coste de las comparaciones. Esto constituye en realidad el peor caso en cuanto a la longitud de las cadenas. Para asegurarnos de esta dependencia haremos un nuevo experimento en otro rango de valores: $l \in [500, 10000]$, con incrementos de 500 en 500. El resultado es el que se muestra en la figura D.5. La dependencia lineal es clara.

Figura D.4: $t(1500, 30, l)$ para $l \in [10, 1000]$.Figura D.5: $t(1500, 30, l)$ para $l \in [500, 10000]$.

- Por último haremos un estudio comparativo de los cinco algoritmos implementados. Para ello ejecutaremos los cinco algoritmos para un tamaño fijo de cadena ($l = 20$) y en distintos rangos de tamaño de entrada, para tratar de establecer qué algoritmo es mejor en qué intervalos de tamaño de entrada. Respecto al caso base, para cada tamaño de entrada concreto haremos una búsqueda automática del óptimo experimental, n_{0opt} , que será utilizado como n_0 para ese n . Para $n \in [10, 100]$ obtenemos los resultados mostrados en la figura D.6. Podemos observar que incluso para $n = 10$

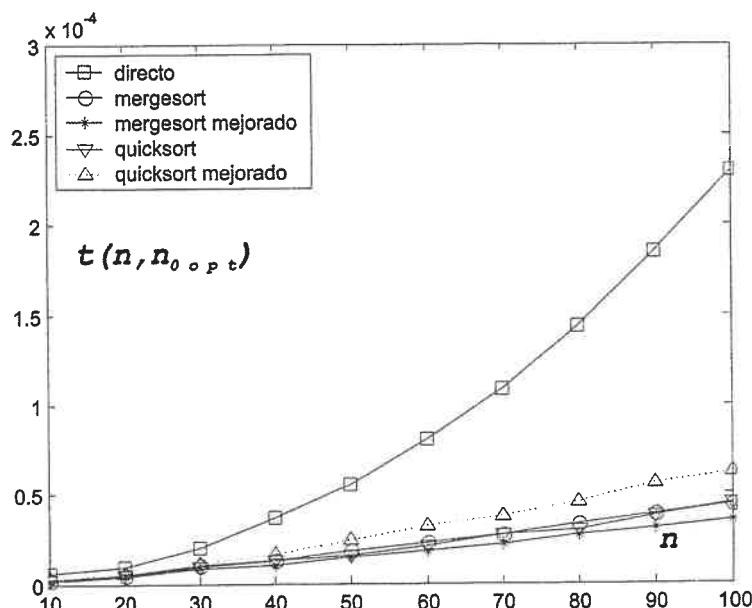


Figura D.6: $t(n, n_{0opt}, 20)$ para $n \in [10, 100]$.

el método directo ya es peor que cualquiera de los otros cuatro. El quicksort mejorado tarda en torno a un 50 % más que los demás. El quicksort no mejorado tarda algo más que los mergesort, aunque en este rango la diferencia no es significativa. De los mergesort, el mejorado tarda algo menos que el no mejorado.

Hacemos otro experimento como el anterior para $n \in [100, 1000]$. Los resultados se muestran en la figura D.7. En este caso no se ha representado el método directo, ya que se sigue alejando del resto y su representación aumenta la escala e impide estudiar los otros con mayor detalle. En esta gráfica se aprecia mejor la ventaja del mergesort sobre el quicksort, ventaja que va aumentando con el tamaño del problema. También se aprecia mejor que el quicksort mejorado tarda bastante más que el resto. Recordemos que su mejora consistía en evitar un peor caso cuadrático, pero requería operaciones adicionales. Por último probaremos con $n \in [1000, 10000]$. No mostramos los resultados obtenidos porque no aportan ninguna información salvo confirmar las tendencias observadas en las ejecuciones anteriores.

Los estudios realizados son solo algunas de las posibilidades, a título orientativo, para uno de los métodos de ordenación. Es parte del trabajo a realizar en la práctica el

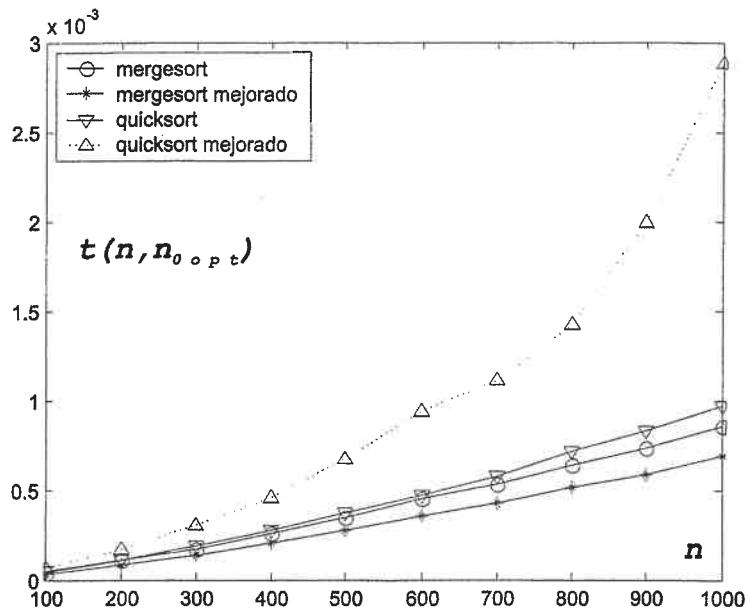


Figura D.7: $t(n, n_{0opt}, 20)$ para $n \in [100, 1000]$.

determinar cuáles serían los estudios más convenientes según los casos.

5.b) Comparación con teórico

El objetivo de este apartado es comparar los resultados experimentales obtenidos con las predicciones del estudio teórico. Se pueden comparar, por ejemplo, los órdenes teórico y experimental de los distintos métodos y ver si coinciden o no. Una de las formas de hacerlo es el método gráfico utilizado en la figura D.3, página 271. Podría ocurrir que no en algún caso. Si es así habrá que explicar por qué.

Se puede estudiar el efecto de la variación de la longitud de las cadenas (como ya hemos hecho en la sección anterior), cuestión no considerada en el estudio teórico, y explicar la causa del comportamiento observado (en nuestro caso, linealidad del tiempo respecto a la longitud de las cadenas).

También podemos ver si coinciden o no los casos base teóricos y experimentales y en caso negativo intentar razonar el porqué de la diferencia. En el ejemplo de caso base experimental estudiado aquí, vemos que no hay coincidencia con la predicción teórica. Las causas pueden ser múltiples: la elección arbitraria de constantes en el estudio teórico, las aproximaciones realizadas, como por ejemplo la de no contar otras operaciones que no sean comparaciones o asignaciones del tipo ordenado, etc. En cualquier caso siempre será mejor utilizar el caso base óptimo teórico, o algún valor próximo a éste, preferentemente mayor, según se argumentó en la página 270.

Podemos comprobar si los experimentos corroboran las mejoras predichas para las optimizaciones del apartado D.2. Por ejemplo, para comprobar la mejora en el quicksort deberíamos generar ficheros correspondientes al peor caso (fichero ordenado o inversa-

mente ordenado) y comprobar que con el algoritmo mejorado ya no obtenemos un orden cuadrático. Este tipo de experimentos nos pueden ayudar a detectar errores de programación ante una discrepancia grave entre las predicciones y los resultados experimentales. Por otra parte, en ocasiones existirán tales discrepancias en ausencia de errores, debido a cuestiones prácticas no tenidas en cuenta en el análisis teórico, como pueda ser el hecho de que por estar haciendo la ordenación sobre un array de punteros a cadena las asignaciones de cadenas no dependan de la longitud de estas. En estos casos será necesario razonar la causa de las discrepancias.

Apéndice E

Práctica 3: Esquemas algorítmicos

El objetivo de esta práctica es llegar a conocer en detalle cada uno de los siguientes esquemas algorítmicos:

- Programación dinámica
- Avance rápido (aproximación del óptimo)
- Backtracking
- Ramificación y poda (utilizando el algoritmo voraz como estimación del óptimo)

Esto implica algo más que simplemente resolver algún problema con cada uno de los métodos: será necesario estudiar las características particulares de cada uno de los esquemas utilizados, determinando para cada uno el tipo de análisis teórico y experimental más adecuado, estudiando aquellos detalles que en cada esquema nos permitan optimizar los programas, los tipos de experimentos a realizar para determinar la bondad de cada elección realizada, etc.

En este apéndice plantearemos un problema de optimización, daremos algunas ideas sobre posibles formas de resolverlo mediante cada uno de los cuatro esquemas, y daremos algunas indicaciones sobre el estudio teórico y experimental más adecuado en cada caso.

Date cuenta de que el problema podría ser también enfocado como un problema de caminos mínimos en grafos, y podríamos resolverlo mediante el algoritmo de Dijkstra. Plantéate cómo.

E.1. Enunciado

Dado un tablero cuadrado de dimensión n por n , con valores naturales en las casillas, y dada una posición inicial en el tablero, se trata de encontrar el camino de longitud mínima para llegar desde el origen al borde del tablero pudiendo acceder de una casilla a otra vecina en vertical, horizontal o diagonal, siendo la longitud de un camino la suma de los valores en las casillas que lo componen.

Realiza un breve estudio teórico sobre la resolución del problema con cada uno de los cuatro esquemas algorítmicos, resuélvelo, y haz un estudio experimental usando el programa que has implementado. Compara los resultados teóricos y experimentales.

E.2. Programación dinámica

E.2.1. Resolución

En este problema el *principio de optimalidad* consiste en que el camino mínimo desde una casilla al borde es la suma del valor de esa casilla más el mínimo de los caminos mínimos desde cada una de las 8 casillas adyacentes a ella al borde. Dicho más formalmente, si llamamos $x_{i,j}$ a la casilla situada en la fila i y columna j y llamamos $C(x)$ a la función que devuelve el coste del camino mínimo desde la casilla x al borde y $P(x)$ al peso de la casilla x tenemos que:

$$C(x_{i,j}) = P(x_{i,j}) + \min \left\{ \begin{array}{l} C(x_{i-1,j-1}), C(x_{i-1,j}), C(x_{i-1,j+1}), C(x_{i,j+1}), \\ C(x_{i+1,j+1}), C(x_{i+1,j}), C(x_{i+1,j-1}), C(x_{i,j-1}) \end{array} \right\}$$

Se trata de una función recursiva cuya ejecución lleva a la repetición de cálculos para muchos valores, situación idónea para un cálculo alternativo mediante programación dinámica. Redefinimos $C(x)$ como una familia de funciones $C_s(x)$ donde para cada s tenemos una función distinta. $C_s(x)$ es el valor del camino de s pasos óptimo desde x al borde. Si no se puede llegar en s pasos desde x al borde ese valor será infinito. Para calcular estas funciones empezaremos por $C_0(x)$: las únicas casillas desde las que se puede llegar en 0 pasos al borde son las casillas del borde, con coste el peso de la propia casilla, y el resto de casilla tendrán valor infinito:

$$C_0(x_{i,j}) = \begin{cases} P(x_{i,j}) & \text{si } i \in \{1, n\} \text{ o } j \in \{1, n\} \\ \infty & \text{en otro caso} \end{cases}$$

A partir de esta función se pueden ir construyendo sucesivamente las funciones para $s = 1, 2, \dots$ mediante aplicación de la siguiente fórmula:

$$C_s(x_{i,j}) = \min \left\{ C_{s-1}(x_{i,j}), P(x_{i,j}) + \min \left\{ \begin{array}{ll} C_{s-1}(x_{i-1,j-1}), & C_{s-1}(x_{i,j-1}), \\ C_{s-1}(x_{i+1,j-1}), & C_{s-1}(x_{i-1,j}), \\ C_{s-1}(x_{i+1,j}), & C_{s-1}(x_{i-1,j+1}), \\ C_{s-1}(x_{i,j+1}), & C_{s-1}(x_{i+1,j+1}) \end{array} \right\} \right\}$$

¿Y hasta qué s calculamos? Pararíamos de calcular a partir del s en que ocurra $\forall x C_s(x) = C_{s-1}(x)$, ya que a partir de él nada cambiaría. Respecto a la tabla de reconstrucción de decisiones que acompaña a cada función $C_s(x)$, en cada casilla simplemente habrá que ir escribiendo, según cuál de los ocho candidatos fue elegido en la segunda función mínimo, respectivamente $\nwarrow, \leftarrow, \nearrow, \downarrow, \searrow, \rightarrow, \nearrow \circ \uparrow$.

Date cuenta de que en realidad por las características del problema a la hora de implementar la construcción de las funciones en forma de tablas será suficiente con mantener cada vez simplemente una tabla para la s que estemos calculando y otra para $s - 1$. Los detalles de este tipo han de ser comentados en la documentación de la práctica.

E.2.2. Estudio teórico y experimental

El camino de máxima longitud para llegar al borde pasaría por todas las casillas que no son del borde – $(n - 2)^2$ – y por último recorrería una del borde, de modo que

su longitud sería $1 + (n - 2)^2$ pasos. El camino óptimo hasta el borde será normalmente bastante más corto que este camino, sobre todo teniendo en cuenta que semejante camino tendrá “atajos”, dado que continuamente encontraremos en él, según se avanza hacia el borde, casillas que tienen como adyacentes a otras que a su vez forman parte del camino por recorrer más adelante, cuando en realidad podemos pasar a ellas en un solo paso. Eso es lo que denominamos un atajo.

Tiendo en cuenta que en cada tablero tenemos n^2 casillas, el número de valores de la función C a calcular como máximo sería, por tanto, $n^2(1 + (n - 2)^2)$. El cálculo de esos valores se realiza en un tiempo constante, por lo que podemos decir, según esta cota superior del número de valores calculados, que el tiempo de ejecución $\in O(n^4)$. ¿Existe realmente una entrada para la que obtengamos este orden? Sí que existe, por ejemplo una entrada del tipo mostrado en la siguiente tabla (con (4, 5) como casilla inicial):

1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	1
1	1	1	1	1	1	0	1
1	0	0	0	0	1	0	1
1	0	1	1	1	1	0	1
1	0	0	0	0	0	0	1
1	1	1	1	1	1	1	1

Esta consideración sobre si existe o no una entrada para la que se alcance el orden de la cota superior, junto con la descripción de tal entrada, en caso de existir, debe ser incluida en la documentación.

En cuanto a la ocupación de memoria, como ya se ha dicho, en realidad no necesitamos almacenar todos los tableros de todos los pasos, sino que nos basta con el tablero actual y el anterior. Teniendo en cuenta que también utilizaremos un tablero para almacenar el movimiento realizado desde cada casilla, tendremos que la memoria utilizada $\in o(4n^2)$.

Eso es todo lo que podemos decir sobre este algoritmo en el estudio teórico. En cuanto al estudio experimental, uno de los parámetros importantes a estudiar es cuál es el número de pasos real que se obtiene, en promedio, para ejecuciones del algoritmo sobre tableros aleatorios. Repetiremos el experimento varias veces para cada tamaño de tablero. Una posible forma de generar estos tableros aleatorios es fijar, mediante parámetros pasados al programa, el rango de valores entre los que se eligen los pesos de las casillas y generar estos aleatoriamente. También se ha de generar aleatoriamente la casilla inicial. Podemos hacerlo de forma totalmente aleatoria, o bien restringir el conjunto de casillas de entre las cuales podemos elegir una aleatoriamente. Por ejemplo, podríamos hacer un programa que acepte como parámetro el radio (en número de casillas) en torno al centro del tablero que determina la zona de casillas elegibles. Para estudiar el peor caso podemos generar tableros del tipo mostrado en la tabla anterior.

Haremos distintas pruebas para cada tamaño dado y probaremos distintos tamaños. Trataremos de sacar conclusiones sobre el comportamiento del número de pasos en función del tamaño de la entrada. También estudiaremos la influencia de la elección de la casilla inicial. Otro experimento interesante es comparar el tiempo empleado para distintas entradas concretas por el algoritmo de programación dinámica frente al resto de algoritmos.

Trataremos de determinar para qué tipo de entrada es mejor cada uno de los métodos y razonar el porqué.

E.3. Avance rápido

E.3.1. Resolución

Se pueden aplicar muchas ideas diferentes para buscar una aproximación del óptimo mediante avance rápido. Por ejemplo, podemos fijarnos en que, si los pesos de las casillas están distribuidos aleatoriamente, no es descabellado pensar que un camino de n casillas tenga un peso $p_{prom} * n$, donde p_{prom} es el peso promedio de las casillas del tablero. De ese modo un camino que recorra el número mínimo de casillas entre dos puntos nos puede dar una buena idea del coste del camino mínimo entre esas dos casillas. Dada una casilla cualquiera tendremos que hay una cierta distancia en línea recta desde ella a cada uno de los cuatro bordes. Elegimos el borde del que se encuentra más cerca. Si hay empate nos quedamos con todos los empatados y para cada uno de ellos aplicamos lo que sigue. Avanzamos desde la casilla original hasta el borde en cuestión eligiendo en cada paso, de entre las 3 casillas *en dirección hacia ese borde*¹, aquella que tenga menos peso. Si lo hicimos con varios bordes empatados al final nos quedaremos con el resultado de aquél que consiga un camino mejor. Veamos un ejemplo:

1	2	3	2	4	1
2	3	1	4	3	3
5	4	4	1	5	2
2	3	2	5	4	4
4	2	1	2	1	2
3	1	4	2	3	4

Supongamos que comenzamos en la casilla (3, 3) (con el peso, 4, en negrita). La distancia en número de casillas a los bordes superior e izquierdo es 3, a los otros dos es 4, luego trabajaremos con los dos primeros. Para llegar al borde superior desde (3, 3) en el primer paso elegimos entre 3 casillas con valores 3, 1 y 4. Gana la de 1, la (2, 3). La siguiente elección es entre 2, 3 y 2. Cualquiera de los dos gana, digamos que el primero: casilla (1, 2). Coste total del camino: $4 + 1 + 2 = 7$. Respecto al borde izquierdo, el camino elegido sería: (3, 3), (2, 2), (1, 1), con coste total 8. Nos quedamos con el primero, aproximación de coste de camino óptimo a partir de (3, 3): 7.

Esta es solo una posibilidad entre miles. Sobre esta misma idea podemos aplicar diferentes modificaciones:

- Por ejemplo, podríamos adoptar la decisión de hacer en cada casilla una estimación de bondad menos local examinando, en lugar de solo las casillas accesibles en un paso en cierta dirección, las casillas accesibles en 2 ó 3 pasos. Tendríamos un incremento

¹Si avanzamos, por ejemplo, hacia el borde izquierdo, las casillas *en dirección* al borde izquierdo son la que hay a la izquierda de la actual (avance en horizontal) y también la superior e inferior a esta última (avance en diagonal).

en el coste de la exploración, pero a cambio obtendremos con toda probabilidad una mejor aproximación al óptimo, dado que nuestro avance rápido tendrá un criterio menos local.

- Por otra parte, podríamos cambiar la forma en que se elige la dirección en la que elegimos explorar. Por ejemplo, en lugar de simplemente contar el número de casillas en línea recta hasta cada uno de los bordes, podríamos considerar la suma del peso de la primera casilla en dirección a cada borde más el producto del número de casillas restantes por el peso promedio del tablero. El cálculo del peso promedio requiere un preprocessamiento (de orden cuadrático): de nuevo añadimos un coste adicional a cambio de obtener alguna ventaja. Podríamos ampliar esta idea a considerar el peso de las 2 ó 3 primeras casillas en una cierta dirección, lo que supondría un mayor coste inicial, pero una exploración más profunda para tomar la decisión inicial, que presumiblemente nos permitiría elegirla mejor.

E.3.2. Estudio teórico y experimental

En este problema el tamaño de la entrada viene dado por n , el lado del cuadrado que constituye el tablero, y por la posición de la casilla inicial, que es la que determina la distancia menor en número de casillas al borde más cercano. Para no considerar el segundo factor, fijaremos siempre esta casilla en el centro del tablero –casilla central en un tablero de lado impar, o cualquiera de las cuatro centrales en un tablero de lado par.

El tiempo de ejecución es lineal respecto a n : valorar inicialmente las distancias en las cuatro direcciones posibles lleva, si utilizamos una tabla, un coste constante, k_1 , ya que podemos deducir esa distancia a partir de las coordenadas de la casilla. En cada paso del avance valoraremos 3, 5 u 8 casillas según estemos explorando simultáneamente en 1, 2 ó 4 direcciones (según la posición de la casilla inicial). En realidad, dada la elección prefijada de casilla inicial, la primera posibilidad queda descartada. En cualquier caso esto supone, para cada paso, un coste constante, k_2 . El número de pasos tiene que ver con la distancia inicial al borde, d , que según hemos dicho quedará fijado en función de n : $d = \lfloor \frac{n}{2} \rfloor$. De este modo $t(n) \in \Theta(\frac{k_2}{2}n)$, con $k_2 = 5$ o $k_2 = 8$ según n sea par o impar.

Si hubiésemos optado por evaluar desde cada casilla los dos siguientes pasos, entonces tendríamos que, si avanzamos en dos direcciones (n par), en cada casilla evaluaremos las 5 inmediatas en esas dos direcciones más 3 por cada una de ellas, es decir, 18 casillas, mientras que, si avanzamos en las cuatro direcciones (n impar), evaluaremos las 8 casillas adyacentes más 3 por cada una, en total 32 casillas. Es decir, $t(n) \in \Theta(\frac{k_2}{2}n)$, con $k_2 = 18$ o $k_2 = 32$ según n sea par o impar. En general, cuanto menos local sea el estudio nos supondrá un coste mayor. Hasta qué punto este incremento de coste merece la pena lo determinará la bondad de las soluciones obtenidas, y esto es algo que solo conoceremos mediante el estudio experimental.

Si optamos por un esquema en el que se lleva a cabo un cierto preprocessamiento, por ejemplo, la propuesta del apartado anterior para determinar inicialmente la dirección de avance, habrá que sumar ese coste al del propio avance rápido. Según el tipo de preprocessamiento realizado la σ pequeña se verá afectada o no.

En el estudio experimental nos fijaremos en la bondad de las soluciones obtenidas y

en los tiempos necesitados para obtenerlas para cada tamaño de problema. Para ello las compararemos con los óptimos reales, obtenidos por ejemplo por programación dinámica y, si hemos programado varios algoritmos de avance rápido, los compararemos entre sí. Esto nos permitirá determinar hasta qué punto merece la pena el tiempo adicional que necesita un cierto método frente a otro, en función de cuánto más se acercan las soluciones que da al óptimo real respecto al método menos costoso. Estudiaremos también si la bondad del algoritmo depende del tamaño de la entrada, de los valores de las casillas, etc. Veremos en qué condiciones es mejor cada uno de los algoritmos programados. Esto nos permitirá, en el apartado E.5, elegir el más adecuado.

E.4. Backtracking

E.4.1. Resolución

Podemos resolver este problema planteando un backtracking en el que cada nodo corresponda a una casilla (cada casilla podría corresponder a varios nodos) y genera como hijos a sus ocho casillas adyacentes. El nodo raíz correspondería a la casilla de partida. Sería solución cualquier nodo que corresponda a una casilla del borde. Se utilizaría una variable global VOA (valor óptimo actual) que se inicializaría a infinito y representa el coste del mejor camino al borde encontrado hasta ahora (camino que será almacenado en SOA, solución óptima actual). En la exploración en profundidad iríamos actualizando una variable, llamémosla C, que indica el costo hasta el momento del camino que estamos explorando. Cualquier camino para el que C supere ya a VOA sería descartado automáticamente por la función criterio. También serían descartados los caminos que incorporen ciclos, ya que obviamente hay un camino mejor sin el ciclo. Los ciclos pueden ser detectados manteniendo una tabla, con tantas casillas como el tablero, en la que podamos ir marcando y desmarcando las casillas visitadas.

La generación de los ocho hijos de cada nodo podría realizarse según un orden prefijado, por ejemplo, en el sentido de las agujas del reloj empezando por el de la derecha, o bien podría cambiar en cada nodo según algún criterio. Por ejemplo, se podría hacer algún tipo de estimación de bondad de cada uno de los ocho hijos y generarlos en el orden dado por ese criterio. Este criterio podría ser muy simple o bien requerir cierto procesamiento. Por ejemplo, podría consistir en aplicar un avance rápido para tomar como estimación de bondad de cada hijo la aproximación al óptimo dada por este método. Esto supone un gran coste adicional, dado que estamos aplicando esta estimación de bondad a cada nodo del árbol de un backtracking. Pero puede ocurrir que este tiempo adicional consiga encontrar enseguida soluciones muy próximas al óptimo, lo cual permitiría un mayor descarte de nodos por la función criterio, de modo que el tiempo de ejecución resultante se reduzca. Esto es algo que habrá que determinar experimentalmente.

Por otra parte podríamos utilizar funciones criterio más elaboradas que la básica, que solo elimina ciclos y caminos que llevan acumulado un coste superior a VOA. Por ejemplo, podríamos hacer un preprocessamiento que nos dé, para cada casilla, alguna cota inferior del coste del camino óptimo desde esa casilla al borde, llamémoslo c_m (camino mínimo). Con esta información podríamos refinar la función criterio: ahora eliminaríamos un nodo

cuando $C+cm > VOA$. Esto requeriría un coste adicional para el preprocesamiento, pero en este caso no es un coste añadido a cada nodo del árbol, sino una etapa inicial única, con lo que la probabilidad de que el tiempo adicional invertido merezca la pena es mayor. Un ejemplo de obtención de este tipo de cota inferior mediante preprocesamiento sería el siguiente: dividimos el tablero en anillos concéntricos, donde el primer anillo, el más exterior, corresponde a las casillas del borde; el segundo está formado por las adyacentes a las del primero; el tercero por las adyacentes al segundo, excluyendo las del primero ... y así sucesivamente. La cota inferior para las casillas del primer anillo corresponde al peso de la casilla en cuestión. Para cada casilla del segundo anillo tomaremos como cota la suma del peso de la propia casilla más el mínimo de los pesos de las casillas del primer anillo. Para cada casilla del tercero, la suma del peso de la casilla más el mínimo de las cotas del segundo anillo ... y así sucesivamente. Veamos como ejemplo el preprocesamiento efectuado sobre el mismo ejemplo utilizado para ilustrar el esquema avance rápido:

TABLERO

1	2	3	2	4	1
2	3	1	4	3	3
5	4	4	1	5	2
2	3	2	5	4	4
4	2	1	2	1	2
3	1	4	2	3	4

anillo 1 (min=1)						anillo 2 (min=2)						anillo 3					
1	2	3	2	4	1	-	-	-	-	-	-	-	-	-	-	-	-
2	-	-	-	-	-	3	-	4	2	5	4	-	-	-	-	-	-
5	-	-	-	-	-	2	-	5	-	-	6	-	-	6	3	-	-
2	-	-	-	-	-	4	-	4	-	-	5	-	-	4	7	-	-
4	-	-	-	-	-	2	-	3	2	3	2	-	-	-	-	-	-
3	1	4	2	3	4	-	-	-	-	-	-	-	-	-	-	-	-

E.4.2. Estudio teórico y experimental

Estudio teórico

En el caso del backtracking no podemos conocer teóricamente el tiempo de ejecución. Como mucho podemos aspirar a plantear una cota superior correspondiente al supuesto de que se recorriesen todos los nodos. Pero precisamente habremos de evitar por todos los medios que esto ocurra, sobre todo mediante la elección de una buena función criterio, ya que el tiempo obtenido es exponencial. Cada nodo genera un máximo de 8 hijos. En el primer nivel hay un nodo, en el segundo 8, en el tercero 64 ... y en el nivel enésimo 8^{n-1} . El número total de nodos es

$$\sum_{i=1}^n 8^{i-1} = \frac{8^n - 1}{7} \quad (\text{E.1})$$

En el estudio teórico de un backtracking hay que analizar también el tiempo de ejecución de cada una de las funciones utilizadas (*generar*, *criterio*, *solución*, *mashermanos* y *retroceder*). En nuestro caso todas son de coste constante. Si alguna función no lo fuese

habría que justificar muy bien por qué no existe una alternativa de coste inferior, ya que utilizar una función no constante en cada nodo de un árbol, con un número exponencial de nodos, es especialmente costoso.

Estudio experimental

Respecto al estudio experimental, lo más interesante es ver cuántos nodos se recorren realmente y el tiempo total empleado en encontrar la solución óptima. De estos dos datos podemos deducir también el tiempo promedio empleado en cada nodo. Estos indicadores nos permitirán determinar el coste y la bondad de las distintas funciones criterio implementadas, así como de los distintos órdenes en la generación de hijos. En este sentido es también interesante estudiar el ratio entre el número de nodos recorridos y el máximo número de nodos, así como comprobar si este ratio depende, para cada función criterio, del tamaño de la entrada.

Haremos pruebas con distintos tamaños de entrada y distintos tableros de partida aleatorios para cada uno. En un backtracking es especialmente importante hacer muchas pruebas para el mismo tamaño de entrada, ya que el número de nodos recorrido depende mucho de la entrada particular introducida. Respecto a la casilla de partida, obviamente influye mucho en el número de nodos recorrido. Podemos hacer pruebas distintas para, por ejemplo, casillas situadas aproximadamente en el centro y otras cercanas al borde.

Para comparar entre sí las distintas funciones criterio y los distintos órdenes de generación de hijos, compararemos los resultados de los indicadores mencionados aplicando las distintas posibilidades a las mismas entradas. Nos fijaremos en los tiempos de ejecución y valoraremos si el tiempo adicional introducido por una función criterio compensa, según la poda de nodos que consiga realizar. Para una misma entrada, la mejor función criterio será la que mejor tiempo de ejecución obtenga. De poco nos valdría una función que haga poda muy buena si, para hacerlo, invoca un nuevo backtracking del árbol completo en cada nodo.

E.5. Ramificación y poda

E.5.1. Resolución

Sobre el esquema del backtracking descrito en la sección anterior podemos construir un esquema de ramificación y poda. Podemos utilizar un esquema de ramificación LC-LIFO o LC-FIFO utilizando como función de estimación de coste el esquema de avance rápido visto anteriormente. Además ocurre que el valor óptimo estimado está basado en un camino real. De ese modo la estimación puede ser tomada como cota superior que podamos utilizar para descartar nodos cuya cota inferior (coste acumulado) supere la cota superior de cualquier otro nodo en la lista de nodos vivos (usaremos una variable CS, cota superior, que sea la menor de las cotas superiores de todos los nodos). Veamos un ejemplo del funcionamiento de este esquema (con LC-LIFO). Sea el siguiente tablero de pesos:

1	2	3	2	4
2	3	1	4	3
5	4	4	5	2
4	2	1	3	1
3	1	4	2	3

Esta sería la evolución de la lista de nodos vivos. Cada fila representa el estado tras haber ramificado el primer nodo de la lista de nodos vivos de la fila anterior, haber podado los hijos descartables, y haber insertado en la Lista de Nodos Vivos los restantes, ordenados según las estimaciones. En la primera fila no ha habido aún ramificación ni poda, simplemente se trata de la inicialización de la lista de nodos vivos con el nodo raíz. En la columna *CS* se indica la cota superior global empleada en esa fila para podar nodos. En la columna *Lista de Nodos Vivos* se muestra la situación de la Lista de Nodos Vivos tras la ramificación y poda correspondiente a esa fila, mientras que en la columna *Nodos Podados en ese paso* se muestran los nodos que han sido podados. La notación para los nodos es:

$$\text{númeroNodo}(fila, \text{columna})/\text{cotaInf}, \text{estimación}-\text{cotaSup}]$$

El número de nodo se asigna a medida que se van generando estos, siendo el orden de generación de los ocho hijos de un cierto nodo: $\uparrow, \nwarrow, \leftarrow, \swarrow, \downarrow, \searrow, \rightarrow, \nearrow$, es decir, en contra de las agujas del reloj empezando por la superior. Pero recuerda que a continuación los nodos se insertan en la lista de nodos vivos ordenados según el coste óptimo estimado. La fila y la columna indican a qué casilla corresponde el nodo, *cotaInf* indica la cota inferior del coste desde del camino desde el origen al borde a través de ese nodo (coste acumulado) y *estimación-cotaSup* es la estimación del camino a través de ese nodo, que al mismo tiempo actúa como cota superior por tratarse de una estimación basada en un camino real:

CS	Lista de Nodos Vivos	Nodos Podados en ese paso	Observaciones
	1(3,3)[4, 6]		Nodo inicial
6	6(4,3)[5, 6], 2(2,3)[5, 7]	3(2,2)[7,8], 4(3,2)[8,10], 5(4,2)[6,7], 7(4,4)[7,8], 8(3,4)[9,10], 9(2,4)[8,10]	Nodo 6 pasa delante del 2, mejor estimación
6	13(5,2)[6,6], 2(2,3)[5, 7]	10(3,3)[9,11], 11(3,2)[9,11], 12(4,2)[7,8], 14(5,3)[9,9], 15(5,4)[7,7], 16(4,4)[8,9], 17(4,4)[10,11]	Nodo 13 solución SOA=1-6-13, seguimos con resto LNV
6		18(1,3)[8,8], 19(7,7)[0,0], 20(2,2)[8,9], 21(9,11)[0,0], 22(3,3)[9,2], 23(3,4)[10,11], 24(2,4)[9,11], 25(1,4)[7,7]	LNV vacía, fin. Óptimo en SOA

Hemos visto un ejemplo de cómo asignar cotas y estimación de bondad de nodos, pero podríamos elegir muchas otras formas de hacerlo. Por ejemplo, podríamos decidir que la estimación va a ser la suma de las cotas inferior y superior dividida entre dos. O bien

podríamos refinar el cálculo de la cota inferior aplicando un esquema de preprocesamiento como el utilizado en backtracking en la sección anterior. Como siempre, añadir más procesamiento para calcular las cotas y estimaciones consume más tiempo, pero puede guiarnos más rápidamente hacia mejores soluciones, de modo que podamos descartar más nodos y finalmente el tiempo de ejecución será menor. Será el estudio experimental el que decida finalmente cuál es la mejor opción.

E.5.2. Estudio teórico y experimental

De forma parecida a lo que ocurría en el backtracking, en el estudio teórico sólo podemos calcular el número máximo de nodos y el tiempo empleado en cada uno de ellos. El número máximo de nodos coincide con el del backtracking, $\frac{8^n - 1}{7}$. En cuanto al tiempo empleado en cada nodo, en este caso hay que añadir el tiempo para calcular las estimaciones y cotas. Como utilizaremos el método de avance rápido de la sección E.3, el tiempo es el mismo que el calculado entonces, teniendo en cuenta que en este caso la casilla origen, que es parte del tamaño de la entrada para el método de avance rápido, variará según el nivel en que nos encontremos en el método de ramificación y poda.

En cuanto al estudio experimental, lo más interesante es estudiar el número de nodos recorridos en total gracias a las cotas y estimaciones utilizadas, así como el tiempo total empleado en encontrar la solución óptima y el tiempo por nodo. Si se utilizan varias estimaciones y cotas diferentes, se hará un estudio comparativo del número de nodos recorrido y el tiempo empleado por cada una de ellas ante el mismo problema. De forma similar, es interesante comparar el número de nodos y tiempo empleados por el backtracking, con las distintas funciones criterio, y el método de ramificación y poda, con las distintas funciones de cota y estimación.

De igual modo que con el backtracking, puede resultar interesante hacer un estudio de la relación existente entre el número máximo de nodos de un árbol y el número de nodos recorridos realmente analizados, estudiando si esta proporción depende del tamaño de la entrada.

Bibliografía

- [Aho74] Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman. *The design and analysis of computer algorithms*. Addison-Wesley, 1974.
- [Aho88] Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman. *Estructuras de datos y algoritmos*. Addison-Wesley, 1988.
- [Baase83] Sara Baase. *Computer algorithms. Introduction to design and analysis*. Addison-Wesley, 1983.
- [Baase00] Sara Baase, Allen Van Gelder. *Computer Algorithms. Introduction to Design and Analysis*. Addison-Wesley, 2000.
- [Brassard90] Brassard, Bratley. *Algorítmica. Concepción y análisis*. Masson, 1990.
- [Brassard97] Brassard, Bratley. *Fundamentos de algoritmia*. Prentice-Hall, 1997.
- [Campos95] Javier Campos Laclaustra. *Estructuras de datos y algoritmos*. Colección Textos Docentes, Prensas Universitarias de Zaragoza, 1995.
- [Collado87] Manuel Collado Machuca, Rafael Morales Fernández, Juan José Moreno Navarro. *Estructuras de datos, realización en Pascal*. Ediciones Díaz de Santos, 1987.
- [Cormen90] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.
- [Davis83] Martin D. Davis, Elaine J. Weyuker. *Computability, Complexity, and Languages*. Academic Press, 1983.
- [Drozdek01] Adam Drozdek. *Data Structures and Algorithms in Java*. Brooks/Cole, 2001.
- [Franch94] Manuel Franch Gutiérrez. *Estructuras de datos, Especificación, diseño e implementación*. Ediciones UPC, 1994.
- [Galve93] Javier Galve, Juan C. González, Ángel Sánchez, J. Ángel Velázquez. *Algorítmica, diseño y análisis de algoritmos Funcionales e Imperativos*. Ra-Ma, 1993.
- [Garey79] Michael R. Garey, David S. Johnson. *Computers and intractability. A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

- [Gonnet91] G. H. Gonnet, R. Baeza-Yates. *Handbook of Algorithms and Data Structures*. The MIT Press, 1991.
- [Gonzalo98] Julio Gonzalo Arroyo, Miguel Rodríguez Artacho. *Esquemas algorítmicos: enfoque metodológico y problemas resueltos*. Universidad Nacional de Educación a Distancia, 1998.
- [Harel97] David Harel. *Algorithms. The Spirit of Computing*. Addison-Wesley, 1997.
- [Heileman98] Gregory L. Heileman. *Estructuras de Datos, Algoritmos y Programación Orientada a Objetos*. McGraw-Hill, 1998.
- [Hernández01] Roberto Hernández, Juan Carlos Lázaro, Raquel Dormido, Salvador Ros. *Estructuras de Datos y Algoritmos*. Prentice Hall, 2001.
- [Hopcroft01] John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman. *Introducción a la Teoría de Autómatas, Lenguajes y Computación*. Addison-Wesley, 2001.
- [Hopcroft79] John E. Hopcroft, Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.
- [Horowitz78] Ellis Horowitz, Sartaj Sahni. *Fundamentals of Computer Algorithms*. Pitman, 1978.
- [Horowitz82] Ellis Horowitz, Sartaj Sahni. *Fundamentals of Data Structures*. Computer Science Press, 1982.
- [Humphrey01] Watt S. Humphrey. *Introducción al proceso software personal*. Pearson Educación, 2001.
- [Johnson90] David S. Johnson. *A Catalog of Complexity Classes*, in *Handbook of Theoretical Computer Science*, Volume A, Algorithms and Complexity, Jan van Leeuwen ed., pp 68-161. Elsevier, 1990.
- [Kernighan91] Brian W. Kernighan, Dennis M. Ritchie. *El lenguaje de programación C*. Prentice-Hall, 1991.
- [Knuth85] D. E. Knuth. *El arte de programar ordenadores. Vol 1: algoritmos fundamentales*. Reverté, 1985.
- [Knuth87] D. E. Knuth. *El arte de programar ordenadores. Vol 3: clasificación y búsqueda*. Reverté, 1987.
- [Kruse89] Robert L. Kruse. *Estructura de datos y diseño de programas*. Prentice-Hall, 1989.
- [Langsam97] Yedidyah Langsam, Moshe J. Augenstein, Aaron M. Tenenbaum. *Estructuras de datos con C y C++*. Prentice-Hall, 1997.
- [Lewis81] Harry R. Lewis, Christos H. Papadimitriou. *Elements of the theory of computation*. Prentice-Hall, 1981.

- [Manber89] Udi Manber. *Introduction to algorithms. A creative approach.* Addison-Wesley, 1989.
- [Meyer99] Bertrand Meyer. *Construcción de Software Orientado a Objetos. Segunda edición.* Prentice Hall, 1999.
- [Peña98] Ricardo Peña Marí. *Diseño de Programas. Formalismo y Abstracción.* Prentice-Hall, 1998.
- [Rabhi99] Fethi Rabhi, Guy Lapalme. *Algorithms, A Functional Programming Approach.* Addison-Wesley, 1999.
- [Stroustrup98] Bjarne Stroustrup. *El lenguaje de programación C++.* Addison Wesley, 1998.
- [Tenenbaum88] Aaron M. Tenenbaum, Moshe J. Augenstein. *Estructuras de datos en Pascal.* Prentice-Hall, 1988.
- [Troya84] José María Troya Linero. *Análisis y diseño de algoritmos.* VI Escuela de Verano de Informática. AEIA, 1984.
- [Weis95] Mark Allen Weiss. *Estructuras de datos y algoritmos.* Addison-Wesley, 1995.
- [Weis00] Mark Allen Weiss. *Estructuras de datos en Java.* Addison-Wesley, 2000.
- [Wirth80] Niklaus Wirth. *Algoritmos+Estructuras de datos=Programas.* Ediciones del Castillo, 1980.
- [Wirth87] Niklaus Wirth. *Algoritmos y estructuras de datos.* Prentice-Hall, 1987.
- [Wood87] Derick Wood. *Theory of Computation.* John Wiley & Sons, 1987.