# Programación y Estructuras de Datos Avanzadas

## Capítulo 5: Programación Dinámica

# Capítulo 5: Programación dinámica

## 5.1 Planteamiento (aquí no hay un esquema general)

- La **programación dinámica** $\rightarrow$ técnica que reduce el coste (temporal) de ejecución de un algoritmo *memorizando soluciones parciales* (o resultados parciales que se utilizan repetidamente) con las que se llega a la solución final.
  - Aplicable a algoritmos "divide y vencerás" en los que se calculan datos de forma repetida $\rightarrow$ almacenaremos los resultados ya calculados en tablas para reutilizarlos.
  - También aplicable a problemas de optimización que cumplen el principio de *Optimalidad de Bellman*: "dada una secuencia óptima de decisiones, toda subsecuencia de ella es, a su vez, óptima" (similitud con algoritmos voraces)
  - **En general, los resultados parciales se almacenan en una tabla: los primeros datos corresponden a los subproblemas más sencillos y a partir de ellos se van construyendo de forma incremental soluciones para los subproblemas hasta llegar a la solución del problema completo.**
  - **La reducción del coste temporal $\rightarrow$ incremento del coste espacial.**

- Pasos en la **programación dinámica**:
  1. Establecimiento de las ecuaciones que representan el problema.
  2. Identificación de los resultados parciales.
  3. Construcción de la tabla de resultados parciales:
     - Inicialización de la tabla con los casos base que establece la ecuación del problema.
     - Establecimiento del orden de llenado de la tabla, de forma que se calculen, en primer lugar, los resultados parciales que se requieren en pasos posteriores.
  4. Sustitución de las llamadas recursivas del algoritmo por consultas a la tabla.

- Ejemplo (en este caso para un DyV): **construcción de la sucesión de Fibonacci**

Sucesión dada por:

$$Fibonacci(n) = \begin{cases} 1 & \text{si } n = 0,1 \\ Fibonacci(n-1) + Finonacci(n-2) & \text{si } n > 1 \end{cases}$$

Algoritmo DyV directo

```
fun Fib(n: entero): entero
    si n ≤ 1 entonces
        dev 1
    sino
        dev Fib(n-1) + Fib(n-2)
    fsi
ffun
```

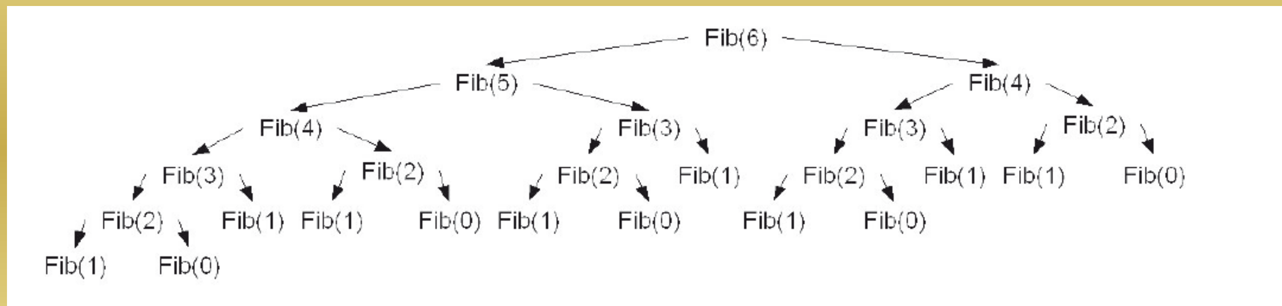
***Coste:***

- Tremendamente ineficiente porque descompone el tamaño de los subproblemas lentamente (reducción por substracción -de 1 ó 2-).
- No vale la fórmula general, pero estará entre:
  - 2 subproblemas de tamaño n-2: a=2, b=2, k=0 $\rightarrow$ $O(2^{n/2})$
  - 2 subproblemas de tamaño n-1: a=2, b=1, k=0 $\rightarrow$ $O(2^n)$

$\rightarrow$ **Coste exponencial: $O(cte^n)$**

- ¿Por qué un algoritmo tan sencillo es tan lento? (n=50 ya suele ser inabordable)

  → Realiza un montón de cálculos repetidos. Para calcular Fib(6) necesita Fib(5) y Fib(4). Pero para calcular Fib(5) vuelve a necesitar Fib(4)…




- Solución (prog. dinámica): los valores que se van calculando se almacenan en una tabla → en ese momento pasan a ser casos triviales (se reduce drásticamente el nº de llamadas recursivas)

  Coste (temporal) → O(n)

```
fun FibDin(n: entero): entero
    var
        i,suma: entero
        t: tabla[0..1] de entero
    fvar
    si n ≤ 1 entonces
        dev 1
    sino
        t[0] ← 1
        t[1] ← 1
        para i ← 2 hasta n hacer
            t[i] ← t[i-1] + t[i-2]
        fpara
    fsi
ffun
```

| Fibonacci(0) | Fibonacci(1) | ··· | Fibonacci(n) |
|---|---|---|---|

Almacena todos:
coste *espacial* → O(n)

```
fun FibDin2(n: entero): entero
    var
        i,suma,f,g: entero
    fvar
    si n ≤ 1 entonces
        dev 1
    sino
        f ← 1
        g ← 1
        para i ← 2 hasta n hacer
            suma ← f + g
            g ← f
            f ← suma
        fpara
    fsi
ffun
```

Almacena sólo los 2 anteriores:
Coste *espacial* → O(1)

# 5.2 Coeficientes binomiales

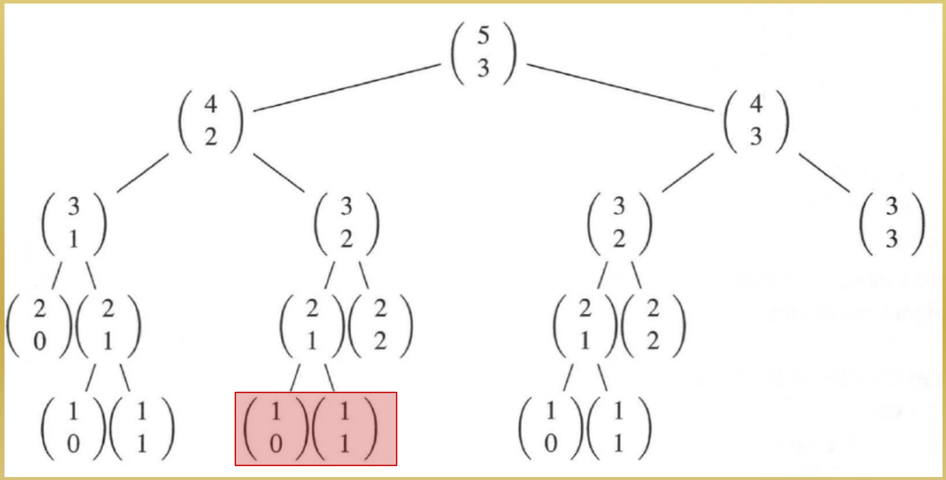
- Utilizados en combinatoria: el *coeficiente binomial de n sobre k*, es el número de subconjuntos de k elementos escogidos de un conjunto con n elementos:

$$C(n,k)=\binom{n}{k}=\frac{n!}{k!(n-k)!}$$

- Se pueden calcular de forma recursiva de la forma siguiente:

$$\binom{n}{k}=\begin{cases} 1 & \text{si } k=0 \text{ ó } k=n \\ \binom{n-1}{k-1}+\binom{n-1}{k} & \text{si } 0<k<n \end{cases}$$

- Si se implementa directamente esta ecuación → se repiten numerosos cálculos: *complejidad exponencial.* Por ejemplo, para el calcular el coeficiente de 5 sobre 3:



- ***Programación dinámica:*** lo mas eficiente es ir almacenando los coeficientes que se van calculando en una tabla. Los coeficientes calculados en forma ascendente $\rightarrow$ *Triángulo de Pascal*

$$\binom{0}{0}$$

$$\binom{1}{0} \quad \binom{1}{1}$$

$$\binom{2}{0} \quad \binom{2}{1} \quad \binom{2}{2}$$

$$\binom{3}{0} \quad \binom{3}{1} \quad \binom{3}{2} \quad \binom{3}{3}$$

$$\cdots$$

- *La tabla de n filas y k columnas se rellena de arriba abajo y de izquierda a derecha:*

| | 0 | 1 | 2 | 3 | ⋯ | $k-1$ | $k$ |
|---|---|---|---|---|---|---|---|
| 0 | 1 | | | | | | |
| 1 | 1 | 1 | | | | | |
| 2 | 1 | 2 | 1 | | | | |
| 3 | 1 | 3 | 3 | 1 | | | |
| ⋯ | ⋯ | | ⋯ | ⋯ | ⋯ | | |
| n-1 | ⋯ | | ⋯ | ⋯ | ⋯ | C(n-1,k-1) | C(n-1,k) |
| n | ⋯ | | ⋯ | ⋯ | ⋯ | ⋯ | C(n-1,k-1) + C(n-1,k) |

- ***Algoritmo***

```
fun CoefBin(n,k: entero): entero
    var
        i,j: entero
        t: Tabla[0..n] de entero
    fvar
    si k ≤ 0 ∨ k = n entonces
        dev 1
    sino
        para i ← 0 hasta n hacer t[i,0] ← 1 fpara
        para i ← 1 hasta n hacer t[i,1] ← i fpara
        para i ← 2 hasta k hacer t[i,i] ← 1 fpara
        para i ← 3 hasta n hacer
            para j ← 2 hasta n-1 hacer
                si j ≤ k entonces
                    t[i,j] ← t[i-1,j-1] + t[i-1,j]
                fsi
            fpara
        fpara
    fsi
ffun
```

***Coste del algoritmo***:

- **Coste espacial:** O(nk)
- **Coste temporal:** un elemento se obtiene sumando los dos superiores a él O(1), por lo que es el de la creación de la tabla → O(nk).

# 5.3 Devolución del cambio

- Conjunto de *N* tipos de monedas: $x_1, x_2, \ldots, x_N$
- **Objetivo:** *devolver una cantidad C utilizando el menor nº de monedas*
  - No siempre se puede resolver por voraz (depende el conjunto de monedas disponibles).
  - Utilizando programación dinámica vamos a poder resolverlo siempre (ojo con coste espacial).
- **Hay que plantear el problema de <u>forma incremental</u>:**
  - Suponemos que *comenzamos con la moneda de mayor valor $x_N$ y queremos devolver una cantidad C* $\rightarrow$ tenemos dos posibilidades:
    - Si $x_N > C$ entonces descartamos la moneda y pasamos a considerar la siguiente $x_{N-1}$
    - Si $x_N \leq C$ tenemos dos opciones:
      1) No tomar la moneda $x_N$ y completar la cantidad C con monedas de menor valor.
      2) Tomar la moneda $x_N$ y completar la cantidad restante $C-x_N$ con otras monedas.

Nos quedaremos con la que minimice el nº de monedas

$$cambio(N,C) = \begin{cases} cambio(N-1,C) & \text{si } x_N > C \\ min\{cambio(N-1,C), cambio(N,C-x_N)+1\} & \text{si } x_N \leq C \end{cases}$$

– Análogamente, para monedas de valores k menores que *N* y para cantidades C' menor que *C*:

$$cambio(k,C') = \begin{cases} cambio(k-1,C') & \text{si } x_k > C' \\ min\{cambio(k-1,C'), cambio(k,C'-x_k)+1\} & \text{si } x_k \leq C' \end{cases}$$

– Casos base:
  - Cuando hemos completado la cantidad *C*: cambio(k,0)=0 si 0≤k≤n
  - Cuando aún no hemos completado *C*, pero ya no quedan más monedas por considerar: cambio(0,C')=∞ si 0<*C'*≤*C*

– Se va construyendo la tabla empezando por los casos base y utilizando la ecuación de recurrencia. Así, *t*[*i*,*j*] será el número mínimo de monedas para dar la cantidad *j* (0≤*j*≤*C*), utilizando sólo monedas de los tipos entre 1 e i (0≤*i*≤*N*).

– La solución será el contenido de la casilla *t*[*N*,*C*].

- **Ejemplo**:
  - Deseamos pagar 12 unidades (C=12), utilizando 3 monedas de valores 1, 6 y 10.
  - Rellenamos la tabla por columnas:
    - desde la 0 a la 5ª columna sería:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $x_1 = 1$ | 0 | 1 | 2 | 3 | 4 | 5 | | | | | | | |
| $x_1 = 6$ | 0 | 1 | 2 | 3 | 4 | 5 | | | | | | | |
| $x_1 = 10$ | 0 | 1 | 2 | 3 | 4 | 5 | | | | | | | |

    - Para la columna 6ª t[1,6]=min(t[0,6],t[1,5]+1)=6, t[2,6]=min(t[1,6],t[2,0]+1)=min(6,1)=1 y t[3,6]=t[3-1,6]=1. Hasta la columna 9ª quedaría:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $x_1 = 1$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | | | |
| $x_1 = 6$ | 0 | 1 | 2 | 3 | 4 | 5 | 1 | 2 | 3 | 4 | | | |
| $x_1 = 10$ | 0 | 1 | 2 | 3 | 4 | 5 | 1 | 2 | 3 | 4 | | | |

    - Para la columna 10ª, t[1,10]=min(t[0,10],t[1,9]+1)=10, t[2,10]=min(t[1,10],t[2,4]+1)=min(10,4+1)=5, t[3,10]=min(t[2,10],t[3,0]+1)=min(5,1)=1
      La tabla final quedaría:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $x_1 = 1$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | |
| $x_1 = 6$ | 0 | 1 | 2 | 3 | 4 | 5 | 1 | 2 | 3 | 4 | 5 | 6 | 2 | |
| $x_1 = 10$ | 0 | 1 | 2 | 3 | 4 | 5 | 1 | 2 | 3 | 4 | 1 | 2 | $\boxed{2}$ | ← Solución |

## Algoritmo para rellenar la tabla (sólo nos indica el nº de monedas)

```
tipo Tabla = matriz[1..N,1..C] de entero
tipo Vector = matriz[0..N] de entero
fun DarCambio(C: entero, moneda: Vector): Tabla
    var
        t: Tabla
        i,j: entero
    fvar
    para i ← 1 hasta N hacer
        t[i,0] ← 0
    fpara
    para j ← 1 hasta C hacer
        para i ← 1 hasta N hacer
            si i = 1 ∧ moneda[i] > j entonces
                t[i,j] ← ∞
            sino
                si i = 1 entonces
                    t[i,j] ← 1 + t[1,j-moneda[i]]
                sino
                    si j < moneda[i] entonces
                        t[i,j] ← t[i-1,j]
                    sino
                        t[i,j] ← min(t[i-1,j], t[i,j-moneda[i]] + 1)
                    fsi
                fsi
            fsi
        fpara
    fpara
    dev t
ffun
```

## Algoritmo adicional que utiliza la tabla para indicar el nº de monedas de cada tipo

```
tipo Tabla = matriz[1..N,1..C] de entero
tipo Vector = matriz[0..N] de entero
fun seleccionar_monedas(C: entero, moneda: Vector, t:Tabla, seleccion: Vector)
    var
        i,j: entero
    fvar
    para i ← 0 hasta N hacer
        seleccion[i] ← 0
    fpara
    i ← N
    j ← C
    mientras j > 0 hacer
        si i > 1 ∧ t[i,j] = t[i-1,j] entonces
            i ← i - 1
        sino
            seleccion[i] ← seleccion[i] + 1
            j ← j - moneda[i]
        fsi
    fmientras
ffun
```

*Coste del algoritmo:*

- **Coste espacial:** $N \times (C+1)$ elementos en la tabla $\rightarrow$ O(NC)
- **Coste temporal:** el mayor es el de la creación de la tabla $\rightarrow$ O(NC).

# 5.4 El viaje por el río

- En un río hay *N* embarcaderos. En cualquiera de ellos se puede alquilar una barca para ir de un embarcadero a otro río abajo.
- Las tarifas de alquiler de las barcas dependen de cada par de embarcaderos: $T_{ij} \rightarrow$ coste para ir del embarcadero i al j.

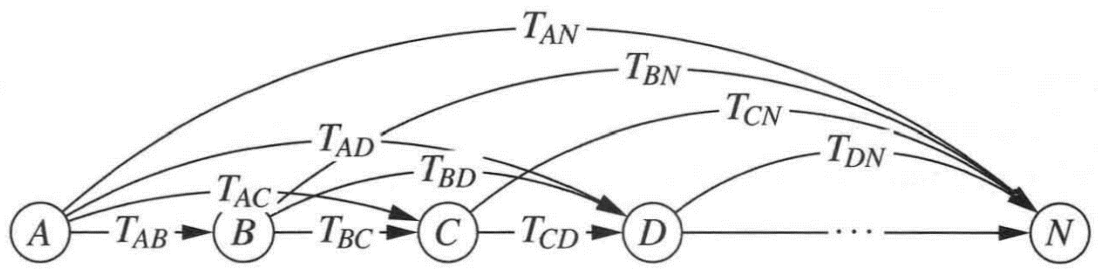



Figura 5.1: *Ejemplo de una serie de embarcaderos A, B, ··· , N por el río. Los arcos indican los costes de los distintos trayectos.*

- A veces el alquiler entre dos embarcaderos i y j es más costoso que el alquiler de barcas para una sucesión de viajes más cortos a embarcadores intermedios.
- **Objetivo:** calcular de forma eficiente la forma más barata de ir de un embarcadero i a otro j (j>i, porque las barcas no pueden ir río arriba).

- El coste de ir del embarcadero $e_i$ hasta el embarcadero $e_j$ pasando por el embarcadero intermedio viene dado por:

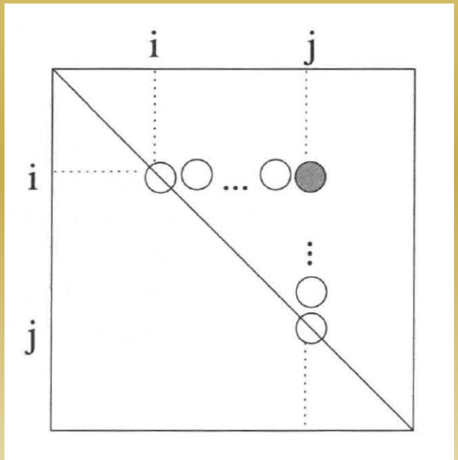
$$C(i,j) = T[e_i, e_k] + C(e_k, e_j)$$

- *La ecuación de recurrencia calcula el mínimo del coste de todas las combinaciones posibles de proyectos haciendo escala en algún embarcadero intermedio k (aunque k puede ser j para comprobar también el viaje directo)*

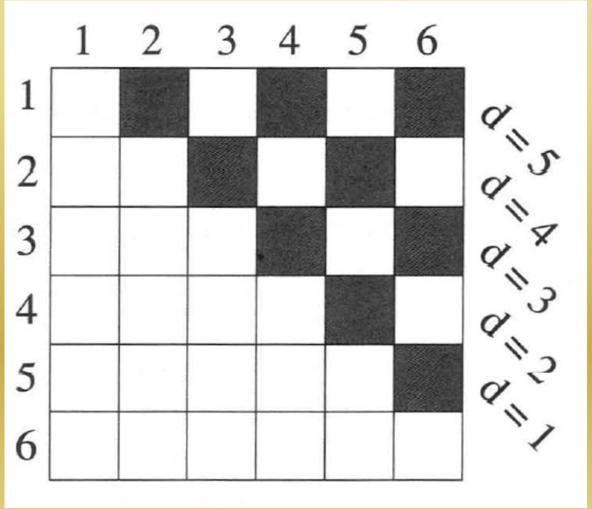
$$C(i,j) = \begin{cases} 0 & \text{si } i = j \\ min_{i<k\leq j}\{T[i,k] + C(k,j)\} & \text{si } i < j \end{cases}$$

- **Estructura para almacenar las soluciones parciales:** matriz *C* de tamaño *N*×*N*

  - Sólo necesitamos la mitad por encima de la diagonal principal.
  - Para calcular la casilla T[i,j] se utilizan las casillas de la fila i entre la diagonal y la columna j, y las casillas de la columna j entre la diagonal y la fila i.
  - Una forma de construir la tabla es por "diagonales" de la mitad superior de la matriz.



- ***Ejemplo:*** para *N*=6, se completarían las "diagonales" *d*=1, 2, 3, 4, 5 y 6 mostradas en la figura.




- **Algoritmo (sólo calcula el coste mínimo):**

```
tipo Tabla = matriz[1..N,1..N] de entero
fun ViajeRio(T: Tabla, N: entero, C: Tabla)
    var
        i,diag: entero
    fvar
    para i ← 1 hasta N hacer
        C[i,i] ← 0
    fpara
    para diag ← 1 hasta N-1 hacer
        para i ← 1 hasta N-diag hacer
            C[i,i+diag] ← MinMultiple(C,i,i+diag)
        fpara
    fpara
ffun
```

```
fun MinMultiple(C: Tabla, i: entero, j: entero): entero
    var
        k, minimo: entero
    fvar
    minimo ← ∞
    para k ← i+1 hasta j hacer
        minimo ← min(minimo, C[k,j] + T[i,k])
    fpara
    dev minimo
ffun
```

***Coste del algoritmo:***

- **Coste espacial:** N×N/2 → O($N^2$)
- **Coste temporal:** 2 bucles anidados que invocan MinMultiple O(n) → O($N^3$)

- Nos puede interesar no saber sólo el coste mínimo ***sino también el plan de viaje de embarcaderos asociado a dicho coste mínimo*** → se define una nueva **tabla E[1..N,1..N]** que registre el embarcadero en el que hemos tenido que hacer escala para alcanzar el coste mínimo.

- **Algoritmo modificado → ViajeRio2:**

```
tipo Tabla = matriz[1..N,1..N] de entero
fun ViajeRio2(T: Tabla, N: entero, var C: Tabla, var E: Tabla)
    var
        i,diag,min,emb: entero
    var
    para i ← 1 hasta N hacer
        C[i,i] ← 0
    fpara
    para diag ← 1 hasta N-1 hacer
        para i ← 1 hasta N-diag hacer
            MinMultiple2(C,i,i+diag,min,emb)
            C[i,i+diag] ← min
            E[i,i+diag] ← emb
        fpara
    fpara
ffun
```

```
fun MinMultiple2(C: Tabla, i: entero, j: entero, minimo: entero, emb: entero
    var
        k: entero
    var
    minimo ← ∞
    emb ← i
    para k ← i+1 hasta j hacer
        si C[k,j] + T[i,k]) < minimo entonces
            minimo ← C[k,j] + T[i,k]
            emb ← k
        fsi
    fpara
ffun
```

# 5.5 La mochila entera (0/1)

- **Datos del problema:**
  - **n**: número de objetos disponibles
  - **V**: capacidad máxima de la mochila.
  - **v = ($v_1$, $v_2$, ..., $v_n$)** volúmenes de los objetos ***(restricción $\rightarrow$ han de ser enteros, ¡si son reales usaremos ramificación y poda!)***.
  - **b = ($b_1$, $b_2$, ..., $b_n$)** beneficios de los objetos.

- **Objetivo:** llenar la mochila de manera que se maximice el valor total de los objetos almacenados en ésta.

- **Formulación matemática:**

$$\text{Maximizar} \sum_{i=1..n} x_i b_i \text{, sujeto a la restricción} \sum_{i=1..n} x_i v_i \leq V \;\; \text{y} \;\; x_i \in \{0,1\}$$

- **¿Esquema algorítmico?:** *Al no poder dividirse los objetos $\rightarrow$ no se puede aplicar el esquema voraz.*

- **Para usar programación dinámica hay que formular el problema de forma incremental.**

## Planteamos el problema de forma incremental:

- **mochila(i,W)=**máximo beneficio para un volumen W disponible en la mochila considerando sólo los objetos entre 1 e i.
- Cuando pasamos a considerar el objeto i hay dos posibilidades:
  - Si $v_i$>W (el objeto excede la capacidad de la mochila) $\rightarrow$ se descarta y pasamos a considerar los siguientes objetos (de 1 a i-1)
  - Si $v_i \leq$W (el objeto cabe en el hueco restante) $\rightarrow$ 2 posibilidades:

Nos quedaremos con la opción que maximice el beneficio total

1) No incluir el objeto pasando a considerar los siguientes (de 1 a i-1).
2) Incluir el objeto, añadiendo el beneficio $b_i$ al valor de la función y restando $v_i$ del espacio libre.

- **Formulamos la ecuación de recurrencia (incluyendo los casos triviales)**:

$$\text{mochila}(i,W) = \begin{cases} 0 & \text{si } i=0 \ \textbf{y}\ W \geq 0 \\ -\infty & \text{si } W < 0 \\ \text{mochila}(i-1,W) & \text{si } i>0 \ \textbf{y}\ v_i > W \\ max\{\text{mochila}(i-1,W), & \text{si } i>0 \ \textbf{y}\ v_i \leq W \\ \qquad b_i + \text{mochila}(i-1,W-v_i)\} & \end{cases}$$

*Se cumple el principio de optimalidad, si el problema se resuelve de forma óptima también se tienen que haber resuelto de forma óptima sus subproblemas*

## Construcción de la tabla:

- M[n,V] con tantas filas como objetos y tantas columnas como indique el volumen V.
- **Necesario que los volúmenes sean valores enteros.**
- Para calcular la posición M[i,j] necesitamos haber calculado 2 de las posiciones de la fila anterior.
- La solución viene dada por el valor de M[n,V].

```
tipo Tabla = matriz[0..n,0..V] de entero
tipo Vector = matriz[0..n] de entero
fun MochilaEntera(vol:Vector, ben:Vector, n: entero, V: entero, M:Tabla)
    var
        i,j: entero
    fvar
    para i ← 1 hasta n hacer
        M[i,0] ← 0
    fpara
    para j ← 1 hasta V hacer
            M[0,j] ← 0
    fpara
    para i ← 1 hasta n hacer
        para j ← 1 hasta V hacer
            si vol[i] > j entonces
                M[i,j] ← M[i-1,j]
            sino
                M[i,j] ← max(M[i-1,j], M[i-1,j-vol[i]] + ben[i])
            fsi
        fpara
    fpara
ffun
```

*Coste del algoritmo:*

- **Coste espacial:** $(n+1)\times(V+1)$ elementos en la tabla $\rightarrow O(nV)$
- **Coste temporal:** $O(nV)$

- **Ejemplo**: V=8, n=5, v={1,3,4,5,7}, b={2,5,10,14,15}.
  - Rellenamos la tabla por columnas (de arriba abajo y de izquierda a derecha):

| Límite de volumen | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | |
|---|---|---|---|---|---|---|---|---|---|---|
| posición 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| $v_1 = 1,\ b_1 = 2$ | 0 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | |
| $v_2 = 3,\ b_2 = 5$ | 0 | 2 | 2 | 5 | 7 | 7 | 7 | 7 | 7 | |
| $v_3 = 4,\ b_3 = 10$ | 0 | 2 | 2 | 5 | 10 | 12 | 12 | 15 | 15 | |
| $v_4 = 5,\ b_4 = 14$ | 0 | 2 | 2 | 5 | 10 | 14 | 16 | 16 | 19 | |
| $v_5 = 7,\ b_5 = 15$ | 0 | 2 | 2 | 5 | 10 | 14 | 16 | 16 | $\boxed{19}$ | ← Solución |

- Si queremos que además del nº mínimo de monedas devuelva las monedas utilizadas, utilizo objetos[i] que indica si el objeto i se incluye o no.

```
fun ObjetosMochila(vol: vector, M:Tabla, n:entero, V:entero, objetos: Vector)
    var
        i,W: entero
    fvar
    W ← V
    para i ← n hasta 1 incremento -1 hacer
        si M[i,W] = M[i-1,W] entonces
            objetos[i] ← 0      ← No incluyo el objeto
        sino
            objetos[i] ← 1      ← Incluyo el objeto cuando varía el
                                  valor al cambiar de fila
            W ← W - vol[i]
        fsi
    fpara
ffun
```

# 5.6 Multiplicación asociativa de matrices.

- Tenemos que calcular el producto de *N* matrices cuyo resultado será la matriz *M*.

Estas dimensiones deben coincidir

- El producto de la matriz A(mxn) por la matriz B(nxp) es una matriz C(mxp) de componentes:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}, 1 \leq i \leq m, 1 \leq j \leq p$$

Esta multiplicación requiere *m*x*n*x*p* productos escalares.

- El producto de matrices no es conmutativo, pero sí es asociativo, es decir, se verifica que (AB)C=A(BC)
- **Ejemplo:**

Número de multiplicaciones escalares en función del orden del producto de matrices

| Matriz | Dimensiones |
|---|---|
| A | $60 \times 2$ |
| B | $2 \times 30$ |
| C | $30 \times 5$ |
| D | $5 \times 20$ |

| Producto | Coste | Cálculo |
|---|---|---|
| ((AB)C)D | 18600 | $60 \times 2 \times 30 + 60 \times 30 \times 5 + 60 \times 5 \times 20$ |
| A((BC)D) | 2900 | $2 \times 30 \times 5 + 2 \times 5 \times 20 + 60 \times 2 \times 20$ |
| (AB)(CD) | 42600 | $60 \times 2 \times 30 + 30 \times 5 \times 20 + 60 \times 30 \times 20$ |
| A(B(CD)) | 6600 | $30 \times 5 \times 20 + 2 \times 30 \times 20 + 2 \times 20 \times 60$ |
| (A(BC))D | 6900 | $2 \times 30 \times 5 + 60 \times 2 \times 5 + 60 \times 5 \times 20$ |

- **Objetivo:** buscar orden óptimo de asociación en el producto de varias matrices para minimizar el nº de multiplicaciones escalares.
- Sup. tenemos una secuencia de matrices a multiplicar $M_1$, $M_2$, .., $M_n$ (dimensiones de $M_i$: $d_{i-1}$x$d_j$)
- Sea E(i,j) el nº mínimo de multiplicaciones escalares necesarias para realizar el producto de las matrices $M_i$, $M_{i+1}$, …, $M_j$, $1 \leq i \leq j \leq n$. *Un modo de realizar este producto es:*

$$(M_i M_{i+1} \cdots M_k)(M_{k+1} \cdots M_j)$$

Nº productos escalares: $E(i,k)$ $E(k+1,j)$

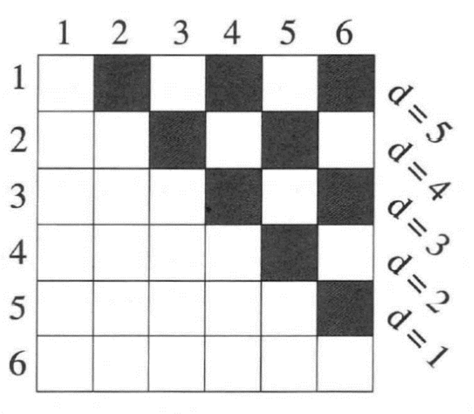
Dimensión resultante: $d_{i-1}$x$d_k$ $d_k$x$d_j$

- *El objetivo es elegir k, para minimizar el nº de productos escalares.* Planteamos la siguiente ecuación de recurrencia:

$$E(i,j) = \begin{cases} min_{i \leq k < j}\{E(i,k) + E(k+1,j) + d_{i-1}d_k d_j\} & \text{si } i < j \\ 0 & \text{si } i = j \end{cases}$$

*Como en el problema del viaje por el río por lo que rellenaremos la tabla por diagonales. Empezaremos por d=1 que está justo encima de la diagonal principal, hasta d=n-1. La fila de los elementos de cada diagonal varía entre i=1 e i=n-d.*

- ***Ejemplo:*** para *N*=6, se completarían las "diagonales" *d*=1, 2, 3, 4, 5 y 6 mostradas en la figura.




- **Algoritmo (sólo calcula el nº mínimo de multiplicaciones:**

```
tipo Tabla = matriz[1..N,1..N] de entero
tipo Vector = matriz[0..N] de entero
fun MultMatrices(d: Vector, N: entero, mult: Tabla)
    var
        i,diag: entero
    fvar
    para i ← 1 hasta N hacer
        mult[i,i] ← 0
    fpara
    para diag ← 1 hasta N-1 hacer
        para i ← 1 hasta N-diag hacer
            mult[i,i+diag] ← MinMultiple(mult,d,i,i+diag)
        fpara
    fpara
ffun
```

```
fun MinMultiple(mult: Tabla, d: Vector, i: entero, j: entero): entero
    var
        k, minimo: entero
    fvar
    minimo ← ∞
    para k ← i hasta j-1 hacer
        minimo ← min(minimo, mult[i,k] + mult[k+1,j] + d[i-1]*d[k]*d[j])
    fpara
    dev minimo
ffun
```

*¡Solución en mult[1,N]!*

***Coste del algoritmo:***
- **Coste espacial:** N×N/2 → O($N^2$)
- **Coste temporal:** 2 bucles anidados que invocan MinMultiple O(n) → O($N^3$)

- Nos puede interesar no saber sólo el nº mínimo de multiplicaciones, ***sino también el parentizado que nos permite obtenerlas*** → se define una nueva tabla **pos[1..N]** que almacena la posición en la que se coloca el paréntesis. *EscribeParentizado* escribe esta asociación de productos de matrices

- **Algoritmo modificado → MultMatrices2:**

```
tipo Tabla = matriz[1..N,1..N] de entero
tipo Vector = matriz[0..N] de entero
fun MultMatrices2(d: Vector, N: entero, mult: Tabla, pos: Tabla)
    var
        i,diag: entero
    fvar
    para i ← 1 hasta N hacer
        mult[i,i] ← 0
    fpara
    para diag ← 1 hasta N-1 hacer
        para i ← 1 hasta N-diag hacer
            MinMultiple2(mult,d,i,i+diag, minimo, posicion)
            mult[i,i+diag] ← minimo
            pos[i,i+diag] ← posicion
        fpara
    fpara
    EscribeParentizado(pos, 1, N)
ffun
```

*MinMultiple2 se modifica para registrar la posición del paréntesis*

```
fun MinMultiple2(mult: Tabla, d: Vector, i: entero, j: entero, minimo: entero,
                 pos: entero)
    var
        k, tmp: entero
    fvar
    minimo ← ∞
    pos ← i
    para k ← i hasta j-1 hacer
        tmp ← mult[i,k] + mult[k+1,j] + d[i-1]*d[k]*d[j]
        si tmp < minimo entonces
            minimo ← tmp
            pos ← k
        fsi
    fpara
ffun
```

```
fun EscribeParentizado(pos: Tabla, i: entero, j: entero)
    var
        k: entero
    fvar
    si i = j entonces
        Imprimir "M", Imprimir i
    sino
    k ← pos[i,j]
    Imprimir "("
    EscribeParentizado(pos,i,k)
    EscribeParentizado(pos,k+1,j)
    Imprimir ")"
ffun
```

# 5.7 Camino de coste mínimo entre nodos de un grafo dirigido

- Sea G=<N,A> grafo dirigido compuesto por *N* nodos y un conjunto de aristas *A* que tienen asociada una longitud no negativa.
- **Objetivo:** calcular la longitud del camino más corto entre cada par de nodos del grafo.
- **Algoritmo voraz:** aplicar el algoritmo de Dijkstra $O(N^2)$ para cada uno de los nodos $\rightarrow$ $O(N^3)$
- **Programación dinámica:** algoritmo de Floyd, también del orden $O(N^3)$. Planteamiento de la ecuación de recurrencia:
  - **M(i,j,k)**: coste mínimo de ir del nodo i al nodo j pudiendo utilizar como nodos intermedios los que están entre 1 y k.
  - **El camino más corto entre los vértices i y j** cuando consideramos un vértice intermedio k puede corresponder a una de las dos siguientes posibilidades:
    - Puede pasar por k, lo que nos lleva a buscar los caminos óptimos entre i y k, y entre k y j (ninguno de los caminos intermedios puede pasar por k puesto que formaría un ciclo).
    - Puede no pasar por k, y entonces hay que considerar el mejor camino de i a j utilizando sólo como posibles vértices intermedios los que están entre 1 y k-1.
  - **El caso base** se alcanza con $k$=0 y corresponde a ir de i a j sin pasar por nodos intermedios (si i=j este coste es 0, sino la longitud de la arista de i a j A[i,j]).
  - La **solución** buscada será **M(i,j,n)**, coste mínimo para ir de i a j considerando cualquiera del resto de los nodos como intermedios.

## ➢ Ecuación de recurrencia:

$$M(i,j,k) = \begin{cases} 0 & \text{si } k = 0 \text{ y } i = j \\ A[i,j] & \text{si } k = 0 \text{ y } i \neq j \\ min(M(i,j,k-1), & \text{si } k > 0 \\ \quad M(i,k,k-1) + M(k,j,k-1)) & \end{cases}$$

- Para calcular un nuevo elemento de la tabla sólo se necesitan los datos correspondientes a haber incluido el nodo k-1.
- Es posible utilizar una única tabla $N \times N$ que se va actualizando a medida que se incluyen nuevos nodos → coste espacial $O(N^2)$.
- Se parte del caso base k=0 y se construye la matriz → ***algoritmo de Floyd***

***Coste del algoritmo***:

- **Coste espacial:** $O(N^2)$
- **Coste temporal:** $O(N^3)$ pero operaciones más simples que Dijkstra por lo que las constantes ocultas son más pequeñas.

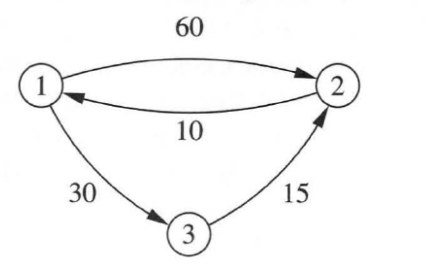
  Sin embargo, para *grafos dispersos*, el algoritmo de *Dijkstra* puede optimizarse utilizando listas de adyacencia por lo que sería $O(N^2 \log N)$

❖ Grafos densos → Algoritmo de Floyd

❖ Grafos dispersos → Algoritmo de Dijkstra

```
tipo Tabla = matriz[1..N,1..N] de entero
fun Floyd(A: Tabla, N: entero, M: Tabla)
    var
        i, j, k: entero
    fvar
    para i ← 0 hasta N hacer
        para j ← 0 hasta N hacer
            M[i,j] ← A[i,j]
        fpara
    fpara
    para k ← 1 hasta N hacer
        para i ← 1 hasta N hacer
            para j ← 1 hasta N hacer
                M[i,j] ← min(M[i,j], M[i,k] + M[k,j])
            fpara
        fpara
    fpara
ffun
```

- **Ejemplo**: grafo de 3 nodos




- **Tabla para k=0** (=matriz de adyacencia del grafo):

| | 1 | 2 | 3 |
|---|---|---|---|
| 1 | 0 | 60 | 30 |
| 2 | 10 | 0 | ∞ |
| 3 | ∞ | 15 | 0 |

- **Tabla para k=1**, añadimos el nodo 1. En este caso se encuentra un camino entre los nodos 2 y 3 que pasa por el nodo 1 y tiene una longitud de 40.

| | 1 | 2 | 3 |
|---|---|---|---|
| 1 | 0 | 60 | 30 |
| 2 | 10 | 0 | 40 |
| 3 | ∞ | 15 | 0 |

- **Tabla para k=2**, añadimos el nodo 2. Tenemos un camino del nodo 3 al 1 que pasa por el nodo 2 y tiene un coste 25.

| | 1 | 2 | 3 |
|---|---|---|---|
| 1 | 0 | 60 | 30 |
| 2 | 10 | 0 | 40 |
| 3 | 25 | 15 | 0 |

- **Tabla para k=3**, añadimos el nodo 3. Encontramos un camino más corto del nodo 1 al 2 con longitud 45.

| | 1 | 2 | 3 |
|---|---|---|---|
| 1 | 0 | 45 | 30 |
| 2 | 10 | 0 | 40 |
| 3 | 25 | 15 | 0 |

- Si queremos que además de la longitud de los caminos mínimos se devuelva la secuencia de nodos → añadimos una tabla adicional ruta:

```
tipo Tabla = matriz[1..N,1..N] de entero
fun Floyd(A: Tabla, N: entero, M: Tabla, ruta: Tabla)
    var
        i,j,k,tmp: entero
    fvar
    para i ← 0 hasta N hacer
        para j ← 0 hasta N hacer
            M[i,j] ← A[i,j]
            ruta[i,j] ← 0
        fpara
    fpara
    para k ← 1 hasta N hacer
        para i ← 1 hasta N hacer
            para j ← 1 hasta N hacer
                tmp ← M[i,k] + M[k,j]
                si tmp < M[i,j] entonces
                    M[i,j] ← tmp
                    ruta[i,j] ← k
                fsi
            fpara
        fpara
    fpara
ffun
```

```
fun VerRutas(N: entero, M: Tabla, ruta: Tabla)
    var
        i,j: entero
    fvar
    para i ← 0 hasta N hacer
        para j ← 0 hasta N hacer
            si M[i,j] ≠ ∞ entonces
                Imprimir('ruta de ',i,' a ',j)
                Imprimir(i)
                ImprimeRutaRec(ruta,i,j)
                Imprimir(j)
            fsi
        fpara
    fpara
ffun
```

```
fun ImprimeRutaRec(ruta: Tabla, i: entero, j: entero)
    var
        k: entero
    fvar
    k ← ruta[i,j]
    si k ≠ 0 entonces
        ImprimeRutaRec(ruta,i,k)
        Imprimir(k)
        ImprimeRutaRec(ruta,k,j)
    fsi
ffun
```

# 5.8 Distancia de edición

- Se resuelve un problema similar con Algoritmos de Ramificación y Poda, aunque en dicho caso hay costes distintos para cada una de las operaciones (borrar, insertar y sustituir)