Metodología y Tecnología de la Programación

Ingeniería en Informática

Curso 2008-2009

Esquemas algorítmicos. Programación dinámica

Yolanda García Ruiz **D228** **ygarciar@fdi.ucm.es**
Jesús Correas **D228** **jcorreas@fdi.ucm.es**

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

(elaborado a partir de [GV00], [BB97] y notas de S. Estévez y R. González del Campo)

Bibliografía

- **Importante**: Estas transparencias son un material de apoyo a las clases presenciales y no sustituyen a la bibliografía básica ni a las propias clases presenciales para el estudio de la asignatura
- Bibliografía básica:
  - [GV00]: capítulo 5
  - [BB97]: capítulo 8
- Ejercicios resueltos:
  - [MOV04]: capítulo 13

Esquemas algorítmicos. Programación dinámica

Características generales de la programación dinámica

- En el esquema *Divide y Vencerás* vimos una serie de problemas cuyas soluciones se pueden representar mediante la subdivisión en subproblemas
- Los subproblemas debían ser independientes
- En caso contrario, la complejidad de los algoritmos es exponencial, por la repetición de cálculos
- El objetivo de la programación dinámica consiste en resolver los subproblemas una sola vez, guardando las soluciones obtenidas hasta el momento en una tabla

Características generales de la programación dinámica (cont.)

- Podemos comparar la programación dinámica con otros métodos:
- Divide y vencerás:
  - El método *Divide y vencerás* es un método descendente, de *refinamiento progresivo*
  - La programación dinámica es una técnica *ascendente*, que soluciona de forma iterativa problemas típicamente recursivos
- Algoritmos voraces:
  - Los algoritmos *voraces* buscan la solución al problema mediante un conjunto de etapas y una elección voraz
  - La programación dinámica resuelve un caso a partir de otros de menor tamaño, y aplica el *principio de optimalidad*

Ejemplo: los números de Fibonacci

- La expresión matemática de la función de Fibonacci es la siguiente ($n \in \mathbb{N}$):

$$Fib(n) = \begin{cases} 1 & si\ \ n < 2 \\ Fib(n-1) + Fib(n-2) & si\ \ n \geq 2 \end{cases}$$

- Si implementamos un algoritmo recursivo de esta definición, el resultado es muy ineficiente ($\mathcal{O}(2^n)$)
- Podemos comprobar que la ineficiencia se debe a que se producen llamadas repetidas para calcular el resultado de la función.
- Por ejemplo:

$$Fib(5) = Fib(4) + Fib(3) = \underbrace{Fib(3) + Fib(2)}_{Fib(4)} + \underbrace{Fib(2) + Fib(1)}_{Fib(3)} = ...$$

Ejemplo: los números de Fibonacci (cont.)

- Podemos almacenar los resultados intermedios en una tabla:

Fib(0)	Fib(1)	Fib(2)	$\cdots$	Fib(n)

- Un algoritmo iterativo que calcula la sucesión de Fibonacci puede ser el siguiente:

```
fun FibIter1(n)
  crear tabla[0..n]
  si n≤1 entonces
    return n
  si no
    tabla[0] ← 0 ; tabla[1] ← 1
    desde i← 2 hasta n hacer
      tabla[i] ← tabla[i-1]+tabla[i-2]
    fin desde
    devolver tabla[n]
  fin si
fin fun
```

- La complejidad temporal de este algoritmo es lineal
- La complejidad espacial también
- ¿Cómo podría mejorarse este algoritmo?

Características generales de la programación dinámica (cont.)

- El término programación dinámica fue utilizado inicialmente en la década de 1940 para resolver problemas de optimización matemática
- En este tipo de problemas se define el Principio de optimalidad:

  En una secuencia de decisiones óptima toda subsecuencia ha de ser también óptima
- Sin embargo, este principio puede no ser aplicable a todos los problemas de optimización
- Cuando este principio no es aplicable, es posible que no se pueda utilizar este método algorítmico
- Ejemplo 3.4 de [NN98], p. 104

Características generales de la programación dinámica (cont.)

- El diseño de un algoritmo de este tipo consta de los siguientes pasos:
  - Planteamiento de la solución como una sucesión de decisiones (y verificación del principio de optimalidad)
  - Definición recursiva de la solución
  - Cálculo del valor de la solución óptima de forma *ascendente* (*bottom-up*), mediante una tabla donde se almacenan soluciones a problemas parciales para reutilizar cálculos
  - Construcción de la solución óptima con la información de la tabla anterior
- Como los resultados intermedios se almacenan en una tabla, es necesario estudiar la complejidad espacial, además de la complejidad temporal

Esquemas algorítmicos. Programación dinámica

1. Características generales de la programación dinámica
2. Ejemplo: Números de Fibonacci
3. Coeficientes Binomiales
4. Devolver el cambio
5. Problema de la mochila 0-1
6. Descenso del río
7. Algoritmo de Floyd
8. Multiplicación encadenada de matrices
9. Programación dinámica y recursión. Funciones con memoria

Coeficientes binomiales

- Este problema consiste en calcular el coeficiente binomial, definido por la expresión:

$$\binom{n}{k} = \begin{cases} 1 & si\ k = 0\ o\ k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & si\ 0 < k < n \\ 0 & \text{en caso contrario} \end{cases}$$

- Un algoritmo recursivo directo de esta expresión repite gran número de cálculos. Por ejemplo:

$$\binom{5}{3} = \binom{4}{2} + \binom{4}{3} = \underbrace{\binom{3}{1} + \binom{3}{2}} + \underbrace{\binom{3}{2} + \binom{3}{3}} =$$

$$\underbrace{\underbrace{\binom{2}{0} + \binom{2}{1}} + \underbrace{\binom{2}{1} + \binom{2}{2}} + \underbrace{\binom{2}{1} + \binom{2}{2}} + 1} = \ldots$$

Coeficientes binomiales (cont.)

- Podemos almacenar los resultados en una tabla (el *triángulo de Pascal*):

	0	1	2	$\cdots$	$k-1$	k
0	1					
1	1	1				
2	1	2	1			
$\cdots$						
$n-1$					$B(n-1,k-1)$	$B(n-1,k)$
n						$B(n,k)$

- Esta matriz se puede ir generando por filas, hasta llegar al valor deseado
- La complejidad temporal de este algoritmo está en $\Theta(nk)$
- La complejidad espacial está en $\Theta(nk)$ ¿Se podría mejorar?
- Ejercicio: realizar el esquema detallado del algoritmo

Esquemas algorítmicos. Programación dinámica

1. Características generales de la programación dinámica
2. Ejemplo: Números de Fibonacci
3. Coeficientes Binomiales
4. Devolver el cambio
5. Problema de la mochila 0-1
6. Descenso del río
7. Algoritmo de Floyd
8. Multiplicación encadenada de matrices
9. Programación dinámica y recursión. Funciones con memoria

Devolución del cambio

- En el tema de algoritmos voraces vimos un algoritmo eficiente para calcular el cambio con el número mínimo de monedas
- Desafortunadamente, ese algoritmo no funciona en todos los casos
- Por ejemplo, si tenemos un sistema monetario con las siguientes monedas: 1, 4, 6 unidades
- Debemos dar 8 unidades de cambio
- El algoritmo voraz calcularía 6+1+1, pues toma primero la moneda de mayor valor menor a 8
- Sin embargo, tomando dos monedas de 4 obtendríamos una solución mejor

Devolución del cambio (cont.)

- Podemos utilizar una tabla $C[1..n, 0..N]$ para almacenar los resultados de la devolución de cambio de cantidades intermedias
- Las filas de la tabla corresponden a los tipos de moneda. en nuestro ejemplo tres filas, correspondientes a las tres monedas: $d_1 = 1, d_2 = 4, d_3 = 6$
- Las columnas de la tabla corresponden a cantidades, desde 0 unidades hasta N unidades
- Cada elemento de la tabla, $C[i, j]$, contiene el número mínimo de monedas necesario para pagar j con monedas con valor desde d_1 hasta d_i
- La solución del problema de calcular número mínimo de monedas necesario para pagar una cantidad N con n monedas distintas es $C[n, N]$

Devolución del cambio (cont.)

- $C[i,0] = 0$ para todos los tipos de monedas $1 \leq i \leq n$
- Para calcular $C[i,j]$, $j > 0$, tenemos dos casos:
  - Si no utilizamos monedas de valor d_i, entonces basta con tomar el número mínimo de monedas necesario para obtener la misma cantidad, pero con monedas de valor hasta d_{i-1}. Esto es: $C[i-1,j]$
  - Si utilizamos (al menos) una moneda de valor d_i, entonces el valor del elemento $C[i,j]$ es $1 + C[i,j-d_i]$
- Se pueden producir ambos casos, pues el número mínimo de monedas puede venir dado por uno u otro caso, como en el ejemplo anterior
- Por tanto, el valor de $C[i,j]$ es

$$C[i,j] = \text{mín}(C[i-1,j], 1 + C[i,j-d_i])$$

- En determinados casos, los elementos a comparar caen fuera de los límites de la tabla: cuando $i = 1$, o bien cuando $d_1 > j$ (cantidad no expresable con el sistema monetario). En estos casos, podemos suponer que su valor es $+\infty$

Devolución del cambio (cont.)

- Un algoritmo que obtiene el número mínimo de monedas puede ser el siguiente:

```
fun monedas(N) // N es el importe a devolver
   crear d[1..n] = [1,4,6] // valores de las monedas
   crear C[1..n,0..N] // resultados intermedios
   desde i ← 1 hasta n hacer
      C[i,0] ← 0
   fin desde
   desde i ← 1 hasta n hacer
      desde j ← 1 hasta N hacer
         si i=1 Y j<d[i] entonces C[i,j]← +∞
         si no si i=1 entonces C[i,j] ← 1 + C[1,j-d[1]]
         si no si j<d[i] entonces C[i,j] ← C[i-1,j]
         si no C[i,j] ← mín(C[i-1,j],1+C[i,j-d[i]])
      fin desde
   fin desde
   devolver C[n,N]
fin fun
```

Devolución del cambio (cont.)

- Se puede modificar el algoritmo anterior para que además devuelva qué monedas de cada tipo forman el cambio
- Basta con ir recorriendo los elementos de la tabla C en sentido inverso a como se han construido, a partir de $C[n, N]$
- Este es un algoritmo voraz que se deja como ejercicio
- La complejidad del algoritmo anterior es la complejidad de construir una tabla de tamaño nxN, $\Theta(nN)$. Como el número de monedas n es una constante del problema, la complejidad es por tanto $\Theta(N)$.

Esquemas algorítmicos. Programación dinámica

1. Características generales de la programación dinámica
2. Ejemplo: Números de Fibonacci
3. Coeficientes Binomiales
4. Devolver el cambio
5. Problema de la mochila 0-1
6. Descenso del río
7. Algoritmo de Floyd
8. Multiplicación encadenada de matrices
9. Programación dinámica y recursión. Funciones con memoria

Problema de la mochila 0-1

- En el tema anterior vimos el problema de la mochila considerando que los elementos se podían fraccionar
- Si los elementos no se pueden fraccionar, entonces el algoritmo voraz visto anteriormente no funciona
- El enunciado del problema es el siguiente: debemos llevar un conjunto de objetos (elegidos de un conjunto de n objetos) en la mochila de forma que se maximice el valor de los objetos v_i, teniendo en cuenta que el peso w_i de los objetos seleccionados no puede superar la capacidad máxima de la mochila W
- Dicho de otra forma:

$$\text{maximizar } \sum_{i=1}^{n} x_i v_i \text{ con la restricción } \sum_{i=1}^{n} x_i w_i \leq W$$

- Como los objetos no son fraccionables, $x_i \in \{0, 1\}$

Problema de la mochila 0-1 (cont.)

- Vamos a seguir los pasos que se han indicado anteriormente para obtener el algoritmo
- En primer lugar, vamos a plantear el problema como una secuencia de decisiones que verifique el principio del óptimo
- Utilizaremos la expresión $V(i, w)$ para representar el valor máximo de la mochila con capacidad w cuando consideramos i objetos, $0 \leq w \leq W$, $1 \leq i \leq n$
- La solución a nuestro problema (valor máximo de la mochila) viene dada por $V(n, W)$.

Problema de la mochila 0-1 (cont.)

- Denominamos $d_1, ..., d_n$ a la secuencia de decisiones que nos permiten obtener el valor de $V(n, W)$, donde d_i puede tomar el valor 1 o 0 dependiendo de si se elige o no el elemento i. Por tanto, podemos tener dos situaciones:
  - $d_n = 1$. En este caso, $d_1, ..., d_{n-1}$ debe ser también óptima para el problema $V(n-1, W-w_n)$
  - $d_n = 0$. Entonces la subsecuencia $d_1, ..., d_{n-1}$ debe ser también óptima para el problema $V(n-1, W)$
- La relación de recurrencia es por tanto:

$$V(i,w) = \begin{cases} 0 & si\ i = 0\ y\ w \geq 0 \\ -\infty & si\ w < 0 \\ \text{máx}\{V(i-1,w), V(i-1,w-w_i)+v_i\} & \text{en otro caso} \end{cases}$$

- El algoritmo utiliza una matriz nxW para almacenar los resultados intermedios (V[1..n,0..W]), y dos vectores con los valores y los pesos de los objetos

Problema de la mochila 0-1 (cont.)

```
fun mochila(n,W,w[1..n],val[1..n])
   crear V[1..n,0..W]
   desde elem ← 1 hasta n hacer
      V[elem,0] ← 0
      desde cap ← 1 hasta W hacer
         si elem=1 Y cap < w[1] entonces
            V[elem,cap] ← 0
         si no si elem=1 entonces
            V[elem,cap] ← val[1]
         si no si cap<w[elem] entonces
            V[elem,cap] ← V[elem-1,cap]
         si no
            V[elem,cap] = Max(V[elem-1,cap],val[elem]+V[elem-1,cap-w[elem]]
         fin si
      fin desde
   fin desde
   devolver V[n,W]
fin fun
```

Problema de la mochila 0-1 (cont.)

- Se puede modificar el algoritmo para obtener la lista de los objetos que maximizan el valor de la mochila
- El procedimiento para obtenerlos es similar al del problema del cambio
- La complejidad para construir la tabla está en $\Theta(nW)$
- La complejidad para obtener la composición de la mochila está en $\Theta(n)$ (errata en [BB97])

Esquemas algorítmicos. Programación dinámica

1. Características generales de la programación dinámica
2. Ejemplo: Números de Fibonacci
3. Coeficientes Binomiales
4. Devolver el cambio
5. Problema de la mochila 0-1
6. Descenso del río
7. Algoritmo de Floyd
8. Multiplicación encadenada de matrices
9. Programación dinámica y recursión. Funciones con memoria

Descenso del río

- A lo largo de un río hay n embarcaderos. En cada uno de ellos se puede alquilar un bote que permite ir a cualquier otro embarcadero río abajo (no es posible ir río arriba). Existe una tabla de tarifas con el coste del viaje para ir desde cualquier embarcadero i a j $(i < j)$.
- Puede suceder que un viaje de i a j sea más caro que varios viajes más cortos (por ejemplo, de i hasta k, y después de k hasta j). No hay coste adicional por cambiar de bote.
- Debe diseñarse un algoritmo eficiente que calcule el coste mínimo para cada par de puntos i, j $(i < j)$

Descenso del río (cont.)

- $T[1..n, 1..n]$ es la tabla de tarifas: $T[i, j], i < j$ contiene la tarifa para ir del embarcadero i al j (de forma directa). Esta tabla es una matriz triangular superior.
- Se cumple el principio de optimalidad: las subsecuencias de la secuencia de coste mínimo de embarcaderos de i a j son también de coste mínimo.
- Representamos con $C(i, j)$ el coste de la secuencia embarcaderos de coste mínimo de i a j.
- $C(i, j)$ se puede calcular de forma recurrente, suponiendo que la primera parada se realiza en un embarcadero intermedio k, donde $i < k \leq j$:

$$C(i, j) = T(i, k) + C(k, j)$$

  En esta expresión se incluye el caso del viaje directo (cuando $k = j$).

Descenso del río (cont.)

- Debemos obtener el embarcadero k que minimiza esta expresión. Por tanto, podemos definir $C(i,j)$ como:

$$C(i,j) = \begin{cases} 0 & si\ i = j \\ \min\limits_{i<k\leq j} \{T(i,k) + C(k,j)\} & si\ i < j \end{cases}$$

- Para utilizar la técnica de programación dinámica, debemos almacenar los resultados intermedios de esta expresión recursiva en una estructura de datos. La estructura más sencilla es utilizar una matriz triangular de costes $C[i,j]$
- En este caso, es importante tener en cuenta la forma en que se deben obtener los valores de la tabla

Descenso del río (cont.)

- Para obtener $C(i,j)$, debe haberse calculado previamente $C(k,j)$, para todo k tal que $i < k < j$

	...	i	...	k	...	j	...
...							
i		0		$T(i,k)$		$C(i,j)$	
...							
k				0		$C(k,j)$	
...							
j						0	
...							

- No es posible rellenar la tabla por filas ni por columnas, pues es necesario haber calculado previamente elementos de la matriz que están en filas posteriores
- Una posible forma de obtener la matriz es rellenándola por diagonales

Descenso del río (cont.)

```
proc costes_embarcaderos(T[1..n,1..n],C[1..n,1..n])
   desde i ← 1 hasta n hacer
      C[i,i] ← 0
   fin desde
   desde diagonal ← 1 hasta n-1 hacer
      desde i ← 1 hasta n-diagonal hacer
         C[i,i+diagonal] = Min(T,C,i,i+diagonal)
      fin desde
   fin desde
fin proc

fun Min(T[1..n,1..n],C[1..n,1..n],i,j)
   min ← ∞
   desde k ← i+1 hasta j hacer
      si min > T[i,k]+C[k,j] entonces min ← T[i,k]+C[k,j]
   fin desde
fin fun
```

- ¿Qué complejidad temporal y espacial tiene este algoritmo?

Esquemas algorítmicos. Programación dinámica

1. Características generales de la programación dinámica
2. Ejemplo: Números de Fibonacci
3. Coeficientes Binomiales
4. Devolver el cambio
5. Problema de la mochila 0-1
6. Descenso del río
7. Algoritmo de Floyd
8. Multiplicación encadenada de matrices
9. Programación dinámica y recursión. Funciones con memoria

Algoritmo de Floyd

- Sea $G = \langle N, A \rangle$ un grafo dirigido, donde N es el conjunto de nodos y A el conjunto de aristas etiquetadas con longitudes no negativas
- Debe diseñarse un algoritmo que calcule la longitud del camino más corto entre cada par de nodos
- Suponemos que los nodos están numerados de 1 a n, $N = \{1, \ldots, n\}$, y que se proporcionan las longitudes de las aristas en una matriz de adyacencia $L[1..n, 1..n]$, donde
  - $L[i, i] = 0$ para $1 < i \leq n$
  - $L[i, j] \geq 0$ si existe una arista (i, j)
  - $L[i, j] = \infty$ si no existe la arista (i, j)

Algoritmo de Floyd (cont.)

- Un algoritmo que sistemáticamente comprobara todos los caminos entre todos los nodos tendría una complejidad peor que exponencial
- Podemos observar que se cumple el principio de optimalidad: si existe un nodo intermedio k en el camino mínimo desde i hasta j, entonces la parte del camino que va desde i hasta k es también un camino mínimo, igual que el camino desde k hasta j
- El resultado del algoritmo es una matriz $D[1..n, 1..n]$ que contiene las longitudes de los caminos mínimos entre cada par de nodos i, j
- Representamos con $D_k[1..n, 1..n]$ la matriz con los caminos mínimos entre nodos que pasan por los nodos $1, 2, \ldots, k$. El valor de cada elemento de esta matriz será:

$$D_k[i,j] = \text{mín}\{D_{k-1}[i,j], D_{k-1}[i,k] + D_{k-1}[k,j]\}$$

Algoritmo de Floyd (cont.)

Por ejemplo:

$$D_0 = L = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & \infty & 0 & 15 \\ 15 & \infty & 5 & 0 \end{pmatrix}$$

$$D_1 = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix} \quad D_2 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

$$D_3 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ 45 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix} \quad D_4 = \begin{pmatrix} 0 & 5 & 15 & 10 \\ 20 & 0 & 10 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

Algoritmo de Floyd (cont.)

- Según la expresión recurrente anterior, para calcular D_k es necesario mantener al menos la matriz de la iteración anterior, D_{k-1}
- Sin embargo, en la k-ésima iteración del algoritmo, los valores de la k-ésima fila y de la k-ésima columna de D_k no cambian, porque $D[k,k]$ siempre es 0
- Por tanto, para calcular

$$D_k[i,j] = \text{mín}\{D_{k-1}[i,j], D_{k-1}[i,k] + D_{k-1}[k,j]\}$$

no es necesario guardar D_{k-1}, pues solamente se utilizan de ella la fila k-ésima y la columna k-ésima, que no cambian

Algoritmo de Floyd (cont.)

- El algoritmo queda de la siguiente forma:

```
proc Floyd(L[1..n,1..n],D[1..n,1..n])
    D ← L // D_0 es la matriz de adyacencia
    desde k ← 1 hasta n hacer
        desde i ← 1 hasta n hacer
            desde j ← 1 hasta n hacer
                D[i,j] ← mín(D[i,j],D[i,k]+D[k,j])
            fin desde
        fin desde
    fin desde
fin proc
```

- La complejidad temporal está en $\Theta(n^3)$ ¿Y la complejidad espacial?
- Se puede comparar este algoritmo con el algoritmo de Dijkstra visto en el tema de métodos voraces

Algoritmo de Floyd (cont.)

- Para obtener los nodos intermedios de los caminos mínimos, utilizamos una matriz auxiliar P

```
proc Floyd2(L[1..n,1..n],D[1..n,1..n],P[1..n,1..n])
   desde i ← 1 hasta n hacer
      desde j ← 1 hasta n hacer
         P[i,j] ← 0
      fin desde
   fin desde
   D ← L // D_0 es la matriz de adyacencia
   desde k ← 1 hasta n hacer
      desde i ← 1 hasta n hacer
         desde j ← 1 hasta n hacer
            si D[i,k]+D[k,j] < D[i,j] entonces D[i,j] ← D[i,k]+D[k,j] ; P[i,j] ← k
         fin desde
      fin desde
   fin desde
fin proc
```

Algoritmo de Floyd (cont.)

- Se pueden obtener los nodos intermedios del camino desde i hasta j a partir de la matriz P mediante el siguiente algoritmo recursivo:

```
proc imprimir_camino(P[1..n,1..n],i,j)
   si P[i,j] ≠ 0 entonces
      imprimir_camino(P,i,P[i,j])
      imprimir P[i,j]
      imprimir_camino(P,P[i,j],j)
   fin si
fin proc
```

Esquemas algorítmicos. Programación dinámica

1. Características generales de la programación dinámica
2. Ejemplo: Números de Fibonacci
3. Coeficientes Binomiales
4. Devolver el cambio
5. Problema de la mochila 0-1
6. Descenso del río
7. Algoritmo de Floyd
8. Multiplicación encadenada de matrices
9. Programación dinámica y recursión. Funciones con memoria

Multiplicación encadenada de matrices

- Se desea calcular el producto de una serie de matrices (no cuadradas):

$$M = M_1 \cdot M_2 \cdot \ldots \cdot M_n$$

de forma que se minimice el número de multiplicaciones escalares
- El problema consiste en saber cuál es el número mínimo de multiplicaciones escalares que se deben realizar
- no consideramos la utilización de algoritmos como el de Strassen, sino el algoritmo tradicional de multiplicación de matrices
- En el producto de dos matrices $M \cdot N$, de dimensiones $p \times q$ y $q \times r$, respectivamente, el número necesario de multiplicaciones escalares es $p \cdot q \cdot r$
- No es posible cambiar el orden de las matrices, pues el producto de matrices no es conmutativo
- Sin embargo, el producto de matrices es asociativo, de forma que podemos seleccionar el orden en el que se realizan las multiplicaciones entre matrices

Multiplicación encadenada de matrices (cont.)

- Es un problema de optimización
- En el producto de más de dos matrices, el orden de realización de las operaciones entre matrices puede ser muy relevante respecto al número de multiplicaciones escalares
- Por ejemplo, consideremos el producto $A_{2\times5} \cdot B_{5\times3} \cdot C_{3\times4}$:
  - Si hacemos el producto $A_{2\times5} \cdot (B_{5\times3} \cdot C_{3\times4})$ se realizan $5 \cdot 3 \cdot 4 + 2 \cdot 5 \cdot 4 = 60 + 40 = 100$ multiplicaciones escalares
  - Si hacemos el producto $(A_{2\times5} \cdot B_{5\times3}) \cdot C_{3\times4}$ se realizan $2 \cdot 5 \cdot 3 + 2 \cdot 3 \cdot 4 = 30 + 24 = 54$ multiplicaciones escalares
- Las dimensiones de cada una de las matrices son conocidas. Utilizamos la notación $d_{i-1} \times d_i$ para representar las dimensiones de la matriz M_i
- Con $C(i,j)$ se representa el número mínimo de multiplicaciones escalares necesario para calcular el producto $M_i \cdot M_{i+1} \cdot \ldots \cdot M_j$
- El producto $(M_i \cdot \ldots \cdot M_k) \cdot (M_{k+1} \cdot \ldots \cdot M_j)$, para algún k $(i \leq k < j)$, tiene el siguiente número de multiplicaciones escalares:

$$C(i,j) = C(i,k) + C(k+1,j) + d_{i-1}d_k d_j \quad (C(i,i) = 0, \forall i) \qquad (1)$$

Multiplicación encadenada de matrices (cont.)

- Vemos que se cumple el principio de optimalidad:
  - Si $C(i,j)$ es el número mínimo de multiplicaciones escalares al multiplicar $M_i \cdot \ldots \cdot M_k \cdot \ldots \cdot M_j$, y éste se obtiene al agrupar $M_i \cdot \ldots \cdot M_k$ por un lado y $M_{k+1} \cdot \ldots \cdot M_j$ por otro, entonces:

  a) $C(i,k)$ es el número mínimo de multiplicaciones escalares de $M_i \cdot \ldots \cdot M_k$, y

  b) $C(k+1,j)$ es el número mínimo de multiplicaciones escalares de $M_{k+1} \cdot \ldots \cdot M_j$

- A partir de (1), se puede obtener la expresión recurrente que permite calcular $C(i,j)$:

$$C(i,j) = \begin{cases} 0 & si\ i = j \\ \min\limits_{i \le k < j} \{C(i,k) + C(k+1,j) + d_{i-1} d_k d_j\} & si\ i < j \end{cases}$$

- Para construir el programa dinámico que resuelva el problema, sustituimos las llamadas recurrentes por accesos a una estructura de datos

Multiplicación encadenada de matrices (cont.)

- La estructura de datos más sencilla es una matriz triangular superior $C[1..n, 1..n]$
- La solución del problema viene determinada por $C[1, n]$
- Como en el caso de estudio *Descenso del río*, el cálculo de los elementos de la matriz debe realizarse en un orden distinto del habitual:
  - Se puede recorrer la matriz por diagonales
  - También se pueden recorrer las filas de izquierda a derecha, pero empezando por la última

Multiplicación encadenada de matrices (cont.)

```
fun matrices(d[0..n]) //d es el vector de dimensiones
   crear C[1..n,1..n]
   desde i ← 1 hasta n hacer
      C[i,i] ← 0
   fin desde
   desde i ← n-1 hasta 1 sumando -1 hacer
      desde j ← i+1 hasta n hacer
         C[i,j] ← ∞
         desde k ← i hasta j-1 hacer
            C[i,j] ← mín(C[i,j],C[i,k]+C[k+1,j]+d[i-1]*d[k]*d[j])
         fin desde
      fin desde
   fin desde
   devolver C[1,n]
fin fun
```

- Se puede obtener el orden en el que se multiplican las matrices utilizando la misma técnica que en el problema *Descenso del río* o en el algoritmo de Floyd

Esquemas algorítmicos. Programación dinámica

1. Características generales de la programación dinámica
2. Ejemplo: Números de Fibonacci
3. Coeficientes Binomiales
4. Devolver el cambio
5. Problema de la mochila 0-1
6. Descenso del río
7. Algoritmo de Floyd
8. Multiplicación encadenada de matrices
9. Programación dinámica y recursión. Funciones con memoria

Programación dinámica y recursión. Funciones con memoria

- Los algoritmos de programación dinámica son generalmente eficientes
- Sin embargo, un algoritmo *ascendente* puede resultar poco natural, especialmente si se está habituado a programar utilizando refinamiento progresivo o recursión
- Por otra parte, es posible que un enfoque ascendente suponga realizar cálculos que no se utilicen para la solución final
- Es posible la combinación de las técnicas de refinamiento progresivo (y recursión) y programación dinámica
- Una forma bastante sencilla de hacer esto consiste en utilizar una *función con memoria*

Programación dinámica y recursión. Funciones con memoria (cont.)

- A la función recursiva se le añade una tabla del tamaño necesario
- Inicialmente, todas las entradas de esta tabla contienen un valor especial que indica que todavía no se han calculado
- Cada vez que se llama a la función recursiva, se examina primero la tabla para ver si el elemento correspondiente al argumento ya ha sido evaluado
  - Si ya se ha evaluado, se devuelve directamente el valor de la tabla
  - Si no se ha evaluado todavía, se calcula la función recursiva, y se almacena el valor obtenido en la tabla
- De esta forma, el algoritmo tiene la claridad de la formulación recursiva con una eficiencia parecida a la obtenida con programación dinámica

Programación dinámica y recursión. Funciones con memoria (cont.)

- Ejemplo: producto encadenado de matrices
- Se inicializa la tabla global mtab[1..n,1..n] al valor especial -1.

```
//mtab[1..n,1..n] es una tabla global con valor inicial -1
fun mat_mem(i,j,d[0..n])
   si i=j entonces devolver 0
   si mtab[i,j]≥ 0 entonces devolver mtab[i,j]
   m ← ∞
   desde k ← i hasta j-1 hacer
      m ← mín(m,mat_mem(i,k,d)+mat_mem(k+1,j,d)+d[i-1]*d[k]*d[j])
   fin desde
   mtab[i,j] ← m
   devolver m
fin fun
```