

# Programación dinámica



Algoritmia básica - Javier Campos (Universidad de Zaragoza)



*Este documento está sujeto a una licencia de uso Creative Commons. No se permite un uso comercial de la obra original ni de las posibles obras derivadas, la distribución de las cuales se debe hacer con una licencia igual a la que regula la obra original.*

# Programación dinámica

---

• Introducción	3
• El problema de la mochila 0-1	8
• Camino de coste mínimo en un grafo multietapa	18
• Multiplicación de una secuencia de matrices	31
• Pasos en una solución de Programación Dinámica	41
• Comparaciones de secuencias	43
• Caminos mínimos entre todos los pares de nodos de un grafo	50
• Árboles binarios de búsqueda óptimos	56
• Un problema de fiabilidad de sistemas	67
• El problema del viajante de comercio	72
• Planificación de trabajos	82
• Una competición internacional	95
• Triangulación de polígonos	101



# Programación dinámica: Introducción

- Recordemos el problema de la mochila (fraccionaria):

- Se tienen  $n$  objetos fraccionables y una mochila.
- El objeto  $i$  tiene peso  $p_i$  y una fracción  $x_i$  ( $0 \leq x_i \leq 1$ ) del objeto  $i$  produce un beneficio  $b_i x_i$ .
- El objetivo es llenar la mochila, de capacidad  $C$ , de manera que se maximice el beneficio.

$$\text{maximizar } \sum_{1 \leq i \leq n} b_i x_i$$

$$\text{sujeto a } \sum_{1 \leq i \leq n} p_i x_i \leq C$$

$$\text{con } 0 \leq x_i \leq 1, b_i > 0, p_i > 0, 1 \leq i \leq n$$

- Una variante: la “mochila 0-1” (el “auténtico” problema de la mochila)
  - $x_i$  sólo toma valores 0 ó 1, indicando que el objeto se deja fuera o se mete en la mochila.
  - Los pesos,  $p_i$ , y la capacidad son números naturales. Los beneficios,  $b_i$ , son reales no negativos.



# Programación dinámica: Introducción

- Ejemplo:

$$n=3 \quad C=15$$

$$(b_1, b_2, b_3) = (38, 40, 24)$$

$$(p_1, p_2, p_3) = (9, 6, 5)$$

- Recordar la estrategia voraz:

- Tomar siempre el objeto que proporcione mayor beneficio por unidad de peso.
- Se obtiene esta solución:  $(x_1, x_2, x_3) = (0, 1, 1)$ , con beneficio 64
- Sin embargo, la solución óptima es:  $(x_1, x_2, x_3) = (1, 1, 0)$ , con beneficio 78

- Por tanto, la estrategia voraz no calcula la solución óptima del problema de la mochila 0-1.



# Programación dinámica: Introducción

R. Bellman: *Dynamic Programming*,  
Princeton University Press, 1957.

- Técnica de programación dinámica
  - Se emplea típicamente para resolver problemas de optimización.
  - Permite resolver problemas mediante secuencias de decisiones.
    - ↳ Como el esquema voraz
    - A diferencia del esquema voraz, se producen *varias secuencias de decisiones* y sólo al final se sabe cuál es la mejor de ellas.
  - Descompone un problema en subproblemas del mismo tipo.
    - ↳ Como en divide y vencerás
    - A diferencia de divide y vencerás, los subproblemas *están superpuestos entre sí*, comparten subproblemas entre ellos → se almacenan y reutilizan sus soluciones (*memoization*)
  - Está basada en el **principio de optimalidad de Bellman**:  
“Cualquier subsecuencia de decisiones de una secuencia óptima de decisiones que resuelve un problema también debe ser óptima respecto al subproblema que resuelve.”



# Programación dinámica: Introducción

---

- Supongamos que un problema se resuelve tras tomar una secuencia  $d_1, d_2, \dots, d_n$  de decisiones.
- Si hay  $d$  opciones posibles para cada una de las decisiones, una técnica de fuerza bruta exploraría un total de  $d^n$  secuencias posibles de decisiones (explosión combinatoria).
- La técnica de programación dinámica evita explorar todas las secuencias posibles por medio de la resolución de subproblemas de tamaño creciente y almacenamiento en una tabla de las soluciones óptimas de esos subproblemas para facilitar la solución de los problemas más grandes.



# Programación dinámica: Introducción

- Más formalmente:
  - Sea  $E_0$  el estado inicial del problema.
  - Sea  $D_1 = \{v_{1,1}, \dots, v_{1,m_1}\}$  el conjunto de valores de decisión posibles para la decisión  $d_1$ .
  - Sea  $E_{1_i}$  el estado del problema tras la elección del valor  $v_{1_i}$ ,  $1 \leq i \leq n_1$ .
  - Sea  $S_{1_i}$  una secuencia óptima de decisiones respecto al estado  $E_{1_i}$ .
    - Principio de optimalidad de Bellman:

Una secuencia óptima de decisiones respecto a  $E_0$  es la mejor de las secuencias de decisión  $\{v_{1_i}, S_{1_i}\}$ ,  $1 \leq i \leq n_1$ .

El mismo razonamiento puede aplicarse a cualquier subsecuencia de decisiones  $d_k, \dots, d_l$ ,  $1 \leq k \leq l \leq n$ , partiendo como estado inicial de  $E_{k-1}$ .

Una solución dinámica para este problema, simbolizado como  $(k, l)$ , debe expresarse en términos de los valores de decisión existentes para la decisión  $d_k$  y el subproblema  $(k+1, l)$ , resultante de aplicar cada valor de decisión.



# El problema de la mochila 0-1

- Sea  $mochila(k,l,P)$  el problema: maximizar  $\sum_{i=k}^l b_i x_i$   
sujeto a  $\sum_{i=k}^l p_i x_i \leq P$   
con  $x_i \in \{0, 1\}$ ,  $k \leq i \leq l$

Supongamos que  $p_i$ ,  $P$  y  $C$  son números naturales

- El problema de la mochila 0-1 es  $mochila(1,n,C)$ .
- Principio de optimalidad:  
Sea  $y_1, \dots, y_n$  una secuencia óptima de valores 0-1 para  $x_1, \dots, x_n$ .
  - Si  $y_1=0$ , entonces  $y_2, \dots, y_n$  forman una secuencia óptima para el problema  $mochila(2,n,C)$ .
  - Si  $y_1=1$ , entonces  $y_2, \dots, y_n$  forman una secuencia óptima para el problema  $mochila(2,n,C-p_1)$ .

Demostración: Si existe una solución mejor  $\tilde{y}_2, \dots, \tilde{y}_n$  para el problema correspondiente, entonces  $y_1, \tilde{y}_2, \dots, \tilde{y}_n$  es mejor que  $y_1, y_2, \dots, y_n$  para el problema  $mochila(1,n,C)$ , en contra de la hipótesis.





# El problema de la mochila 0-1

- Lo mismo se cumple en cualquier etapa de decisión:
  - Si  $y_1, \dots, y_n$  es una solución óptima del problema  $mochila(1, n, C)$ , entonces para todo  $j$ ,  $1 \leq j \leq n$ :
    - $y_1, \dots, y_j$  es solución óptima de

$$mochila\left(1, j, \sum_{i=1}^j p_i x_i\right)$$

- $y_{j+1}, \dots, y_n$  es solución óptima de

$$mochila\left(j+1, n, C - \sum_{i=1}^j p_i x_i\right)$$



# El problema de la mochila 0-1

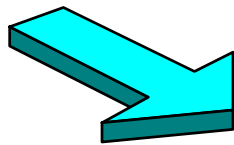
- Ecuación de recurrencia hacia adelante:
  - Si  $\vec{g}_j(c)$  es el beneficio (o ganancia total) de una solución óptima de *mochila*( $j,n,c$ ), entonces

$$\vec{g}_j(c) = \max\{\vec{g}_{j+1}(c), \vec{g}_{j+1}(c - p_j) + b_j\}$$

dependiendo de que el objeto  $j$ -ésimo entre o no en la solución (nótese que sólo puede entrar si  $c - p_j \geq 0$ ).

- Además,

$$\vec{g}_{n+1}(c) = 0, \text{ para cualquier capacidad } c$$



Ambas ecuaciones permiten calcular  $\vec{g}_1(C)$ , que es el valor de una solución óptima de *mochila*( $1,n,C$ ).

(Nótese que la ecuación de recurrencia es hacia adelante pero el cálculo se realiza hacia atrás.)

# El problema de la mochila 0-1

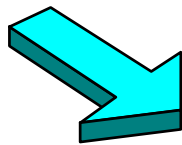
- Ecuación de recurrencia hacia atrás:
  - Si  $\overleftarrow{g}_j(c)$  es el beneficio (o ganancia total) de una solución óptima de *mochila*(1, $j$ , $c$ ), entonces

$$\overleftarrow{g}_j(c) = \max\{\overleftarrow{g}_{j-1}(c), \overleftarrow{g}_{j-1}(c - p_j) + b_j\}$$

dependiendo de que el objeto  $j$ -ésimo entre o no en la solución (nótese que sólo puede entrar si  $c - p_j \geq 0$ ).

- Además,

$$\overleftarrow{g}_0(c) = 0, \text{ para cualquier capacidad } c$$



Ambas ecuaciones permiten calcular  $\overleftarrow{g}_n(C)$ , que es el valor de una solución óptima de *mochila*(1, $n$ , $C$ ).

(Ahora la recurrencia es hacia atrás pero el cálculo se realiza hacia adelante.)

# El problema de la mochila 0-1

- Tanto la recurrencia hacia adelante como hacia atrás permiten escribir un algoritmo recursivo de forma inmediata.

```
función mochila1(p,b:vector[1..n] de nat; C:nat) devuelve nat
principio
  devuelve g(n,C)
fin
```

```
función g(j,c:nat) devuelve nat
principio
  si j=0 entonces devuelve 0
  sino
    si c<p[j] entonces devuelve g(j-1,c)
    sino
      si g(j-1,c)≥g(j-1,c-p[j])+b[j] entonces devuelve g(j-1,c)
      sino
        devuelve g(j-1,c-p[j])+b[j]
      fsi
    fsi
  fsi
fin
```



# El problema de la mochila 0-1

- Problema: ineficiencia

- Un problema de tamaño  $n$  se reduce a dos subproblemas de tamaño  $(n-1)$ .
- Cada uno de los dos subproblemas se reduce a otros dos...
- Por tanto, se obtiene un algoritmo **exponencial en  $n$** .

- Sin embargo, el número total de subproblemas a resolver no es tan grande:

La función  $\bar{g}_j(c)$  tiene dos parámetros:

- el primero puede tomar  $n$  valores distintos y
- el segundo,  $C$  valores.

→ ¡Luego sólo hay  $nC$  problemas diferentes!

- Por tanto, la solución recursiva está generando y resolviendo el mismo problema muchas veces.



# El problema de la mochila 0-1

- **Memoization**: las soluciones de los subproblemas se almacenan en una tabla para evitar hacer más de una vez la misma llamada recursiva → se obtiene un algoritmo iterativo.

- Matriz  $n \times C$  cuyo elemento  $(j,c)$  almacena  $\hat{g}_j(c)$

- Para el ejemplo anterior:

$$n=3 \quad C=15$$

$$(b_1, b_2, b_3) = (38, 40, 24)$$

$$(p_1, p_2, p_3) = (9, 6, 5)$$

*En castellano...*

*“memoización”*

*“memorialización”*

...

		C=...															
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$j=0$		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$j=1$	$p_1 = 9$	0	0	0	0	0	0	0	0	0	38	38	38	38	38	38	38
$j=2$	$p_2 = 6$	0	0	0	0	0	0	40	40	40	40	40	40	40	40	40	78
$j=3$	$p_3 = 5$	0	0	0	0	0	24	40	40	40	40	40	64	64	64	64	78

$$\hat{g}_j(c) = \max \left\{ \hat{g}_{j-1}(c), \hat{g}_{j-1}(c - p_j) + b_j \right\}$$



# El problema de la mochila 0-1

```
algoritmo mochila(ent p,b:vect[1..n]de nat; ent Cap:nat;
                  sal g:vect[0..n,0..Cap]de nat)
variables c,j:nat
principio
  para c:=0 hasta Cap hacer g[0,c]:=0 fpara;
  para j:=1 hasta n hacer g[j,0]:=0 fpara;
  para j:=1 hasta n hacer
    para c:=1 hasta Cap hacer
      si c<p[j] entonces
        g[j,c]:=g[j-1,c]
      sino
        si g[j-1,c]≥g[j-1,c-p[j]]+b[j] entonces
          g[j,c]:=g[j-1,c]
        sino
          g[j,c]:=g[j-1,c-p[j]]+b[j]
      fsi
    fsi
  fpara
fpara
fin
```



# El problema de la mochila 0-1

- Cálculos posibles a partir de la tabla  $g$ :
  - beneficio total:  $g[n, \text{Cap}]$
  - los objetos metidos en la mochila:

```
algoritmo objetos(ent p,b: vect[1..n] de nat; ent Cap:nat;  
                 ent g: vect[0..n,0..Cap] de nat)  
principio  
  test(n, Cap)  
fin
```

```
algoritmo test(ent j,c:nat)  
principio  
  si j>0 entonces  
    si c<p[j] entonces test(j-1,c)  
    sino  
      si g[j-1,c-p[j]]+b[j]>g[j-1,c] entonces  
        test(j-1,c-p[j]); escribir('meter ',j)  
      sino test(j-1,c)  
  fsi fsi fsi  
fin
```





# El problema de la mochila 0-1

---

- Consideraciones finales

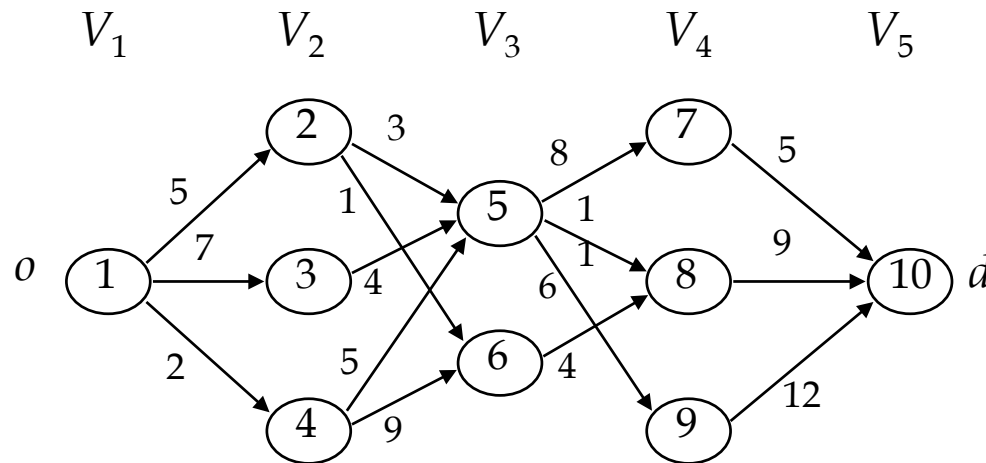
- Cada componente de la tabla  $g$  se calcula en tiempo constante, luego el coste de construcción de la tabla es  $O(nC)$ .  
→ coste *pseudo-polinómico* (= exponencial)  
( $C$  crece exponencialmente en su número de dígitos/bits)
- El algoritmo `test` se ejecuta una vez por cada valor de  $j$ , desde  $n$  descendiendo hasta  $0$ , luego su coste es  $O(n)$ .
- Si  $C$  es muy grande, entonces esta solución no es buena.
- Si los pesos  $p_i$  o la capacidad  $C$  son reales, esta solución no sirve.



# Camino de coste mínimo en un grafo multietapa

- Grafo multietapa:

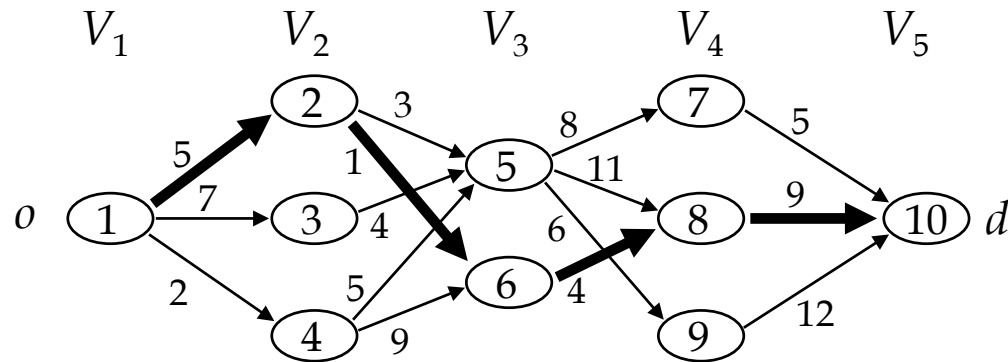
- Un grafo multietapa  $G=(V,A)$  es un grafo dirigido en el que se puede hacer una partición del conjunto  $V$  de vértices en  $k$  ( $k \geq 2$ ) conjuntos distintos  $V_i$ ,  $1 \leq i \leq k$ , tal que todo arco del grafo  $(u,v)$  es tal que  $u \in V_i$  y  $v \in V_{i+1}$  para algún  $i$ ,  $1 \leq i < k$ .
- Los conjuntos  $V_1$  y  $V_k$  tienen un solo vértice que se llama vértice origen,  $o$ , y vértice destino,  $d$ , respectivamente.



- Consideraremos grafos etiquetados.  
Denotamos por  $c(u,v)$  el coste del arco  $(u,v)$ .

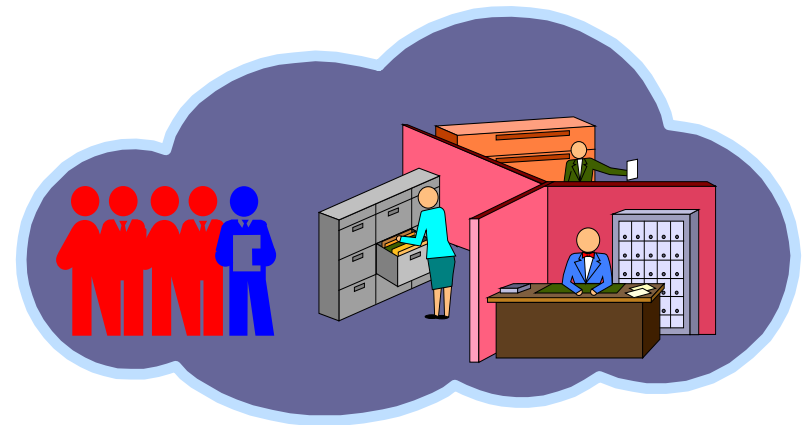
# Camino de coste mínimo en un grafo multietapa

- El problema: Encontrar un camino de coste mínimo que vaya de  $o$  a  $d$ .
  - Todo camino de  $o$  a  $d$  tiene exactamente un vértice en cada  $V_i$ , por eso se dice que cada  $V_i$  define una etapa del grafo.



# Camino de coste mínimo en un grafo multietapa

- Ejemplo de aplicación:
  - Se tienen  $n$  unidades de un recurso que deben asignarse a  $r$  proyectos.
  - Si se asignan  $j$ ,  $0 \leq j \leq n$ , unidades al proyecto  $i$  se obtiene un beneficio  $N_{i,j}$ .
  - El problema es asignar el recurso a los  $r$  proyectos maximizando el beneficio total.



# Camino de coste mínimo en un grafo multietapa

## – Formulación como grafo multietapa:

- Número de etapas:  $r+1$
- La etapa  $i$ ,  $1 \leq i \leq r$ , representa el proyecto  $i$ .
- Hay  $n+1$  vértices  $v_{i,j}$ ,  $0 \leq j \leq n$ , en cada etapa  $i$ ,  $2 \leq i \leq r$ .
- Las etapas 1 y  $r+1$  tienen un vértice,  $o=v_{1,0}$  y  $d=v_{r+1,n}$ , respectivamente.
- El vértice  $v_{i,j}$ ,  $2 \leq i \leq r$ , representa el estado en el que se asignan un total de  $j$  unidades del recurso a los proyectos 1, 2, ...,  $i-1$ .
- Los arcos son de la forma  $(v_{i,j}, v_{i+1,l})$  para todo  $j \leq l$  y  $1 \leq i < r$ .
- El arco  $(v_{i,j}, v_{i+1,l})$ ,  $j \leq l$ , tiene asignado un coste  $N_{i,l-j}$  que corresponde a asignar  $l-j$  unidades del recurso al proyecto  $i$ ,  $1 \leq i < r$ .
- Además hay arcos de la forma  $(v_{r,j}, v_{r+1,n})$ , que tienen asignado un coste

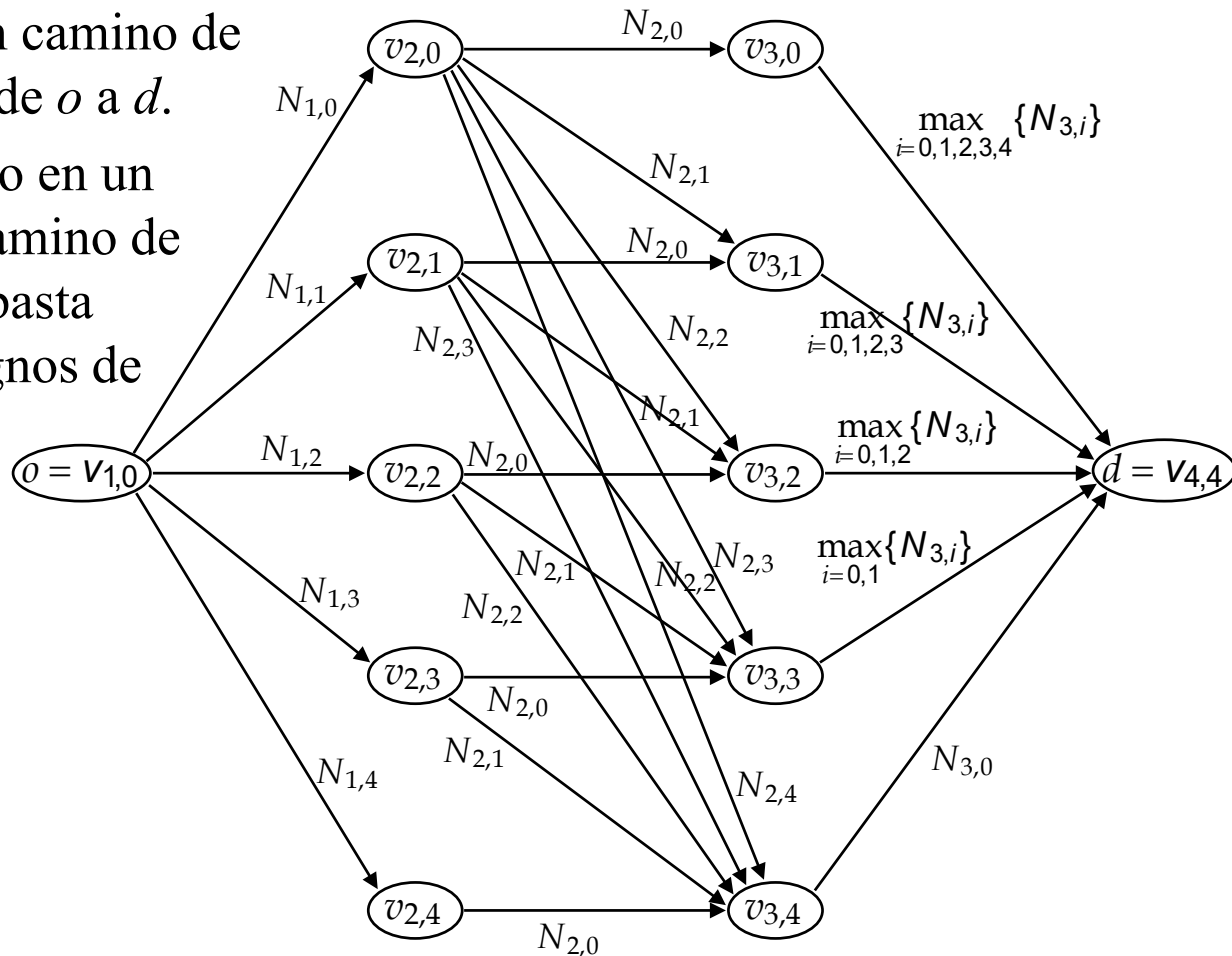
$$\max_{0 \leq p \leq n-j} \{N_{r,p}\}.$$



# Camino de coste mínimo en un grafo multietapa

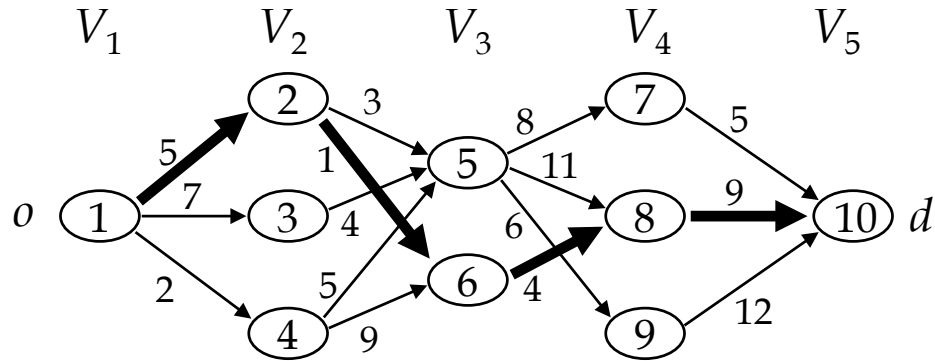
– Grafo resultante para  $r=3$  y  $n=4$ .

- La asignación óptima está definida por un camino de coste máximo de  $o$  a  $d$ .
- Para convertirlo en un problema de camino de coste mínimo basta cambiar los signos de las etiquetas.



# Camino de coste mínimo en un grafo multietapa

- Solución de programación dinámica:



- Cada camino de  $o$  a  $d$  es el resultado de una secuencia de  $k-2$  decisiones.
- Decisión  $i$ -ésima: determinar, a partir de un vértice  $v_i$  de  $V_i$ , un arco que tenga a  $v_i$  como origen y algún nodo de  $V_{i+1}$  como destino.
- Principio de optimalidad:

El camino de coste mínimo debe contener subcaminos de coste mínimo entre otros nodos.

Dem.: En otro caso, podrían sustituirse dichos subcaminos por otros mejores, resultando un camino total de coste menor.

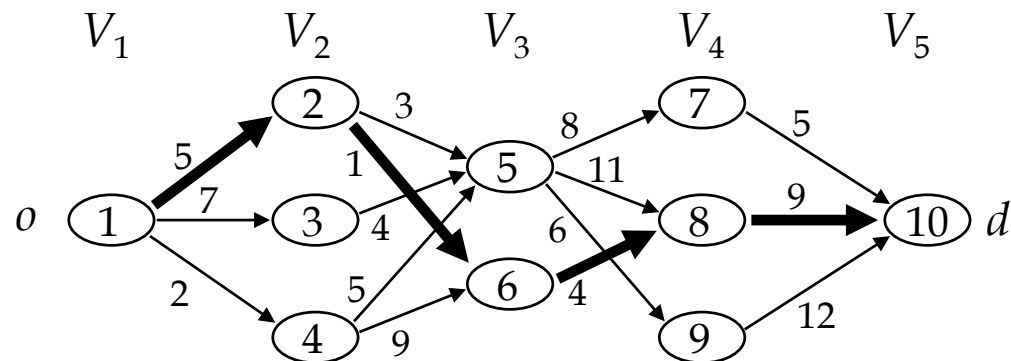
# Camino de coste mínimo en un grafo multietapa

– Ecuación de recurrencia hacia adelante:

- Sea  $s(i,j)$  un camino de coste mínimo  $C^*(i,j)$  desde el vértice  $j$  del conjunto  $V_i$  hasta el vértice destino  $d$ .
- Entonces:

$$C^*(k-1, j) = \begin{cases} c(j, d), & \text{si } (j, d) \in A \\ \infty, & \text{en otro caso} \end{cases}$$

$$C^*(i, j) = \min_{\substack{l \in V_{i+1} \\ (j,l) \in A}} \{c(j, l) + C^*(i+1, l)\}, \text{ para } 1 \leq i \leq k-2$$

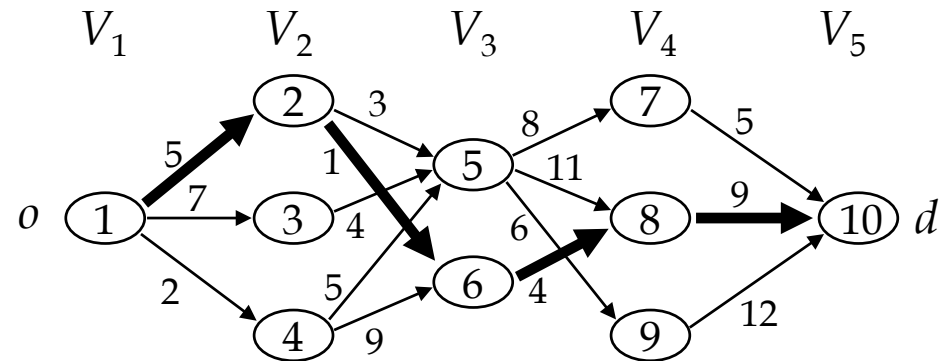




# Camino de coste mínimo en un grafo multietapa

$$C^*(k-1, j) = \begin{cases} c(j, d), & \text{si } (j, d) \in A \\ \infty, & \text{en otro caso} \end{cases}$$

$$C^*(i, j) = \min_{\substack{l \in V_{i+1} \\ (j, l) \in A}} \{c(j, l) + C^*(i+1, l)\}, \text{ para } 1 \leq i \leq k-2$$



$$C^*(3, 5) = \min \{8 + C^*(4, 7), 11 + C^*(4, 8), 6 + C^*(4, 9)\} = 13$$

$$C^*(3, 6) = 4 + C^*(4, 8) = 13$$

$$C^*(2, 2) = \min \{3 + C^*(3, 5), 1 + C^*(3, 6)\} = 14$$

$$C^*(2, 3) = 4 + C^*(3, 5) = 17$$

$$C^*(2, 4) = \min \{5 + C^*(3, 5), 9 + C^*(3, 6)\} = 18$$

$$C^*(1, 1) = \min \{5 + C^*(2, 2), 7 + C^*(2, 3), 2 + C^*(2, 4)\} = 19$$



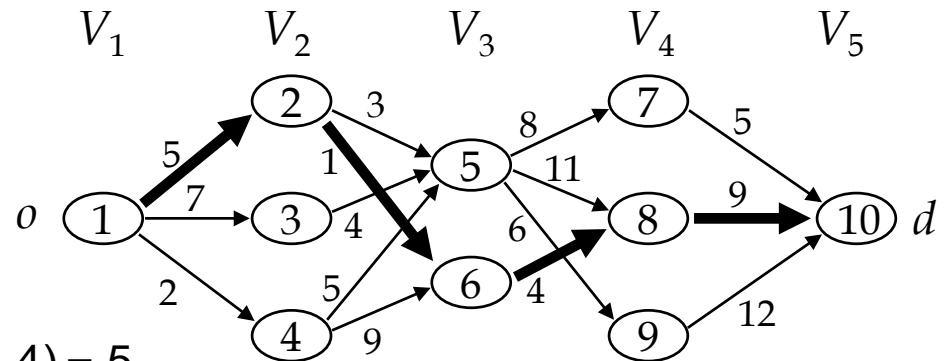
# Camino de coste mínimo en un grafo multietapa

– Falta almacenar las decisiones hechas en cada etapa que minimizan el coste:

- Sea  $D(i,j)$  el valor de  $l$  que minimiza  $c(j,l) + C^*(i+1,l)$ .

- Entonces el camino de coste mínimo es:

$$v_1=1; v_2=D(1,1); v_3=D(2,D(1,1)); \text{ etc.}$$



$$\begin{aligned} D(3,5) &= 7; & D(3,6) &= 8 \\ D(2,2) &= 6; & D(2,3) &= 5; & D(2,4) &= 5 \\ D(1,1) &= 2 \end{aligned}$$

$$\begin{aligned} v_1 &= 1 \\ v_2 &= D(1,1) = 2 \\ v_3 &= D(2, D(1,1)) = 6 \\ v_4 &= D(3, D(2, D(1,1))) = 8 \end{aligned}$$



# Camino de coste mínimo en un grafo multietapa

```
algoritmo multietapa(ent G=(V,A,c):grafo; ent k,n:nat;  
                    sal P:vect[1..k]de 1..n)  
{Los vértices están numerados de forma que los índices de los  
vértices de una etapa son mayores que los índices de los de la  
etapa anterior.  
El primer índice de C* y D, que sólo identificaba la etapa, se ha  
suprimido.}  
variables C:vect[1..n]de real; D:vect[1..n]de 1..n; j,r:1..n  
principio  
  C[n]:=0.0; {Cálculo de C* y D}  
  para j:=n-1 descendiendo hasta 1 hacer  
    r:=vértice t.q. (j,r)∈A ∧ c(j,r)+C[r] es mínimo;  
    C[j]:=c(j,r)+C[r];  
    D[j]:=r  
  fpara;  
  P[1]:=1; P[k]:=n; {Construcción del camino}  
  para j:=2 hasta k-1 hacer  
    P[j]:=D[P[j-1]]  
  fpara  
fin
```



# Camino de coste mínimo en un grafo multietapa

---

- Coste del algoritmo:
  - Si  $G$  está representado mediante listas de adyacencia, entonces el cálculo de  $r$  en el interior del primer bucle lleva un tiempo proporcional al grado del vértice  $j$ .
  - Por tanto, si  $a$  es el número de arcos del grafo, el coste total del algoritmo es  $\Theta(a)$ .

(El segundo bucle lleva un tiempo  $\Theta(k)$ .)



# Camino de coste mínimo en un grafo multietapa

- Análogamente, se desarrolla la recurrencia hacia atrás.

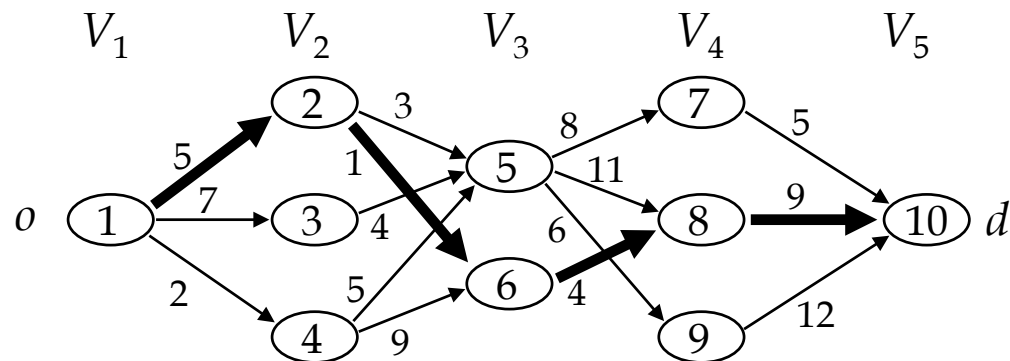
– Ecuación de recurrencia hacia atrás:

- Sea  $s(i,j)$  un camino de coste mínimo  $C^*(i,j)$  desde el vértice origen  $o$  hasta el vértice  $j$  del conjunto  $V_i$ .

- Entonces:

$$C^*(2, j) = \begin{cases} c(o, j), & \text{si } (o, j) \in A \\ \infty, & \text{en otro caso} \end{cases}$$

$$C^*(i, j) = \min_{\substack{l \in V_{i-1} \\ (l, j) \in A}} \{c(l, j) + C^*(i-1, l)\}, \text{ para } 3 \leq i \leq k$$



# Camino de coste mínimo en un grafo multietapa

```
algoritmo multietapaB(ent G=(V,A,c):grafo; ent k,n:nat;  
                    sal P:vect[1..k]de 1..n)  
{Los vértices están numerados de forma que los índices de los  
vértices de una etapa son mayores que los índices de los de la etapa  
anterior. El primer índice de C* y D, que sólo identificaba la  
etapa, se ha suprimido.}  
variables C:vect[1..n]de real; D:vect[1..n]de 1..n; j,r:1..n  
principio  
  C[1]:=0.0; {Cálculo de C* y D}  
  para j:=2 hasta n hacer  
    r:=vértice t.q. (r,j)∈A ∧ c(r,j)+C[r] es mínimo;  
    C[j]:=c(r,j)+C[r];  
    D[j]:=r  
  fpara;  
  P[1]:=1; P[k]:=n; {Construcción del camino}  
  para j:=k-1 descendiendo hasta 2 hacer  
    P[j]:=D[P[j+1]]  
  fpara  
fin
```

**Nota:** La eficiencia es la misma si  $G$  está representado mediante listas de adyacencia inversa.



# Multiplicación de una secuencia de matrices

- Se desea calcular el producto matricial:  $M = M_1 M_2 \dots M_n$

Como es asociativo, existen varias formas...

(Recordar que el algoritmo resultante de la definición del producto de dos matrices  $p \times q$  y  $q \times r$  necesita  $pqr$  multiplicaciones de escalares.)

- Ejemplo: se quiere calcular el producto  $ABCD$ , de las matrices  $A(13 \times 5)$ ,  $B(5 \times 89)$ ,  $C(89 \times 3)$  y  $D(3 \times 34)$ .

	nº multip.
$((AB)C)D$	10582
$(AB)(CD)$	54201
$(A(BC))D$	2856
$A((BC)D)$	4055
$A(B(CD))$	26418

¡El caso más eficiente es casi 19 veces más rápido que el más lento!



# Multiplicación de una secuencia de matrices

- ¿Cómo hallar el mejor método?
  1. Insertar los paréntesis de todas las formas posibles (significativamente diferentes).
  2. Calcular para cada una el número de multiplicaciones escalares requeridas.
- ¿Cuántas formas posibles  $T(n)$  de insertar paréntesis existen en un producto de  $n$  matrices?
  - Si cortamos entre la  $i$  y la  $(i+1)$ -ésima:  $M = (M_1 M_2 \dots M_i)(M_{i+1} M_{i+2} \dots M_n)$   
Entonces tenemos  $T(i)T(n-i)$  formas distintas.
  - Como  $i$  puede tomar valores entre 1 y  $n-1$ :

$$T(n) = \sum_{i=1}^{n-1} T(i)T(n-i), \text{ para } n > 1$$
$$T(1) = 1$$

Números de Catalan





# Multiplicación de una secuencia de matrices

---

- Los números de Catalan crecen exponencialmente.
  - De hecho puede demostrarse que:

$$T(n) = \frac{1}{n} \binom{2n-2}{n-1}$$

Por ejemplo:

$n$	1	2	3	4	5	10	15
$T(n)$	1	1	2	5	14	4862	2674440

- Luego el método directo “no sirve” (por costoso).



# Multiplicación de una secuencia de matrices

S. Godbole: “On efficient computation of matrix chain products”, *IEEE Transactions on Computers*, 22(9), pp. 864-866, 1973.

- Aplicación del principio de optimalidad:

Si el mejor modo de realizar el producto exige dividir inicialmente entre las matrices  $i$  e  $(i+1)$ -ésima, los productos

$$M_1M_2\cdots M_i \text{ y } M_{i+1}M_{i+2}\cdots M_n$$

deberán ser realizados de forma óptima para que el total también sea óptimo.

- Método:

- Construir la matriz  $[m_{ij}]$ ,  $1 \leq i \leq j \leq n$ , donde  $m_{ij}$  da el óptimo (i.e., el número de multiplicaciones escalares requeridas) para la parte  $M_iM_{i+1}\cdots M_j$  del producto total.
- La solución final vendrá dada por  $m_{1n}$ .



# Multiplicación de una secuencia de matrices

## – Construcción de $[m_{ij}]$ , $1 \leq i \leq j \leq n$ :

- Guardar las dimensiones de las  $M_i$ ,  $1 \leq i \leq n$ , en un vector  $d$ , de  $0..n$  componentes, de forma que  $M_i$  tiene dimensiones  $d_{i-1} \times d_i$ .
- La diagonal  $s$  de  $[m_{ij}]$  contiene los  $m_{ij}$  tales que  $j-i=s$ :

$$s = 0: \quad m_{j,j} = 0, \text{ para } i = 1, 2, \dots, n$$

$$s = 1: \quad m_{j,j+1} = d_{j-1}d_jd_{j+1}, \text{ para } i = 1, 2, \dots, n-1$$

$$1 < s < n: \quad m_{j,j+s} = \min_{i \leq k \leq i+s-1} (m_{ik} + m_{k+1,j+s} + d_{i-1}d_kd_{j+s}),$$

$$\text{para } i = 1, 2, \dots, n-s$$

- El tercer caso representa que para calcular  $M_i M_{i+1} \dots M_{i+s}$  se intentan todas las posibilidades  $(M_i M_{i+1} \dots M_k)(M_{k+1} M_{k+2} \dots M_{i+s})$  y se escoge la mejor.
- De forma más compacta:

$$m_{ij} = \begin{cases} 0, & \text{si } i = j \\ \min_{i \leq k < j} \{m_{ik} + m_{k+1,j} + d_{i-1}d_kd_j\}, & \text{si } i < j \end{cases}$$



# Multiplicación de una secuencia de matrices

- Para el ejemplo anterior:

- $A(13 \times 5)$ ,  $B(5 \times 89)$ ,  $C(89 \times 3)$  y  $D(3 \times 34)$ . Se tiene  $d=(13,5,89,3,34)$ .

- Para  $s=1$ :  $m_{12}=5785$ ,  $m_{23}=1335$ ,  $m_{34}=9078$ .

- Para  $s=2$ :
 
$$m_{13} = \min (m_{11} + m_{23} + 13 \times 5 \times 3, m_{12} + m_{33} + 13 \times 89 \times 3)$$

$$= \min (1530, 9256) = 1530$$

$$m_{24} = \min (m_{22} + m_{34} + 5 \times 89 \times 34, m_{23} + m_{44} + 5 \times 3 \times 34)$$

$$= \min (24208, 1845) = 1845$$

- Para  $s=3$ :
 
$$m_{14} = \min \left( \begin{array}{l} \{k=1\} \quad m_{11} + m_{24} + 13 \times 5 \times 34, \\ \{k=2\} \quad m_{12} + m_{34} + 13 \times 89 \times 34, \\ \{k=3\} \quad m_{13} + m_{44} + 13 \times 3 \times 34 \end{array} \right)$$

$$= \min (4055, 54201, 2856) = 2856$$

- La matriz es:

	$j = 1$	$2$	$3$	$4$	
$i = 1$	0	5785	1530	2856	$s = 3$
$2$		0	1335	1845	$s = 2$
$3$			0	9078	$s = 1$
$4$				0	$s = 0$

$(s = j - i,$   
es decir,  
 $j = i + s)$



# Multiplicación de una secuencia de matrices

---

- Solución recursiva inmediata:
  - Aplicación de la ecuación recurrente.
  - Problema: complejidad exponencial.
- Almacenamiento de las soluciones de los subproblemas en una tabla:
  - Número de subproblemas:  $\Theta(n^2)$ .



# Multiplicación de una secuencia de matrices

```
algoritmo parentOpt(ent d:vect[0..n]de nat;  
    sal m:vect[1..n,1..n]de nat; sal km:vect[1..n,1..n]de 1..n)  
{m es la matriz  $[m_{i,j}]$  definida antes; km[i,j] guarda el índice k para  
  el que se alcanza el mínimo al calcular m[i,j].}  
variables i,s,j,k,q:nat  
principio  
  para i:=1 hasta n hacer m[i,i]:=0 fpara;  
  para s:=1 hasta n-1 hacer  
    para i:=1 hasta n-s hacer  
      j:=i+s;  
      m[i,j]:=∞;  
      para k:=i hasta j-1 hacer  
        q:=m[i,k]+m[k+1,j]+d[i-1]*d[k]*d[j];  
        si q<m[i,j] entonces  
          m[i,j]:=q; km[i,j]:=k  
        fsi  
      fpara  
    fpara  
  fpara  
fin
```



# Multiplicación de una secuencia de matrices

---

- Coste en tiempo:

- $\Theta(n^3)$

- Coste en memoria:

- $\Theta(n^2)$



# Multiplicación de una secuencia de matrices

- ¡Falta hacer el producto!
  - El elemento  $km[i,j]$  guarda el valor de  $k$  tal que la división óptima de  $M_i M_{i+1} \cdots M_j$  parte el producto entre  $M_k$  y  $M_{k+1}$ .
  - Por tanto:

```
función multSec (M: vect [1..n] de matriz;  
                km: vect [1..n, 1..n] de 1..n;  
                i, j: 1..n) devuelve matriz  
variables X, Y: matriz  
principio  
  si j > i entonces  
    X := multSec (M, km, i, km[i, j]);  
    Y := multSec (M, km, km[i, j] + 1, j);  
    devuelve mult (X, Y)  
  sino  
    devuelve M[i]  
  fsi  
Fin
```





# Pasos en una solución de Programación Dinámica

Para plantear una solución de programación dinámica deben seguirse los pasos siguientes:

1. Definir de manera precisa una **función parametrizada** tal que, para algunos valores de sus parámetros, nos aporte la solución del problema propuesto.
2. Escribir una **ecuación recurrente** (o relación de recurrencia) para esa función parametrizada, detallando para qué valores de sus parámetros es válida la ecuación, y escribir los **casos base** de la función (casos particulares de sus parámetros para los que se puede expresar el valor de la función sin necesidad de recurrencias).
3. Detallar qué **tabla es necesaria para almacenar los valores** de la función parametrizada, y qué **orden puede seguirse para calcularla**. Detallar si se precisa alguna otra **tabla auxiliar** para reconstruir la solución para la que la función alcanza el óptimo.
4. Escribir en pseudocódigo el **algoritmo para rellenar la tabla** de los valores óptimos de la función parametrizada y, si se solicita, escribir el **algoritmo para construir la solución** del problema original.



# Pasos en una solución de Programación Dinámica

Ejemplo para el problema de multiplicación de una secuencia de matrices:

1. Función  $m(i,j)$  = número mínimo de multiplicaciones de escalares necesarias para multiplicar la subsecuencia de matrices  $M_i, \dots, M_j$ .  
El problema originalmente propuesto es calcular  $m(1,n)$ .
2.  $m(i,i) = 0$ , para  $i = 1..n$  (casos base).  
 $m(i,j) = \min_{i \leq k < j} \{m(i,k) + m(k+1,j) + d_{i-1}d_kd_j\}$ , para los  $i, j$  tales que  $1 \leq i < j \leq n$ .
3. Matriz triangular superior para almacenar los  $m(i,j)$ , con  $1 \leq i \leq j \leq n$ .  
Puede calcularse por diagonales, empezando por la diagonal principal y, desde ahí, hacia arriba.  
Matriz auxiliar de iguales dimensiones para almacenar los índices  $k$  para los que se consigue el mínimo de la ecuación recurrente para cada uno de los  $m(i,j)$ . Es la matriz  $km[i,j]$  en el algoritmo de la transparencia nº 38.
4. Algoritmos de la página 38 y de la página 40.



# Comparaciones de secuencias

D. Gusfield: *Algorithms on Strings, Trees, and Sequences. Computer Science and Computational Biology*, Cambridge University Press, 1997.

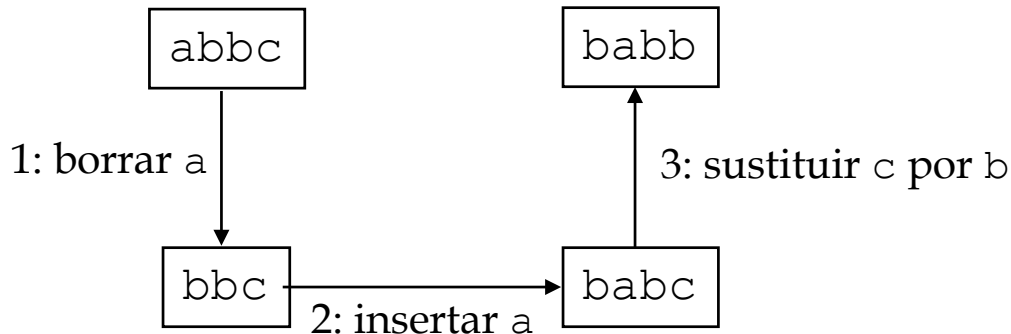
- Problemas relacionados con biología molecular.
- Problema de la *distancia de edición*:  
“Mínimo número de pasos de edición para transformar una cadena en otra.”
  - Sean  $A=a_1a_2\dots a_n$  y  $B=b_1b_2\dots b_m$  dos cadenas de caracteres.
  - Se quiere modificar  $A$ , carácter a carácter, hasta convertirlo en  $B$ .
  - Se permiten tres tipos de cambios (pasos de edición) y cada uno tiene coste unidad:
    1. **Insertar** un carácter en la cadena.
    2. **Borrar** un carácter de la cadena.
    3. **Sustituir** un carácter por otro diferente.



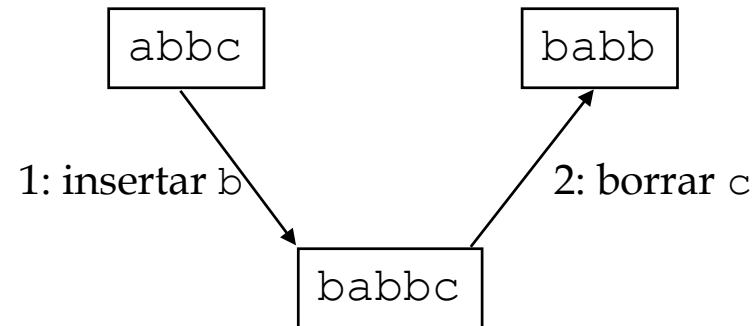
# Comparaciones de secuencias

– Ejemplo: 

Una solución:



Otra solución (mejor):



- Aplicaciones en comparación y mantenimiento de versiones de ficheros:
  - Si se tienen varias versiones parecidas de un fichero es más eficiente almacenar sólo la 1ª versión y para el resto de versiones almacenar los pasos de edición (normalmente inserciones y borrados) desde la 1ª.

# Comparaciones de secuencias

R.A. Wagner y M.J. Fischer:

“The string-to-string correction problem”,  
*Journal of the ACM*, 21, pp. 168-173, 1974.

- Solución de programación dinámica:
  - Sean  $A(i)$  y  $B(j)$  las subcadenas prefijo de  $A$  y  $B$  (i.e.,  $A(i)=a_1a_2\dots a_i$ ,  $B(j)=b_1b_2\dots b_j$ ).
  - $C(i,j)$  el coste mínimo de transformar  $A(i)$  en  $B(j)$ .
  - El problema original es:  $A(n) \rightarrow B(m)$
  - Fijémonos en los posibles tratamientos de  $a_i$  en el problema  $A(i) \rightarrow B(j)$ :
    - bien es borrado, y el problema se reduce a transformar  $A(i-1)$  en  $B(j)$  y luego borrarlo;
    - bien se le hace coincidir con algún carácter de  $B$  anterior a  $b_j$ , en cuyo caso el problema se reduce a transformar  $A(i)$  en  $B(j-1)$  y luego insertar un carácter igual a  $b_j$ ;
    - bien se le sustituye por un carácter igual a  $b_j$ , y el problema se reduce a transformar  $A(i-1)$  en  $B(j-1)$  y luego sustituir  $a_i$ ;
    - o bien  $a_i$  coincide con  $b_j$ , y entonces basta con transformar  $A(i-1)$  en  $B(j-1)$ .



# Comparaciones de secuencias

- Si denotamos:

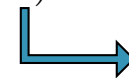
$$c(i, j) = \begin{cases} 0, & \text{si } a_i = b_j \\ 1, & \text{si } a_i \neq b_j \end{cases}$$

- Se tiene:

$$C(i, j) = \min \begin{cases} C(i-1, j) + 1 & \text{(borrando } a_i) \\ C(i, j-1) + 1 & \text{(insertando } b_j) \\ C(i-1, j-1) + c(i, j) & \text{(otros casos)} \end{cases}$$

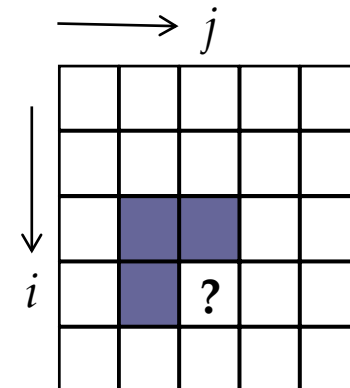
$$C(i, 0) = i, \text{ for all } i, 0 \leq i \leq n$$

$$C(0, j) = j, \text{ for all } j, 0 \leq j \leq m$$



- No hacer nada
- Cambiar  $a_i$  por  $b_j$

- Solución recursiva obvia: ¡coste exponencial!
- Sin embargo, sólo existen  $nm$  subproblemas diferentes ( $C(i, j)$ ,  $1 \leq i \leq n, 1 \leq j \leq m$ ).
- Orden de cálculo (a partir de la ecuación):



# Comparaciones de secuencias

```
algoritmo compSec(ent A:cad[n]; ent B:cad[m];  
    sal C:vect[0..n,0..m]de nat; sal T:vect[1..n,1..m]de Transf)  
{Transf=(borrar,insert,sustit,nada); i.e. posibles transformaciones}  
variables i,j,x,y,z:nat  
principio  
    para i:=0 hasta n hacer C[i,0]:=i fpara;  
    para j:=0 hasta m hacer C[0,j]:=j fpara;  
    para i:=1 hasta n hacer  
        para j:=1 hasta m hacer  
            x:=C[i-1,j]+1; y:=C[i,j-1]+1;  
            si A[i]=B[j] entonces z:=C[i-1,j-1]  
            sino z:=C[i-1,j-1]+1  
            fsi;  
            C[i,j]:=min(x,y,z);  
            T[i,j]:= (min=x) => borrar;    {último cambio de A[i] a B[j]}  
                    (min=y) => insert;  
                    (min=z) & A[i]=B[j]) => nada;  
                    (min=z) & A[i]≠B[j]) => sustit;  
        fpara  
    fpara  
Fin
```



# Comparaciones de secuencias

```
algoritmo res(ent i,j:nat)
{Para resolver el problema, ejecutar res(n,m).}
variable k:nat
principio
  selección
    i=0: para k:=1 hasta j hacer
      escribir('Añadir',B[k],'en el lugar',k) fpara
    j=0: para k:=1 hasta i hacer
      escribir('Borrar car.nº',k) fpara
  otros:
    selección
      T[i,j]=borrar: res(i-1,j); escribir('Borrar car.nº',i)
      T[i,j]=insert: res(i,j-1); escribir('Insertar car.nº',j,
        'de B tras la posición',i)
      T[i,j]=sustit: res(i-1,j-1); escribir('Sustit. car.nº',i,
        'de A por nº',j,'de B')
      T[i,j]=nada: res(i-1,j-1)
    fselección
  fselección
fin
```





# Comparaciones de secuencias

---

- Coste:
  - En tiempo:  $\Theta(nm)$
  - En espacio:  $\Theta(nm)$



# Caminos mínimos entre todos los pares de nodos de un grafo

R.W. Floyd:

“Algorithm 97: Shortest path”,  
*Communications of the ACM*, 5(6), p. 345, 1962.

Bernard Roy (1959)

Robert Floyd (1962)

Stephen Warshall (1962)

- Problema:

Cálculo de los caminos de coste mínimo entre todos los pares de vértices de un grafo dirigido sin ciclos de peso negativo.

- Principio de optimalidad:

Si  $i_1, i_2, \dots, i_k, i_{k+1}, \dots, i_n$  es un camino de coste mínimo de  $i_1$  a  $i_n$ , entonces:

- $i_1, i_2, \dots, i_k$  es un camino de coste mínimo de  $i_1$  a  $i_k$ , y
- $i_k, i_{k+1}, \dots, i_n$  es un camino de coste mínimo de  $i_k$  a  $i_n$ .

- Aplicación del principio:

- Si  $k$  es el vértice intermedio de mayor índice en el camino óptimo de  $i$  a  $j$ , entonces el subcamino de  $i$  a  $k$  es un camino óptimo de  $i$  a  $k$  que, además, sólo pasa por vértices de índice menor que  $k$ .
- Lo análogo ocurre con el subcamino de  $k$  a  $j$ .



# Caminos mínimos entre todos los pares de nodos de un grafo

- Sea  $C(i,j)$  el peso de la arista  $(i,j)$  o infinito si esa arista no existe.  
Sea  $C(i,i)=0$ .
- Sea  $D_k(i,j)$  la longitud (o distancia) del camino de coste mínimo de  $i$  a  $j$  que no pasa por ningún vértice de índice mayor que  $k$ .
- Sea  $D(i,j)$  la longitud del camino de coste mínimo de  $i$  a  $j$ .
- Entonces:

$$D(i,j) = D_n(i,j), \quad 1 \leq i \leq n, 1 \leq j \leq n$$

- Ahora, un camino óptimo de  $i$  a  $j$  que no pase por ningún vértice de índice mayor que  $k$ , o bien pasa por el vértice  $k$  o no pasa.
  - Si pasa por  $k$  entonces:  $D_k(i, j) = D_{k-1}(i, k) + D_{k-1}(k, j)$
  - Si no pasa por  $k$  entonces ningún vértice intermedio tiene índice superior a  $k-1$ :  $D_k(i, j) = D_{k-1}(i, j)$



# Caminos mínimos entre todos los pares de nodos de un grafo

---

- En resumen:
  - Se tiene la siguiente ecuación recurrente que define el método de programación dinámica.

$$D_k(i, j) = \min \left\{ D_{k-1}(i, j), D_{k-1}(i, k) + D_{k-1}(k, j) \right\}$$

$k \geq 1$

$$D_0(i, j) = C(i, j), \quad 1 \leq i \leq n, \quad 1 \leq j \leq n$$



# Caminos mínimos entre todos los pares de nodos de un grafo

```
{Pre: g es un grafo dirigido etiquetado sin ciclos negativos}
```

```
función Floyd(g:grafo) devuelve vector[vért,vért] de etiq
```

```
variables D:vector[vért,vért] de etiq;
```

```
    i,j,k:vért
```

```
principio
```

```
{inicialmente la distancia entre dos vértices tiene el valor de la  
  arista que los une; las diagonales se ponen a cero}
```

```
para todo i en vért hacer
```

```
    para todo j en vért hacer
```

```
        D[i,j]:=etiqueta(g,i,j)  {∞ si no hay arco}
```

```
    fpara;
```

```
    D[i,i]:=0
```

```
fpara;
```

```
...
```



# Caminos mínimos entre todos los pares de nodos de un grafo

```
...
para todo k en vért hacer
  para todo i en vért hacer
    para todo j en vért hacer
      si  $D[i,k]+D[k,j]<D[i,j]$  entonces
         $D[i,j]:=D[i,k]+D[k,j]$ 
      fsi
    fpara
  fpara
fpara;
devuelve D
fin
{Post:  $D=\text{caminosMínimos}(g)$ }
```

Ver en la web (material adicional) por qué se puede eliminar el subíndice  $k$  de la ecuación en recurrencias y por tanto evitar el uso de otra matriz.



# Caminos mínimos entre todos los pares de nodos de un grafo

---

- Eficiencia temporal:  $\Theta(n^3)$ 
  - representación con matriz de adyacencia:  
igual que reiterar Dijkstra (*Algoritmos voraces*, pág. 28), aunque el interior del bucle en Floyd es más simple
  - representación con listas de adyacencia:  
reiterando Dijkstra + colas con prioridad está en  $\Theta(n \log n)$
- Espacio:
  - Floyd exige  $\Theta(n^2)$  mientras que Dijkstra precisa  $\Theta(n)$
- Ejercicio: cálculo de las secuencias de nodos que componen los caminos mínimos
  - si el camino mínimo de  $m$  a  $n$  pasa primero por  $p$  y después por  $q$ , la secuencia de vértices que forman el camino mínimo de  $p$  a  $q$  forma parte de la secuencia de vértices que forman el camino mínimo de  $m$  a  $n$
  - usar un vector bidimensional  $C$  indexado por vértices:  $C[v,w]$  contiene un nodo  $u$  que forma parte del camino mínimo entre  $v$  y  $w$



# Árboles binarios de búsqueda óptimos

- Recordar **árbol binario de búsqueda**:
  - La clave de todo nodo es mayor o igual que las de sus descendientes izquierdos y menor que las de sus descendientes derechos.
- El problema:
  - Se tiene un conjunto de claves distintas:  $w_1 < w_2 < \dots < w_n$  (ordenadas alfab) que deben almacenarse en un árbol binario de búsqueda.
  - Se conoce la probabilidad  $p_i$ ,  $1 \leq i \leq n$ , con la que se pide buscar la clave  $w_i$  y su información asociada.
  - Se conoce también la probabilidad  $q_i$ ,  $0 \leq i \leq n$ , de búsqueda de una clave inexistente situada entre  $w_i$  y  $w_{i+1}$  (con el significado obvio para  $q_0$  y  $q_n$ ).
  - Se tiene que
$$\sum_{i=1}^n p_i + \sum_{i=0}^n q_i = 1$$
  - Se quiere construir un árbol binario de búsqueda para guardar las claves que minimice el número medio de comparaciones para encontrar una clave o para garantizar que no está.





# Árboles binarios de búsqueda óptimos

- Recordar que la profundidad de la raíz es 0, la de sus hijos es 1, etc.
- Si construimos un árbol en el que la clave  $w_i$  está en un nodo de profundidad  $d_i$ ,  $1 \leq i \leq n$ , entonces se necesitan  $d_i + 1$  comparaciones para encontrarla.
- Si con probabilidad  $q_i$ ,  $0 \leq i \leq n$ , buscamos una clave que no está en el árbol pero que, en caso de estar, ocuparía un nodo de profundidad  $d_i$  entonces se necesitan  $d_i$  comparaciones para garantizar que no está.
- Por tanto, el número medio de comparaciones para encontrar una clave o para garantizar que no está (función que queremos minimizar) es:

$$C = \sum_{i=1}^n p_i (d_i + 1) + \sum_{i=0}^n q_i d_i$$

- Ejemplo:

$$q_i = 0, \quad 0 \leq i \leq 7$$

Palabra	Probabilidad
a	0,22
al	0,18
ama	0,20
eso	0,05
si	0,25
sin	0,02
su	0,08

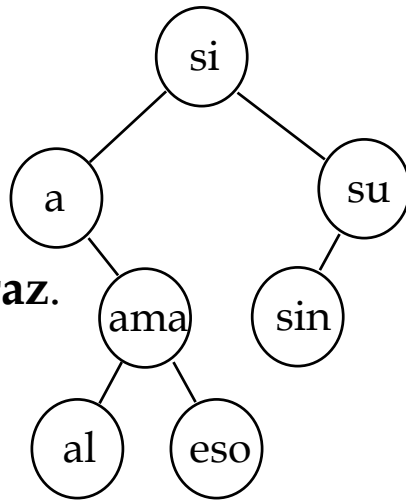


# Árboles binarios de búsqueda óptimos

- Solución 1:

Creada con estrategia **voraz**.

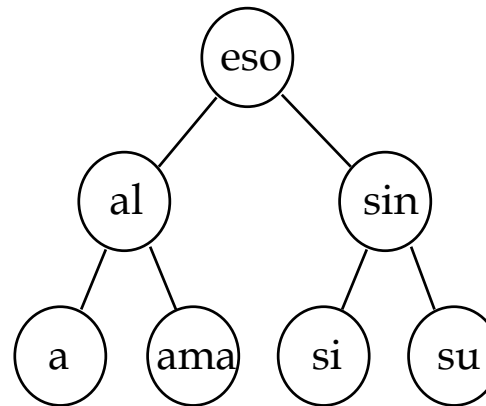
$C=2,43$



- Solución 2:

Árbol perfectamente equilibrado.

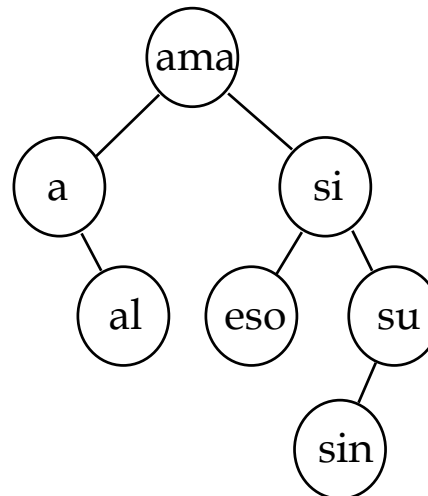
$C=2,70$



- Solución 3:

Es óptima.

$C=2,15$



# Árboles binarios de búsqueda óptimos

E.N. Gilbert y E.F. Moore: “Variable length encodings”,  
*Bell System Technical Journal*, 38(4), pp. 933-968, 1959.

- Solución de programación dinámica:

Principio de optimalidad:

“Todos los subárboles de un árbol óptimo son óptimos con respecto a las claves que contienen.”

- Consideremos un subárbol óptimo que contenga las claves  $w_{i+1}, w_{i+2}, \dots, w_j$ .
- La probabilidad de que una clave buscada esté o debiera estar en ese subárbol es:

$$m_{ij} = \sum_{k=i+1}^j p_k + \sum_{k=i}^j q_k$$

- Denotemos por  $C_{ij}$  el número medio de comparaciones efectuadas en un subárbol óptimo que contiene las claves  $w_{i+1}, w_{i+2}, \dots, w_j$  durante la búsqueda de una clave en el árbol principal (y convenimos en que  $C_{ii}=0$ ).
- Supongamos ahora que  $w_k$  ocupa la raíz de ese subárbol.
- Sea  $C_{ij}^K$  el número medio de comparaciones efectuadas en ese subárbol durante la búsqueda de una clave en el árbol principal.



# Árboles binarios de búsqueda óptimos

– Entonces:

$$C_{ij}^k = m_{ij} + C_{i,k-1} + C_{kj}$$

- $C_{i,k-1}$  es el nº medio de comparaciones en el subárbol izquierdo.
- $C_{kj}$  es el nº medio de comparaciones en el subárbol derecho.
- $m_{ij}$  es el nº medio de comparaciones con la raíz.

– Ahora se trata de escoger la raíz de forma que se minimice  $C_{ij}$ :

$$C_{ij} = m_{ij} + \min_{i < k \leq j} \{C_{i,k-1} + C_{kj}\}, \quad \text{si } 0 \leq i < j \leq n$$
$$C_{ij} = 0, \quad \text{si } 0 \leq i \leq n$$



# Árboles binarios de búsqueda óptimos

- El ejemplo:  $q_i = 0, 0 \leq i \leq 7$

Palabra	Probabilidad
a	0,22
al	0,18
ama	0,20
eso	0,05
si	0,25
sin	0,02
su	0,08

$$C_{ij} = m_{ij} + \min_{i < k \leq j} \{C_{i,k-1} + C_{kj}\}, \text{ si } 0 \leq i < j \leq n$$

$$C_{ij} = 0, \text{ si } 0 \leq i \leq n$$

$C =$

	0	1	2	3	4	5	6	7	
0	0	0,22	0,58	1,02	1,17	1,83	1,89	2,15	7°
1		0	0,18	0,56	0,66	1,21	1,27	1,53	6°
2			0	0,20	0,30	0,80	0,84	1,02	5°
3				0	0,05	0,35	0,39	0,57	4°
4					0	0,25	0,29	0,47	3°
5						0	0,02	0,12	2°
6							0	0,08	1°
7								0	



# Árboles binarios de búsqueda óptimos

```
tipos probP=vector[1..n] de real;  
      probQ=vector[0..n] de real;  
      matC=vector[0..n,0..n] de real;  
      matSol=vector[0..n,0..n] de entero
```

```
algoritmo abbÓpt(ent p:probP; ent q:probQ; sal C:matC; sal r:matSol)  
{C es la matriz definida previamente. En cada componente i,j de r se  
  guarda el k para el que C[i,j] resulta mínimo.}  
variables i,j,k,d:entero; min,aux:real; m:matC  
principio  
  para i:=0 hasta n hacer  
    C[i,i]:=0; m[i,i]:=q[i];  
    para j:=i+1 hasta n hacer  
      m[i,j]:=m[i,j-1]+p[j]+q[j]  
    fpara  
      fpara;  
      para j:=1 hasta n hacer  
        C[j-1,j]:=m[j-1,j]; r[j-1,j]:=j  
      fpara;  
{Ya están determinados los árboles de 1 nodo.} ...
```

# Árboles binarios de búsqueda óptimos

```
...
para d:=2 hasta n hacer
  para j:=d hasta n hacer
    i:=j-d;
    min:=maxEntero;
    para k:=i+1 hasta j hacer
      aux:=C[i,k-1]+C[k,j];
      si aux<min entonces
        min:=aux;
        r[i,j]:=k
      fsi
    fpara;
    C[i,j]:=m[i,j]+min
  fpara
fpara
fin
```



# Árboles binarios de búsqueda óptimos

```
tipos vectClaves=vector[1..n] de cadena;  
árbol= $\uparrow$ nodo;  
nodo=registro  
    dato:cadena;  
    iz,de:árbol  
freg
```

```
algoritmo creaABB(ent w:vectClaves; ent r:matSol; sal a:árbol)  
  algoritmo creaRec(sal a:árbol; ent i,j:entero)  
  principio  
    si i=j entonces a:=nil  
    sino  
      nuevoDato(a);  
      a $\uparrow$ .dato:=w[r[i,j]];  
      creaRec(a $\uparrow$ .iz,i,r[i,j]-1);  
      creaRec(a $\uparrow$ .de,r[i,j],j)  
    fsi  
  fin  
principio  
  creaRec(a,0,n)  
fin
```





# Árboles binarios de búsqueda óptimos

- Complejidad:

$\Theta(n^3)$ , usando  $\Theta(n^2)$  posiciones de memoria.

- Es posible transformar automáticamente ciertos algoritmos cúbicos de programación dinámica, como este, en algoritmos cuadráticos...

F.F. Yao: “Efficient dynamic programming using quadrangle inequalities”,  
*Proceedings of the 12th Annual ACM Symposium on the Theory of Computing*, pp. 429-435, 1980.



# Árboles binarios de búsqueda óptimos

D.E. Knuth: “Optimum binary search trees”,  
*Acta Informatica*, 1, pp. 14-25, 1971.

- En este caso concreto basta con demostrar que:

$$r[i, j-1] \leq r[i, j] \leq r[i+1, j], \text{ si } j-i \geq 2$$

(por inducción en  $j-i$  [Knu87, p. 492] D.E. Knuth. *El arte de programar ordenadores. Volumen III: Clasificación y búsqueda*. Editorial Reverté, 1987)

Ahora, el coste de los dos bucles internos de  $abb_{\text{ópt}}$  es:

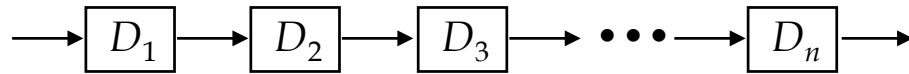
$$\begin{aligned} \sum_{\substack{d \leq j \leq n \\ i = j-d}} (r[i+1, j] - r[i, j-1] + 1) \\ = r[n-d+1, n] - r[0, d-1] + n - d + 1 < 2n \end{aligned}$$

Por tanto, el coste es  $\Theta(n^2)$ .



# Un problema de fiabilidad de sistemas

- El problema:
  - Diseñar un sistema compuesto de varios dispositivos conectados en serie.



- Sea  $r_i$  la **fiabilidad** de  $D_i$ , i.e., la probabilidad de que funcione correctamente.
- Entonces, la fiabilidad del sistema sistema entero es:

$$\prod_{i=1}^n r_i$$

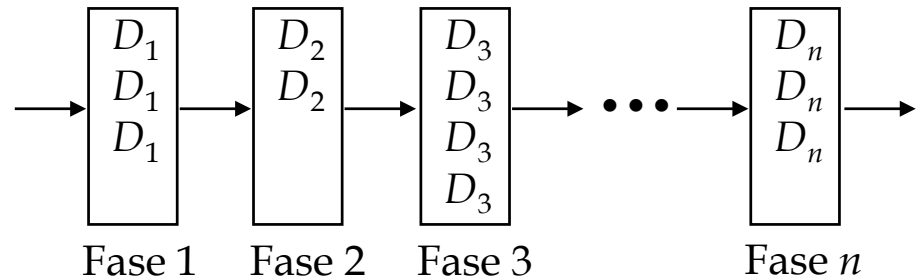
- Por ejemplo, si  $n=10$  y  $r_i=0,99$ ,  $1 \leq i \leq 10$ , la fiabilidad de cada dispositivo es muy alta y sin embargo

$$\prod_{i=1}^{10} r_i = 0,904$$



# Un problema de fiabilidad de sistemas

- Una forma de aumentar la fiabilidad es replicar los dispositivos (en paralelo).



- Si la fase  $i$  contiene  $m_i$  copias de  $D_i$ , la probabilidad de que toda la fase falle es  $(1 - r_i)^{m_i}$
- Luego la fiabilidad de la fase  $i$  es  $1 - (1 - r_i)^{m_i}$
- Por tanto, si  $r_i = 0,99$  y  $m_i = 2$ , la fiabilidad de la fase  $i$  es 0,9999.
- En realidad, la fiabilidad de la fase  $i$  es algo menor que  $1 - (1 - r_i)^{m_i}$  (las copias de un mismo dispositivo no son completamente independientes pues su diseño es común, por ejemplo); si denotamos la fiabilidad de la fase  $i$  por  $\phi_i(m_i)$
- entonces la fiabilidad del sistema es:

$$\prod_{1 \leq i \leq n} \phi_i(m_i)$$



# Un problema de fiabilidad de sistemas

- El problema: maximizar la fiabilidad replicando los dispositivos y con alguna limitación en el coste.

$$\text{maximizar } \prod_{1 \leq i \leq n} \phi_i(m_i)$$

$$\text{sujeto a } \sum_{1 \leq i \leq n} c_i m_i \leq c$$

$$m_i \geq 1 \text{ y entero, } 1 \leq i \leq n$$

Donde  $c_i$  es el coste de cada unidad de dispositivo  $i$ .

- Como  $c_i > 0$  y  $m_j \geq 1$ , entonces  $1 \leq m_i \leq u_i$  con

$$u_i = \left\lfloor \left( c + c_i - \sum_{j=1}^n c_j \right) / c_i \right\rfloor$$



# Un problema de fiabilidad de sistemas

– Una solución óptima  $m_1, m_2, \dots, m_n$  es el resultado de una secuencia de decisiones, una por cada  $m_i$ .

– Denotemos:

$$f_i(x) = \text{máximo} \prod_{1 \leq j \leq i} \phi_j(m_j)$$

sujeto a  $\sum_{1 \leq j \leq i} c_j m_j \leq x$

$$1 \leq m_j \leq u_j, \quad 1 \leq j \leq i$$

Entonces el valor de una solución óptima es  $f_n(c)$ .



# Un problema de fiabilidad de sistemas

- La última decisión requiere elegir  $m_n$  de entre  $\{1,2,3,\dots,u_n\}$ .
- Una vez tomada la última decisión, las restantes decisiones deben utilizar el resto de fondos  $c - c_n m_n$  de forma óptima.
- Se cumple el principio de optimalidad y

$$f_n(c) = \max_{1 \leq m_n \leq u_n} \{ \phi_n(m_n) f_{n-1}(c - c_n m_n) \}$$

- En general, para  $f_i(x)$ ,  $i \geq 1$ , se tiene:

$$f_i(x) = \max_{1 \leq m_i \leq u_i} \{ \phi_i(m_i) f_{i-1}(x - c_i m_i) \}$$

$$f_0(x) = 1, \text{ para todo } x, 0 \leq x \leq c$$

(ver, además, <http://webdiis.unizar.es/asignaturas/AB/?p=1292> )

- Se resuelve de forma similar al problema de la mochila 0-1 (ejercicio).



# El problema del viajante de comercio

- Recordar:
  - Encontrar un recorrido de longitud mínima para un viajante que tiene que visitar varias ciudades y volver al punto de partida, conocida la distancia existente entre cada dos ciudades.
  - Es decir, dado un grafo dirigido con arcos de longitud no negativa, se trata de encontrar un circuito de longitud mínima que comience y termine en el mismo vértice y pase exactamente una vez por cada uno de los vértices restantes (circuito *hamiltoniano*).





# El problema del viajante de comercio

- Sean  $G=(V,A)$  un grafo orientado,  
 $V=\{1,2,\dots,n\}$ ,  
 $L_{ij}$  la longitud de  $(i,j)\in A$ ,  
 $L_{ij}=\infty$  si no existe el arco  $(i,j)$ .
- El circuito buscado empieza en el vértice 1.  
Se compone de  $(1,j)$ , con  $j\neq 1$ , seguido de un camino de  $j$  a 1 que pasa exactamente una vez por cada vértice de  $V\setminus\{1,j\}$ .
- Principio de optimalidad: si el circuito es óptimo, el camino de  $j$  a 1 debe serlo también.
- Sea  $S\subseteq V\setminus\{1\}$  un subconjunto de vértices e  $i\in V\setminus S$  un vértice;  
llamamos  $g(i,S)$  a la longitud del camino mínimo desde  $i$  hasta 1 que pase exactamente una vez por cada vértice de  $S$ .

Entonces:

$$\begin{aligned}\text{longitud del circuito óptimo} &= g(1, V \setminus \{1\}) = \\ &= \min_{2 \leq j \leq n} \{L_{1j} + g(j, V \setminus \{1, j\})\}\end{aligned}$$



# El problema del viajante de comercio

---

- Más en general, si  $i \neq 1$ ,  $S \neq \emptyset$  e  $i \notin S$ :

$$g(i, S) = \min_{j \in S} \{L_{ij} + g(j, S \setminus \{j\})\} \quad (*)$$

Además:

$$g(i, \emptyset) = L_{i1}, \quad i = 2, 3, \dots, n$$

- Método de resolución:
  - Usar (\*) y calcular  $g$  para todos los conjunto  $S$  con un solo vértice (distinto del 1).
  - Volver a usar (\*) y calcular  $g$  para todos los conjuntos  $S$  de dos vértices (distintos del 1) y así sucesivamente.
  - Cuando se conoce el valor de  $g$  para todos los conjuntos  $S$  a los que sólo les falta un vértice (distinto del 1) basta calcular  $g(1, V \setminus \{1\})$ .



# El problema del viajante de comercio

- Ejemplo. Sea  $G$  el grafo completo de cuatro vértices con longitudes:

$$L = \begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$$

– Inicialización:

$$g(2, \emptyset) = 5; \quad g(3, \emptyset) = 6; \quad g(4, \emptyset) = 8.$$

Usar  $g(i, S) = \min_{j \in S} \{L_{ij} + g(j, S \setminus \{j\})\}$  para obtener:

$$g(2, \{3\}) = L_{23} + g(3, \emptyset) = 15;$$

$$g(2, \{4\}) = L_{24} + g(4, \emptyset) = 18;$$

$$g(3, \{2\}) = 18; \quad g(3, \{4\}) = 20;$$

$$g(4, \{2\}) = 13; \quad g(4, \{3\}) = 15.$$



# El problema del viajante de comercio

- Ahora, utilizando de nuevo (\*) para conjuntos de dos elementos:

$$\begin{aligned}g(2, \{3, 4\}) &= \min \{L_{23} + g(3, \{4\}), L_{24} + g(4, \{3\})\} = \\ &= \min \{29, 25\} = 25;\end{aligned}$$

$$\begin{aligned}g(3, \{2, 4\}) &= \min \{L_{32} + g(2, \{4\}), L_{34} + g(4, \{2\})\} = \\ &= \min \{31, 25\} = 25;\end{aligned}$$

$$\begin{aligned}g(4, \{2, 3\}) &= \min \{L_{42} + g(2, \{3\}), L_{43} + g(3, \{2\})\} = \\ &= \min \{23, 27\} = 23.\end{aligned}$$

- Finalmente:

$$\begin{aligned}g(1, \{2, 3, 4\}) &= \min \{ L_{12} + g(2, \{3, 4\}), \\ &L_{13} + g(3, \{2, 4\}), \\ &L_{14} + g(4, \{2, 3\}) \} = \\ &= \min \{ 35, 40, 43 \} = 35.\end{aligned}$$



# El problema del viajante de comercio

- Si además se quiere saber cómo se construye el circuito óptimo:

Utilizar una función adicional

$J(i,S)$  es el valor de  $j$  que minimiza  $g(i,S)$  al aplicar la fórmula (\*).

- En el ejemplo:

$$J(2, \{3,4\}) = 4; \quad J(3, \{2,4\}) = 4;$$

$$J(4, \{2,3\}) = 2; \quad J(1, \{2,3,4\}) = 2.$$

Y el circuito óptimo será:

$$1 \rightarrow J(1, \{2,3,4\}) = 2$$

$$\rightarrow J(2, \{3,4\}) = 4$$

$$\rightarrow J(4, \{3\}) = 3$$

$$\rightarrow 1$$



# El problema del viajante de comercio

- Implementación recursiva ineficiente:

```
función g(i,S) devuelve nat
variables másCorto,distancia,j:nat
principio
  si S=∅ entonces devuelve L[i,1]
  sino
    másCorto:=∞;
    para todo j en S hacer
      distancia:=L[i,j]+g(j,S\{j});
      si distancia<másCorto entonces
        másCorto:=distancia
      fsi
    fpara;
    devuelve másCorto
  fsi
fin
```

Se calcula repetidas veces el mismo valor de g:  $\Omega((n-1)!)$



# El problema del viajante de comercio

- Utilización de una “función con memoria”:

```
{se usa una tabla gtab cuyos elementos se inicializan con -1}
función g(i,S) devuelve nat
variables másCorto,distancia,j:nat
principio
  si S=∅ entonces devuelve L[i,1]
  sino
    si gtab[i,S]≥0 entonces devuelve gtab[i,S]
    sino
      másCorto:=∞;
      para todo j en S hacer
        distancia:=L[i,j]+g(j,S\{j});
        si distancia<másCorto entonces másCorto:=distancia fsi
      fpara;
      gtab[i,S]:=másCorto;
      devuelve másCorto
    fsi
  fsi
fin
```



# El problema del viajante de comercio

- Coste del algoritmo:

- cálculo de  $g(j, \emptyset)$ :  $n-1$  consultas a una tabla,
- cálculo de los  $g(j, S)$  tales que  $1 \leq \text{card}(S) = k \leq n-2$ :

$$(n-1) \binom{n-2}{k} k \text{ sumas en total } ,$$

- cálculo de  $g(1, V \setminus \{1\})$ :  $n-1$  sumas.

→ Tiempo de cálculo:  $\Theta\left(2(n-1) + \sum_{k=1}^{n-2} (n-1)k \binom{n-2}{k}\right) = \Theta(n^2 2^n)$

puesto que  $\sum_{k=1}^r k \binom{r}{k} = r 2^{r-1}$

(Este tiempo es mejor que  $\Omega(n!)$  que resultaría de la estrategia de fuerza bruta, pero...)

- Coste en espacio (para conservar  $g$  y  $J$ ):  $\Omega(n 2^n)$





# El problema del viajante de comercio

- Para hacernos una idea del coste...

número de vértices	tiempo fuerza bruta	tiempo prog. dinámica	espacio prog. dinámica
$n$	$n!$	$n^2 2^n$	$n 2^n$
5	120	800	160
10	3.628.800	102.400	10.240
15	$1,31 \times 10^{12}$	7.372.800	491.520
20	$2,43 \times 10^{18}$	419.430.400	20.971.520
25	$1,55 \times 10^{25}$	20.971.520.000	838.860.800
30	$2,65 \times 10^{32}$	966.367.641.600	32.212.254.720
50	$3,04 \times 10^{64}$	$2,81 \times 10^{18}$	$5,62 \times 10^{16}$

Si las unidades son microsegundos:

menos de 7 minutos

más de 77.000 años



# Planificación de trabajos

- El problema:

Sea un sistema en el que la realización de un conjunto de trabajos requiere la ejecución por parte de un conjunto de agentes (o procesadores) de una serie de tareas diferentes para cada trabajo.

- $n$  trabajos requiriendo cada uno  $m$  tareas:

$$T_{1i}, T_{2i}, \dots, T_{mi}, 1 \leq i \leq n$$

- la tarea  $T_{ji}$  la realiza el procesador  $P_j$ ,  $1 \leq j \leq m$ , y requiere un tiempo  $t_{ji}$

- Planificación para los  $n$  trabajos:

Es una asignación de tareas a intervalos de tiempo en los procesadores.

- la tarea  $T_{ji}$  debe asignarse a  $P_j$
- un procesador no puede tener más de una tarea asignada en cada instante de tiempo
- para todo trabajo  $i$ , el procesamiento de  $T_{ji}$ ,  $j > 1$ , no puede empezar hasta que  $T_{j-1,i}$  haya terminado



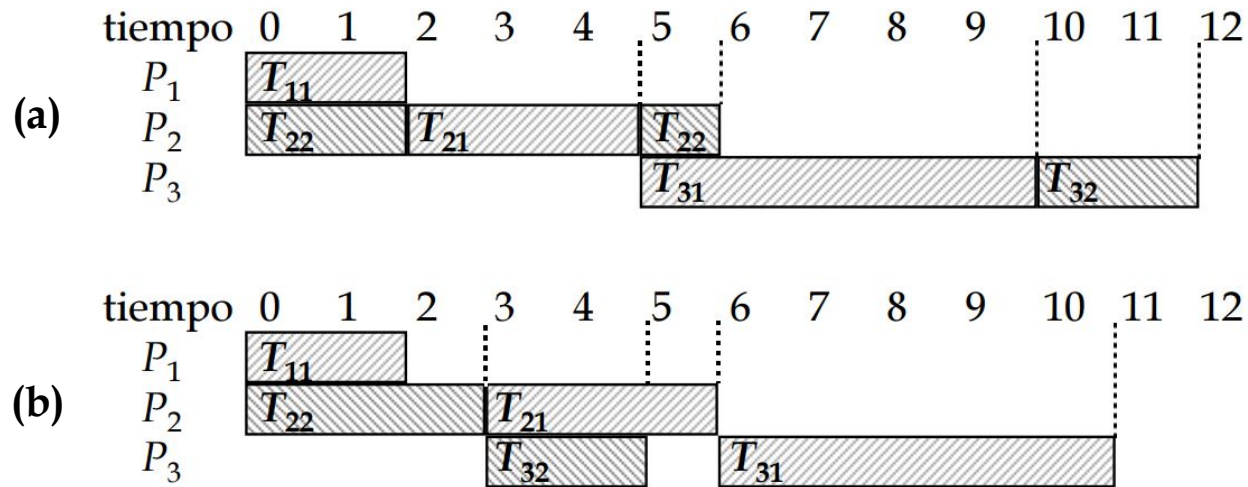
# Planificación de trabajos

- Ejemplo:

Se tiene que planificar la ejecución de dos trabajos en tres procesadores, de forma que los tiempos de cada tarea vienen dados por:

$$T = \begin{bmatrix} 2 & 0 \\ 3 & 3 \\ 5 & 2 \end{bmatrix}$$

Dos planificaciones posibles:



La planificación (b) se dice no-interruptiva (*non-preemptive*) porque el procesamiento de una tarea no se interrumpe hasta que ésta ha terminado.

La planificación (a) se dice interruptiva (*preemptive*) porque el trabajo 1 se apropia del procesador 2 antes de que éste termine con el trabajo 2.



# Planificación de trabajos

- El tiempo de terminación del trabajo  $i$  en la planificación  $S$  es el instante,  $f_i(S)$ , en que todas las tareas del trabajo  $i$  han terminado.

En el ejemplo (a),  $f_1(S_a)=10$  y  $f_2(S_a)=12$ .

En el ejemplo (b),  $f_1(S_b)=11$  y  $f_2(S_b)=5$ .

- El tiempo de terminación,  $f(S)$ , de la planificación  $S$  es:

$$F(S) = \max_{1 \leq i \leq n} \{f_i(S)\}$$

- El **tiempo medio de terminación**,  $\text{MFT}(S)$ , se define como:

$$\text{MFT}(S) = \frac{1}{n} \sum_{1 \leq i \leq n} f_i(S)$$



# Planificación de trabajos

---

- Planificación con tiempo de terminación óptimo (OFT) para un conjunto de trabajos:  
es una planificación no-interruptiva,  $S$ , para la que  $F(S)$  es mínimo entre todas las planificaciones no-interruptivas.
- Planificación interruptiva y con tiempo de terminación óptimo (POFT):  
es una planificación interruptiva,  $S$ , para la que  $F(S)$  es mínimo entre todas las planificaciones interruptivas.
- Planificación con tiempo medio de terminación óptimo (OMFT):  
es una planificación no-interruptiva,  $S$ , para la que  $MFT(S)$  es mínimo entre todas las planificaciones no-interruptivas.
- Planificación interruptiva y con tiempo medio de terminación óptimo (POMFT):  
es una planificación interruptiva,  $S$ , para la que  $MFT(S)$  es mínimo entre todas las planificaciones interruptivas.



# Planificación de trabajos

- El cálculo de OFT y POFT para  $m > 2$  y el cálculo de OMFT es computacionalmente difícil (es *NP-duro*).
- El cálculo de OFT para  $m = 2$  puede hacerse mediante programación dinámica.
- Caso  $m = 2$ :
  - Denotemos  $T_{1i}$  como  $a_i$  y  $T_{2i}$  como  $b_i$ .
  - Una planificación está completamente especificada fijando una permutación de los trabajos en uno de los procesadores (coincidirá con el otro procesador).

Cada tarea empezará tan pronto como sea posible.

Ejemplo con 5 trabajos:



planificación (5,1,3,2,4)

# Planificación de trabajos

---

- Supongamos, para simplificar, que  $a_i \neq 0$ ,  $1 \leq i \leq n$   
(si hay trabajos con  $a_i = 0$ , se construye primero la planificación óptima para los trabajos con  $a_i \neq 0$  y después se añaden delante los trabajos con  $a_i = 0$ ).

- Principio de optimalidad:

Una permutación (planificación) óptima es tal que, fijado el primer trabajo de la permutación, el resto de la permutación es óptimo con respecto al estado en que quedan los dos procesadores después de terminar el primer trabajo.



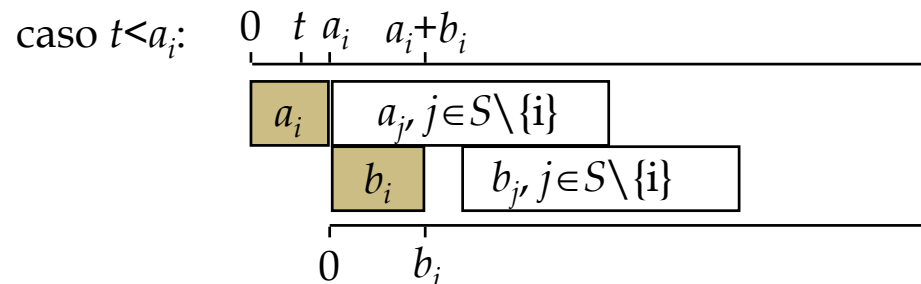
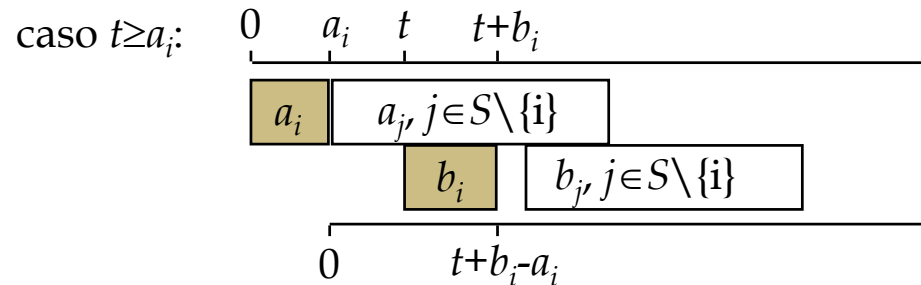
# Planificación de trabajos

- Sea  $g(S,t)$  la longitud (duración) de una planificación óptima para el subconjunto de trabajos  $S$  suponiendo que el procesador 2 no estará disponible hasta el instante  $t$ .

Entonces:

$$g(S,t) = \min_{i \in S} \{a_i + g(S \setminus \{i\}, b_i + \max\{t - a_i, 0\})\}$$

con  $g(\emptyset, t) = \max\{t, 0\}$  y  $a_i \neq 0, 1 \leq i \leq n$ .





# Planificación de trabajos

- La ecuación recursiva resultante podría resolverse de forma análoga a la del problema del viajante de comercio, pero existe una solución mejor...
- Supongamos que  $i$  y  $j$  son los dos primeros trabajos (y en ese orden) en la planificación óptima del subconjunto  $S$ ; entonces:

$$\begin{aligned} g(S, t) &= a_i + g(S \setminus \{i\}, b_i + \max\{t - a_i, 0\}) = \\ &= a_i + a_j + \underbrace{g(S \setminus \{i, j\}, b_j + \max\{b_i + \max\{t - a_i, 0\} - a_j, 0\})}_{t_{ij}} \end{aligned}$$

Pero:

$$\begin{aligned} t_{ij} &= b_j + \max\{b_i + \max\{t - a_i, 0\} - a_j, 0\} = \\ &= b_j + b_i - a_j + \max\{\max\{t - a_i, 0\}, a_j - b_i\} = \\ &= b_j + b_i - a_j + \max\{t - a_i, a_j - b_i, 0\} = \\ &= b_j + b_i - a_j - a_i + \max\{t, a_i + a_j - b_i, a_i\} \end{aligned}$$

Si los dos primeros trabajos fueran  $j$  e  $i$ :

$$g'(S, t) = a_j + a_i + g'(S \setminus \{j, i\}, b_i + b_j - a_i - a_j + \max\{t, a_j + a_i - b_j, a_j\})$$



# Planificación de trabajos

- Entonces:  $g(S, t) \leq g'(S, t) \Leftrightarrow$   
 $\Leftrightarrow \max\{t, a_i + a_j - b_i, a_i\} \leq \max\{t, a_j + a_i - b_j, a_j\}$

Para que esto sea cierto para todo valor de  $t$ , se precisa:

$$\max\{a_i + a_j - b_i, a_i\} \leq \max\{a_j + a_i - b_j, a_j\}$$

Es decir:

$$a_i + a_j + \max\{-b_i, -a_j\} \leq a_j + a_i + \max\{-b_j, -a_i\}$$

O sea:

$$\min\{b_i, a_j\} \geq \min\{b_j, a_i\} \quad (*)$$

- Luego existe una planificación óptima en la que cada par  $(i, j)$  de trabajos adyacentes verifica (\*).
- Puede demostrarse que todas las planificaciones que verifican (\*) tienen la misma longitud.

Por tanto, basta generar una permutación para la que se cumpla (\*) para todo par de trabajos adyacentes.



# Planificación de trabajos

---

- Ahora, si

$$\min \{a_1, a_2, \dots, a_n, b_1, b_2, \dots, b_n\} = a_i$$

Entonces el trabajo  $i$  debería ser el primero en una planificación óptima.

- En cambio, si

$$\min \{a_1, a_2, \dots, a_n, b_1, b_2, \dots, b_n\} = b_j$$

Entonces el trabajo  $j$  debería ser el último en una planificación óptima.

- Luego podemos decidir la posición de uno de los trabajos (el primero o el último).

Repitiendo el proceso, se puede construir la planificación óptima.



# Planificación de trabajos

---

- Por tanto la solución es:
  - i) ordenar los  $a_i$  y  $b_i$  en orden no decreciente;
  - ii) si el siguiente número de la secuencia es  $a_i$  y el trabajo  $i$  no ha sido planificado todavía, planificar el trabajo  $i$  en la posición más a la izquierda de entre los que restan;  
si el siguiente número es  $b_j$  y el trabajo  $j$  no ha sido planificado todavía, planificar el trabajo  $j$  en la posición más a la derecha de entre los que restan;

(nótese que el algoritmo sirve también si hay trabajos con  $a_i=0$ )



# Planificación de trabajos

- Ejemplo:
  - Sean  $n=4$ ,  $(a_1, a_2, a_3, a_4) = (3, 4, 8, 10)$  y  $(b_1, b_2, b_3, b_4) = (6, 2, 9, 15)$ .
  - La secuencia ordenada de los  $a_i$  y los  $b_i$  es:  
 $(b_2, a_1, a_2, b_1, a_3, b_3, a_4, b_4) = (2, 3, 4, 6, 8, 9, 10, 15)$ .
  - Sea  $\sigma_1, \sigma_2, \sigma_3, \sigma_4$  la secuencia óptima.
    - Como el número menor es  $b_2$ , entonces  $\sigma_4=2$ .
    - El siguiente número es  $a_1$  luego  $\sigma_1=1$ .
    - El siguiente es  $a_2$  pero el trabajo 2 ya ha sido planificado.
    - El siguiente es  $b_1$  pero 1 ya ha sido planificado.
    - El siguiente es  $a_3$  luego hacemos  $\sigma_2=3$ .
    - Por tanto,  $\sigma_3=4$ .



# Planificación de trabajos

---

- Coste:  $O(n \log n)$
- Nótese que la solución directa de

$$g(S, t) = \min_{i \in S} \{a_i + g(S \setminus \{i\}, b_i + \max\{t - a_i, 0\})\}$$

hubiera llevado al menos  $O(2^n)$ , que es el número de subconjuntos  $S$  diferentes para los que habría que calcular  $g(S, t)$ .



# Una competición internacional

- El problema:

Dos equipos  $A$  y  $B$  se enfrentan un máximo de  $2n - 1$  veces, ganando el primer equipo que acumule  $n$  victorias.

Supongamos que no es posible un partido nulo, que los resultados de cada partido son independientes y que hay una probabilidad constante  $p$  de que  $A$  gane un partido y  $q = 1 - p$  de que lo gane  $B$ .

¿Cuál es la probabilidad, a priori, de que gane  $A$ ?

- El planteamiento:

- Sea  $P(i,j)$  la probabilidad de que  $A$  gane la competición sabiendo que le faltan todavía  $i$  victorias y a  $B$  le faltan  $j$ .
- Entonces, antes del primer partido, la probabilidad de que  $A$  gane la competición es  $P(n,n)$ .
- Si  $A$  ya ha acumulado todas las victorias necesarias entonces es el ganador de la competición, es decir:  $P(0,j) = 1$ ,  $1 \leq j \leq n$ .
- Igualmente,  $P(i,0) = 0$ , para  $1 \leq i \leq n$ .
- La ecuación recurrente es:

$$P(i,j) = pP(i-1,j) + qP(i,j-1), \text{ para } i,j \geq 1.$$



# Una competición internacional

- La solución directa:

```
función P(i,j) devuelve real
{pp (pq) es la probabilidad de que gane A (B)}
principio
  si i=0 entonces
    devuelve 1
  sino
    si j=0 entonces
      devuelve 0
    sino
      devuelve pp*P(i-1,j)+pq*P(i,j-1)
  fsi
fisi
fin
```





# Una competición internacional

- Coste de la solución directa:

- Sea  $T(k)$  el tiempo necesario en el caso peor para calcular  $P(i,j)$  con  $i+j=k$ .

$$T(1) = c$$

$$T(k) \leq 2T(k-1) + d, \quad k > 1$$

donde  $c$  y  $d$  son dos constantes.

- Luego,  $T(k)$  es  $O(2^k)$  y por tanto  $T(2n)$  es  $O(2^{2n})$ .

- Exactamente el número de llamadas recursivas es  $2 \binom{i+j}{j} - 2$

- Por tanto, el tiempo para calcular  $P(n,n)$  es  $\Omega\left(\binom{2n}{n}\right)$

y puede demostrarse que  $\binom{2n}{n} \geq 2^{2n}/(2n+1)$

- En definitiva, el tiempo de la solución directa para calcular  $P(n,n)$  es  $O(4^n)$  y  $\Omega(4^n/n)$ .

Puede demostrarse que el tiempo es  $\Theta\left(4^n/\sqrt{n}\right)$



# Una competición internacional

- El problema de la solución directa:
  - Como siempre en programación dinámica, se calcula muchas veces el valor de cada  $P(i,j)$ .
  - Solución mejor: usar una tabla para almacenar los  $P(i,j)$ .
  - Ejemplo (para  $p = q = 1/2$ ):

	1/2	21/32	13/16	15/16	1	4
	11/32	1/2	11/16	7/8	1	3
	3/16	5/16	1/2	3/4	1	2
	1/16	1/8	1/4	1/2	1	1
	0	0	0	0		0
	4	3	2	1	0	

←  $i$

$\uparrow$   
 $j$

$P(i,j) = pP(i-1,j) + qP(i,j-1)$

la matriz se completa, por ejemplo,  
 por filas (de abajo a arriba,  $j=0, 1, \dots, n$ )  
 y en cada fila por columnas (de dch. a izq.,  $i=0, 1, \dots, n$ )



# Una competición internacional

- La solución:

```
función apuestas(n:natural; p:real) devuelve real
variables tabP:vector[0..n,0..n]de real;
           q:real; i,j:natural
principio
  q:=1-p;
  para j:=1 hasta n hacer
    tabP[0,j]:=1; tabP[j,0]:=0;
    para i:=1 hasta n hacer
      tabP[i,j]:=p*tabP[i-1,j] + q*tabP[i,j-1]
    fpara
  fpara;
  devuelve tabP[n,n]
fin
```



# Una competición internacional

---

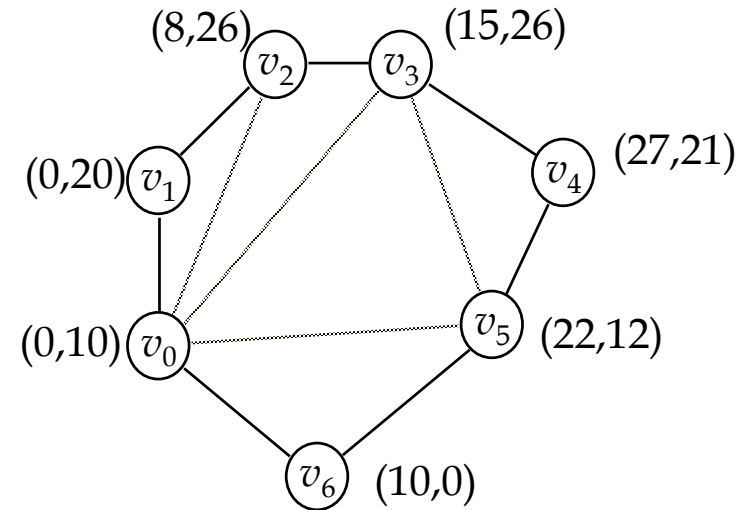
- Coste:
  - En tiempo:  $\Theta(n^2)$
  - En espacio:  $\Theta(n^2)$ , aunque se puede reducir fácilmente a  $\Theta(n)$



# Triangulación de polígonos

- Problema:

Dados los vértices de un polígono, se trata de seleccionar un conjunto de **cuerdas** (líneas entre vértices no adyacentes) de modo que ningún par de cuerdas se cruce entre sí y que todo el polígono quede dividido en triángulos.

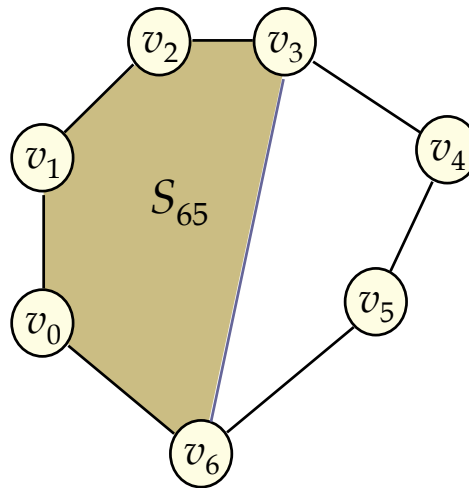


Además, la longitud total de las cuerdas debe ser mínima (triangulación minimal).

- Utilidad: se puede emplear para sombrear objetos tridimensionales en una imagen virtual (bidimensional).
- Otra: interpolación numérica de funciones de dos variables.

# Triangulación de polígonos

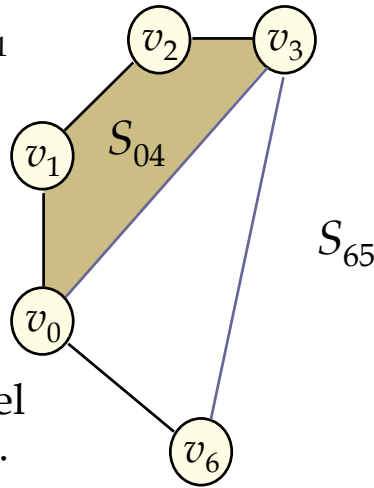
- Resolución de programación dinámica:
  - Consideremos un polígono definido por sus vértices  $v_0, v_1, \dots, v_{n-1}$ .
  - Sea  $S_{is}$  el subproblema de tamaño  $s$  partiendo del vértice  $v_i$ , es decir, el problema de la triangulación minimal del polígono formado por los  $s$  vértices que comienzan en  $v_i$  y siguen en el sentido de las agujas del reloj ( $v_i, v_{i+1}, \dots, v_{i+s-1}$ ), contando con la cuerda  $(v_i, v_{i+s-1})$ .



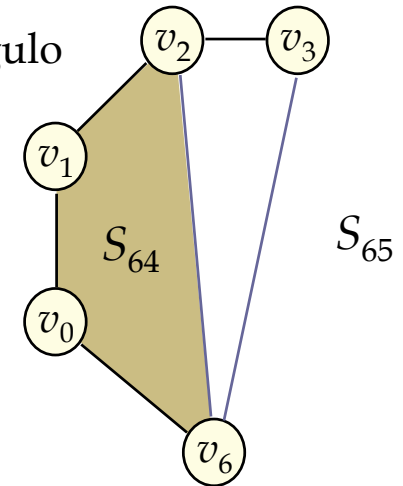
# Triangulación de polígonos

– Ahora, para triangular el polígono  $S_{is}$ , hay  $s - 2$  posibilidades:

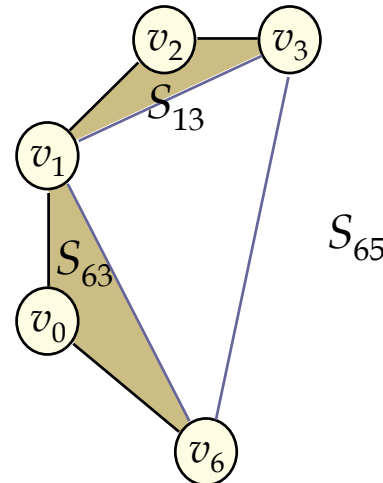
1. Tomar el vértice  $v_{i+1}$  para formar un triángulo con las cuerdas  $(v_i, v_{i+s-1})$  y  $(v_{i+1}, v_{i+s-1})$  y con el tercer lado  $(v_i, v_{i+1})$ , y después resolver el subproblema  $S_{i+1, s-1}$ .



2. Tomar el vértice  $v_{i+s-2}$  para formar un triángulo con las cuerdas  $(v_i, v_{i+s-1})$  y  $(v_i, v_{i+s-2})$  y con el tercer lado  $(v_{i+s-2}, v_{i+s-1})$ , y después resolver el subproblema  $S_{i, s-1}$ .



3. Para algún  $k$  entre 2 y  $s-3$ , tomar el vértice  $v_{i+k}$  y formar un triángulo con lados  $(v_i, v_{i+k})$ ,  $(v_{i+k}, v_{i+s-1})$  y  $(v_i, v_{i+s-1})$ , y después resolver los subproblemas  $S_{i, k+1}$  y  $S_{i+k, s-k}$ .



# Triangulación de polígonos

- Por tanto, si denotamos por  $C_{is}$  el coste de la triangulación  $S_{is}$ , se obtiene la siguiente relación recursiva:

$$C_{is} = \min_{1 \leq k \leq s-2} \left\{ C_{i, k+1} + C_{i+k, s-k} + D(v_i, v_{i+k}) + \right. \\ \left. + D(v_{i+k}, v_{i+s-1}) \right\}$$

para  $0 \leq i \leq n-1$ ,  $4 \leq s \leq n$ ;

donde:

$D(v_p, v_q)$  es la longitud de la cuerda entre los vértices  $v_p$  y  $v_q$  si  $v_p$  y  $v_q$  no son vértices adyacentes en el polígono; y  $D(v_p, v_q)$  es 0 si  $v_p$  y  $v_q$  son adyacentes.

Además,  $C_{is} = 0$  para  $0 \leq i \leq n-1$ ,  $2 \leq s < 4$ .





# Triangulación de polígonos

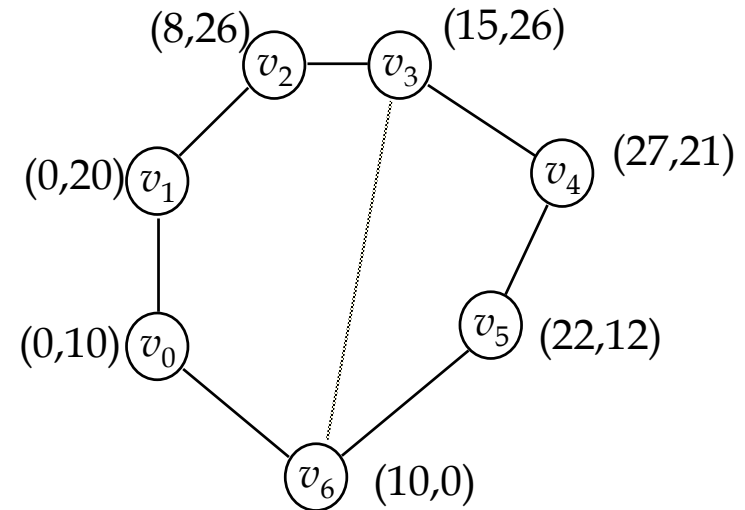
---

- Solución recursiva inmediata:
  - Aplicando la ecuación recurrente anterior
  - Problema: el número de llamadas crece exponencialmente con el número de vértices
  - Sin embargo, sin contar el problema original, sólo hay  $n(n-4)$  subproblemas diferentes que hay que resolver
  - Por tanto, la solución recursiva resuelve muchas veces un mismo subproblema
- Solución eficiente:
  - Utilización de una tabla para almacenar los costes de las soluciones de los subproblemas



# Triangulación de polígonos

$$C_{65} = \min \left\{ \begin{array}{l} (1) C_{62} + C_{04} + D(v_6, v_0) + D(v_0, v_3), \\ (2) C_{63} + C_{13} + D(v_6, v_1) + D(v_1, v_3), \\ (3) C_{64} + C_{23} + D(v_6, v_2) + D(v_2, v_3) \end{array} \right\} = \\ = \min \{ 38,09 ; 38,52 ; 43,97 \} = 38,09$$



7	$C_{07} =$ 75,43						
6	$C_{06} =$ 53,54	$C_{16} =$ 55,22	$C_{26} =$ 57,58	$C_{36} =$ 64,69	$C_{46} =$ 59,78	$C_{56} =$ 59,78	$C_{66} =$ 63,62
5	$C_{05} =$ 37,54	$C_{15} =$ 31,81	$C_{25} =$ 35,49	$C_{35} =$ 37,74	$C_{45} =$ 45,50	$C_{55} =$ 39,98	$C_{65} =$ 38,09
(1) 4	$C_{04} =$ 16,16	$C_{14} =$ 16,16	$C_{24} =$ 15,65	$C_{34} =$ 15,65	$C_{44} =$ 22,69	$C_{54} =$ 22,69	$C_{64} =$ 17,89
3	$C_{03} =$ 0	$C_{13} =$ 0 (2)	$C_{23} =$ 0 (3)	$C_{33} =$ 0	$C_{43} =$ 0	$C_{53} =$ 0	$C_{63} =$ 0 (2)
2	$C_{02} =$ 0	$C_{12} =$ 0	$C_{22} =$ 0	$C_{32} =$ 0	$C_{42} =$ 0	$C_{52} =$ 0	$C_{62} =$ 0 (1)
s	$i = 0$	1	2	3	4	5	6

$$D(v_2, v_3) = D(v_6, v_0) = 0$$

(son aristas y no cuerdas)

$$D(v_6, v_2) = 26,08;$$

$$D(v_1, v_3) = 16,16;$$

$$D(v_6, v_1) = 22,36;$$

$$D(v_0, v_3) = 21,93$$



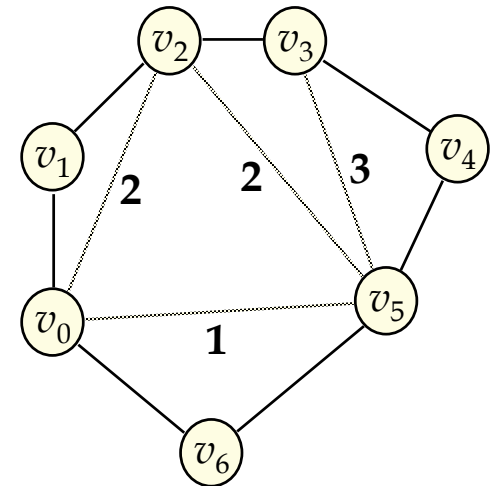
# Triangulación de polígonos

- Hemos calculado el coste de la triangulación mínima pero, ¿cuál es esa triangulación?
  - Para cada posición  $(i,s)$  de la tabla se necesita almacenar, además del coste, el valor del índice  $k$  que produjo el mínimo.
  - Entonces la solución consta de las cuerdas  $(v_i, v_{i+k})$  y  $(v_{i+k}, v_{i+s-1})$  (a menos que una de ellas no sea cuerda, porque  $k=1$  o  $k=s-2$ ), más las cuerdas que estén implicadas por las soluciones de  $S_{i,k+1}$  y  $S_{i+k,s-k}$ .



# Triangulación de polígonos

- En el ejemplo:
  - El valor de  $C_{07}$  procede de  $k=5$ . Es decir, el problema  $S_{07}$  se divide en  $S_{06}$  y  $S_{53}$ .  $S_{53}$  es un problema trivial, de coste 0. Así, se introduce la cuerda  $(v_0, v_5)$ , de coste 22'09, y se debe resolver  $S_{06}$ .



- El valor de  $C_{06}$  procede de  $k=2$ . Por tanto, el problema se divide en  $S_{03}$  y  $S_{24}$ .  $S_{03}$  es un triángulo con vértices  $v_0$ ,  $v_1$  y  $v_2$ , luego no precisa ser resuelto, mientras que  $S_{24}$  es un cuadrilátero, definido por  $v_2$ ,  $v_3$ ,  $v_4$  y  $v_5$ , y debe ser resuelto. Además, hay que incluir los costes de las cuerdas  $(v_0, v_2)$  y  $(v_2, v_5)$ , que son 17'89 y 19'80.
- El valor de  $C_{24}$  se obtiene con  $k=1$ , dando los subproblemas  $S_{22}$  y  $S_{33}$ , que tienen tamaño menor o igual que tres y, por tanto, coste 0. Se introduce la cuerda  $(v_3, v_5)$ , con coste 15'65.