Programación dinámica

Yolanda Ortega Mallén

Dpto. de Sistemas Informáticos y Computación

Universidad Complutense de Madrid

Sumario

- Un ejemplo: el triángulo de Pascal.
- Esquema general.
- Aplicaciones.

Coeficientes binomiales

$$\binom{n}{k} = \frac{n!}{k!\ (n-k)!} \quad \text{si } 0 \leq k \leq n$$

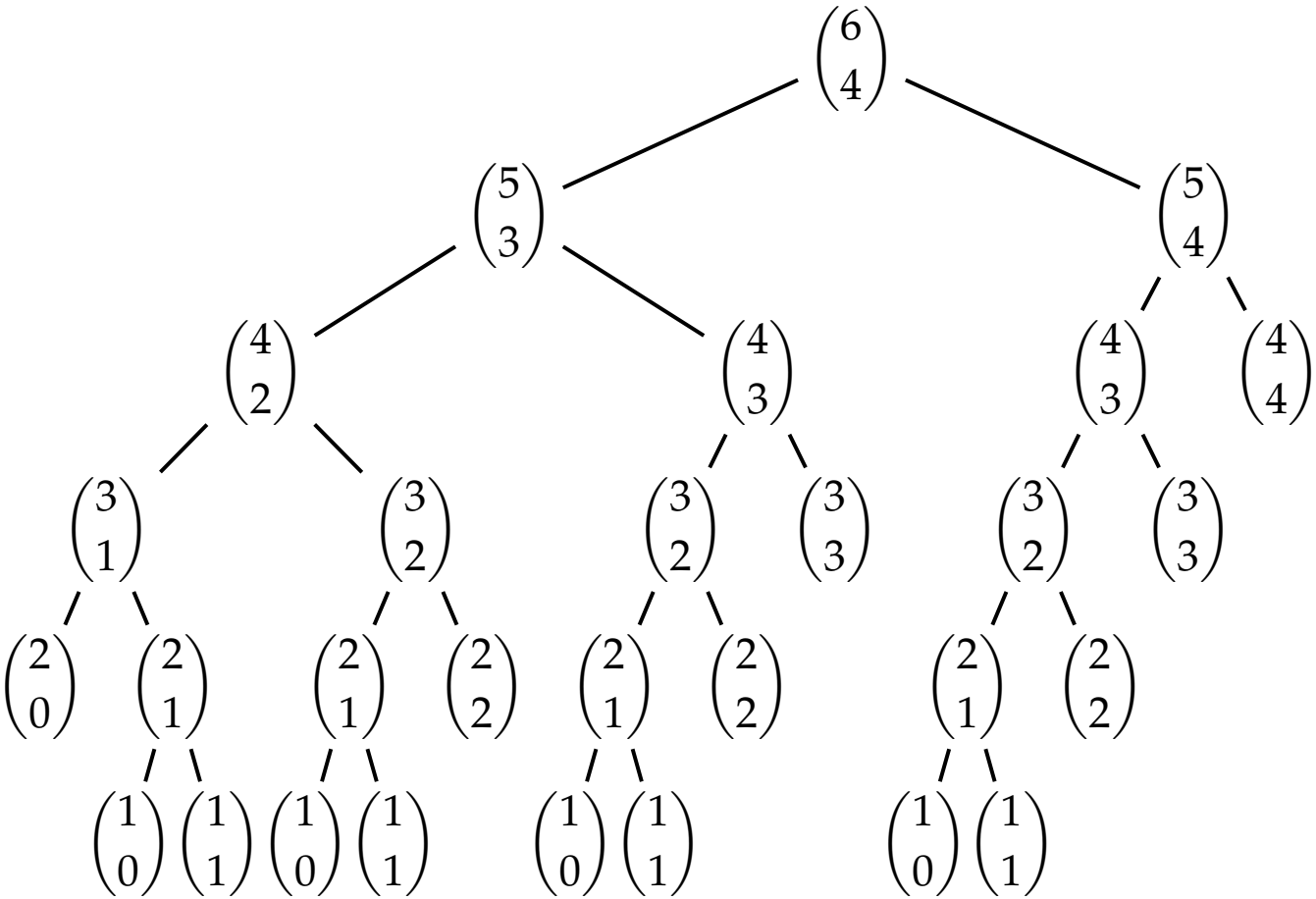
$k!$ y $n!$ solo pueden calcularse para valores muy pequeños.

- Tomar el primer elemento y $k-1$ elementos de entre los $n-1$ restantes.
- No tomar el primer elemento y k elementos de entre los $n-1$ restantes.

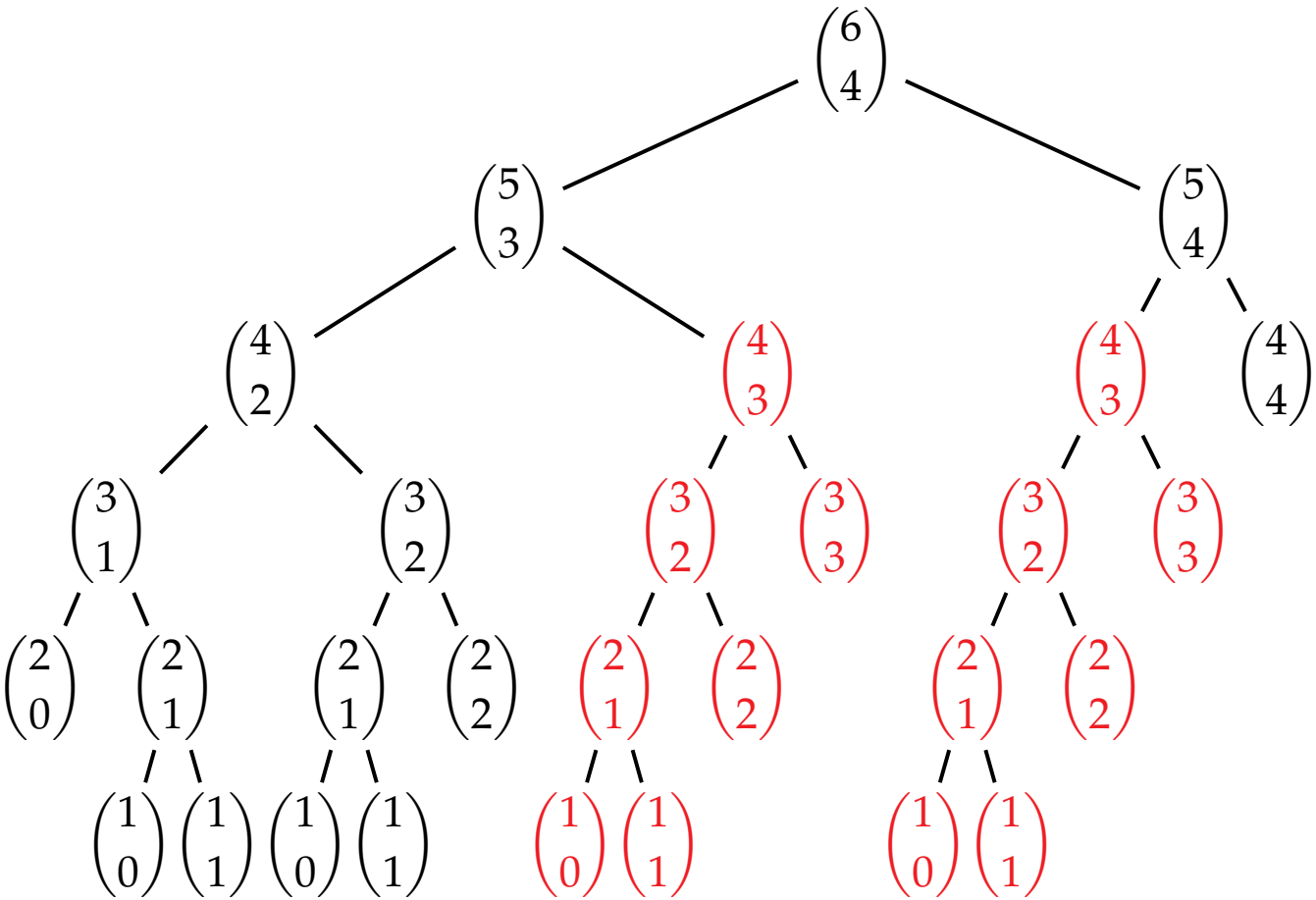
$$\binom{n}{k} = \begin{cases} 1 & \text{si } k = 0 \ \vee \ k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{si } 0 < k < n \end{cases}$$

Muy ineficiente. Calcula $2\binom{n}{k} - 1$ términos, pero muchos se calculan varias veces.

Coeficientes binomiales



Coeficientes binomiales



Coeficientes binomiales: Función con memoria

Añadir como parámetro una tabla con los valores ya calculados.

Tabla inicializada con valores no válidos, por ejemplo -1.

```
                    E          E/S                          S
proc coef-bin(n,k : nat, T  [0..n,0..k] de ent, cb: nat)
    si k = 0 ∨ k = n entonces cb := 1
    si no si T[n,k] ≠ −1 entonces cb := T[n,k]
          si no
              coef-bin(n − 1,k − 1,T,cb1)
              coef-bin(n − 1,k,T,cb2)
              cb := cb1 + cb2 ; T[n,k] := cb
          fsi
    fsi
fproc
```

$T[0..n,0..k] := [-1]$

coef-bin(n,k,T,cb)

Coste

Tiempo $O(nk) \approx O(n^2)$.

Espacio $O(nk)$

Versión iterativa: El triángulo de Pascal

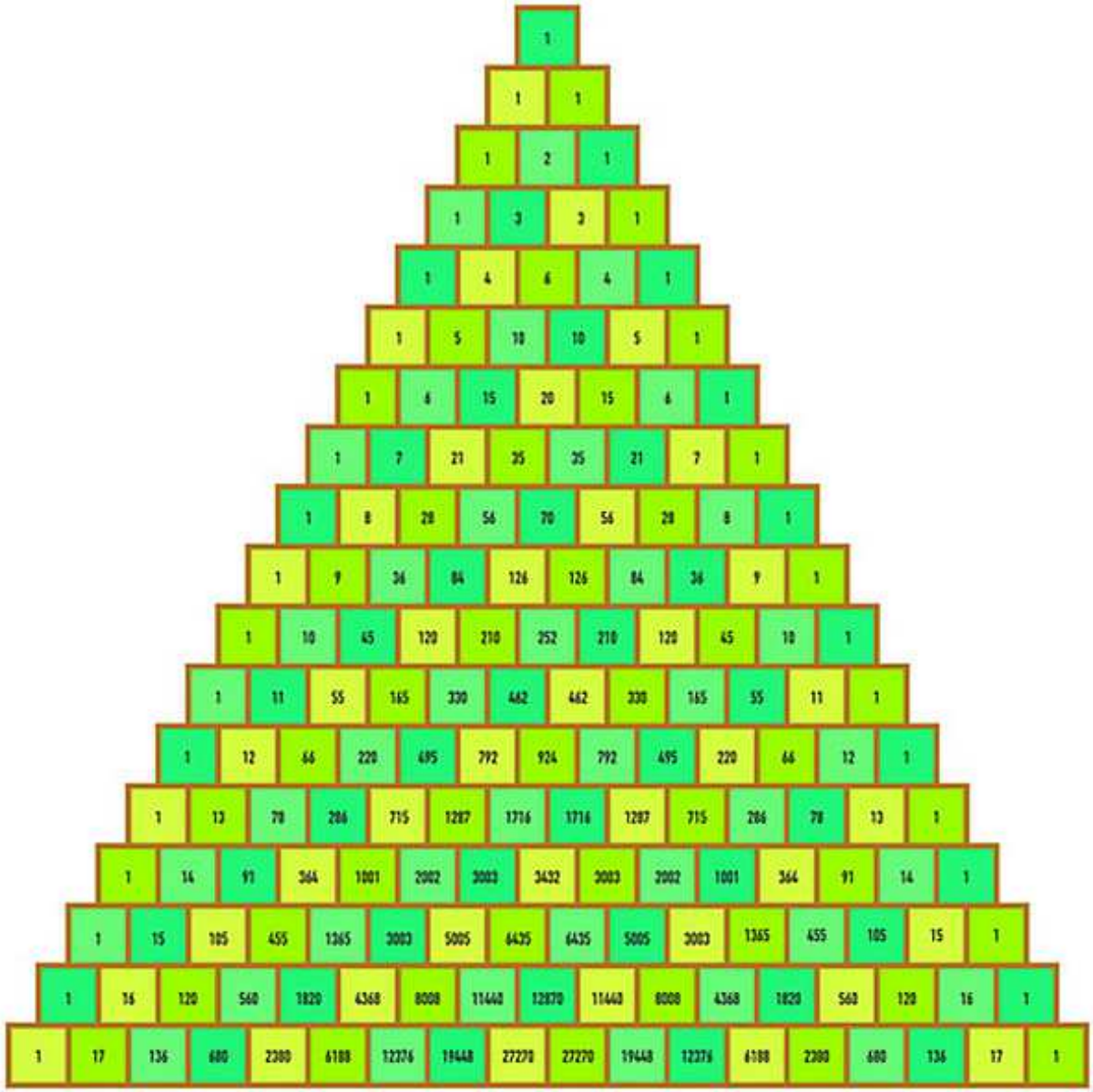
```
fun pascal(n,k : nat) dev cb : nat
var T[0..n,0..k] de nat
    T[0,0] := 1
    T[0,1..k] := [0]
    para i = 1 hasta n hacer
        T[i,0] := 1
        para j = 1 hasta k hacer
            T[i,j] := T[i-1,j-1] + T[i-1,j]
        fpara
    fpara
    cb := T[n,k]
ffun
```

	0	1	2	...	k
0	1	0	0	...	0
1	1	1	0	...	0
2	1	2	1	...	0
⋮	⋮	⋮	⋮	⋱	⋮
n	1	n	$\binom{n}{2}$	...	$\binom{n}{k}$

¿Cuántas casillas se rellenan?

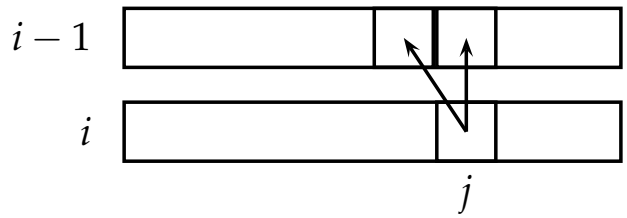
$(n-k+1)(k+1) = nk+n-k^2+1 \leq n(k+1)+1 \in O(nk)$

El triángulo de Pascal



El triángulo de Pascal: Mejora

Para calcular $T(i,j)$ se necesitan $T(i-1,j-1)$ y $T(i-1,j)$, en la fila anterior.



Reducir el espacio adicional a un vector que se rellena de derecha a izquierda.

```
fun pascal2(n,k : nat) dev cb : nat
var T[0..k] de nat
    T[0] := 1 ; T[1..k] := [0]
    para i = 1 hasta n hacer
        para j = k hasta 1 paso −1 hacer
            T[j] := T[j] + T[j − 1]
        fpara
    fpara
    cb := T[k]
ffun
```

Diseño descendente vs diseño ascendente

Divide y vencerás

Dividir en subproblemas; resolver (recursión); combinar.
Repeticiones si hay problemas solapados.

Programación dinámica

Resolver todos los subproblemas que se puedan necesitar (iterativo); combinarlos hasta llegar a resolver el problema original.
Usar una tabla para guardar los resultados.
$\Rightarrow$ Cada subproblema se resuelve una sola vez.

Esquema de programación dinámica

Identificación

1. Especificar la función que representa el problema a resolver.
2. Determinar las ecuaciones recurrentes para calcular dicha función.
3. Comprobar el alto coste de cálculo debido a la repetición de subproblemas a resolver.

Construcción

1. Sustituir la función por una tabla.
2. Inicializar la tabla según los casos base de la función.
3. Sustituir las llamadas recursivas por consultas a la tabla.
4. Planificar un orden adecuado para rellenar la tabla.

Problema de la mochila (versión entera)

Hay n objetos, cada uno con un peso (natural) $p_i > 0$ y un valor (real) $v_i > 0$.

La mochila soporta un peso total máximo $M > 0$.

Los objetos no se pueden fraccionar.

Maximizar

$$\sum_{i=1}^{n} x_i v_i$$

con la restricción

$$\sum_{i=1}^{n} x_i p_i \leq M,$$

donde $x_i \in \{0, 1\}$ indica si hemos cogido (1) o no (0) el objeto i.

La estrategia voraz ya no es correcta.

Problema de la mochila (versión entera)

Definimos una función

$$mochila(i,j) \quad = \quad \text{máximo valor que podemos poner en la mochila de peso máximo } j \text{ considerando los objetos del 1 al } i.$$

Definición recursiva

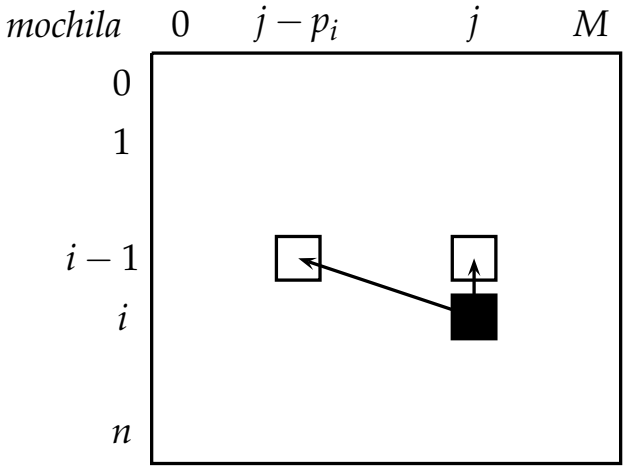
$$mochila(i,j) \quad = \quad \begin{cases} mochila(i-1,j) & \text{si } p_i > j \\ \text{máx}\{\underbrace{mochila(i-1,j)}_{\substack{\text{no cogemos} \\ \text{el objeto } i}}, \underbrace{mochila(i-1,j-p_i)+v_i}_{\substack{\text{sí cogemos} \\ \text{el objeto } i}}\} & \text{si } p_i \leq j \end{cases}$$

con $1 \leq i \leq n$ y $1 \leq j \leq M$.

Casos básicos:

$$\begin{array}{lcll} mochila(0,j) & = & 0 & 0 \leq j \leq M \\ mochila(i,0) & = & 0 & 0 \leq i \leq n. \end{array}$$

Problema de la mochila



Las comparaciones para llenar una posición siempre se refieren a posiciones de la fila anterior.

Recorrer la matriz por filas de arriba abajo y cada fila de izquierda a derecha.

Si solo se busca el valor máximo alcanzable, se puede optimizar el espacio adicional: utilizar un vector que se recorre de derecha a izquierda.

Si se desea obtener los objetos de la solución óptima no interesa optimizar, porque es necesario guardar las decisiones.

```
fun mochila-pd(P[1..n] de nat+, V[1..n] de real+, M : nat+)
    dev ⟨ valor : real, cuáles[1..n] de 0..1 ⟩
{ cuáles[i] indica si hemos cogido o no el objeto i }
var mochila[0..n, 0..M] de real
    { inicialización }
    mochila[0..n, 0] := [0] ; mochila[0, 1..M] := [0]
    { rellenar la matriz }
    para i = 1 hasta n hacer
        para j = 1 hasta M hacer
            si P[i] > j entonces mochila[i, j] := mochila[i − 1, j]
            si no mochila[i, j] := máx(mochila[i − 1, j], mochila[i − 1, j − P[i]] + V[i])
            fsi
        fpara
    fpara
    valor := mochila[n, M]
    { cálculo de los objetos }
    resto := M
    para i = n hasta 1 paso − 1 hacer
        si mochila[i, resto] = mochila[i − 1, resto] entonces { no coger objeto i }
            cuáles[i] := 0
        si no { sí coger objeto i }
            cuáles[i] := 1 ; resto := resto − P[i]
        fsi
    fpara
ffun
```

Problema de la mochila: Coste

$\Theta(nM)$ tanto en tiempo como en espacio adicional.

Aunque es lineal en n no es eficiente en todos los casos, pues depende de M que es un valor (no un tamaño) y no está relacionado con n.

$$t_M = \lceil \log M \rceil \Rightarrow M \in \Theta(2^{t_M}) \Rightarrow \Theta(nM) \approx \Theta(n2^{t_M})$$

Si M es demasiado grande con respecto a n, puede ser mejor calcular todas las posibilidades (exponencial).

Es un problema NP-difícil.

Pesos con valores reales: usar funciones escalonadas.

Problemas de optimización

Es necesaria una subestructura óptima para las soluciones.

Principio de optimalidad de Bellman

Toda solución óptima para una instancia de un problema contiene soluciones óptimas para todas las subinstancias.

En una secuencia óptima de decisiones, toda subsecuencia es óptima.

Problema de la mochila:

$X \subseteq \{1, \ldots, n\}$ solución óptima para n objetos y peso máximo M.

- $\forall i \in X$, $X - \{i\}$ es solución óptima para los n objetos menos el i-ésimo y peso máximo $M - p_i$.
- $\forall j \notin X$, X es solución óptima para los n objetos menos el j-ésimo y peso máximo M.

Caminos mínimos: Algoritmo de Floyd

Grafo $G = \langle N, A \rangle$ dirigido y valorado,
$N = \{1, 2, \ldots, n\}$, aristas de longitud positiva.

Matriz de costes $G[i,j] = \begin{cases} coste & \text{si } i \to j \in A \\ +\infty & \text{si } i \to j \notin A \end{cases}$

Coste de los caminos mínimos entre cada par de vértices del grafo.

$C^k(i,j)$ = mínimo coste para ir de i a j pudiendo utilizar como vértices intermedios aquellos entre 1 y k.

Definición recursiva

$$C^k(i,j) = \min\{C^{k-1}(i,j), C^{k-1}(i,k) + C^{k-1}(k,j)\}$$

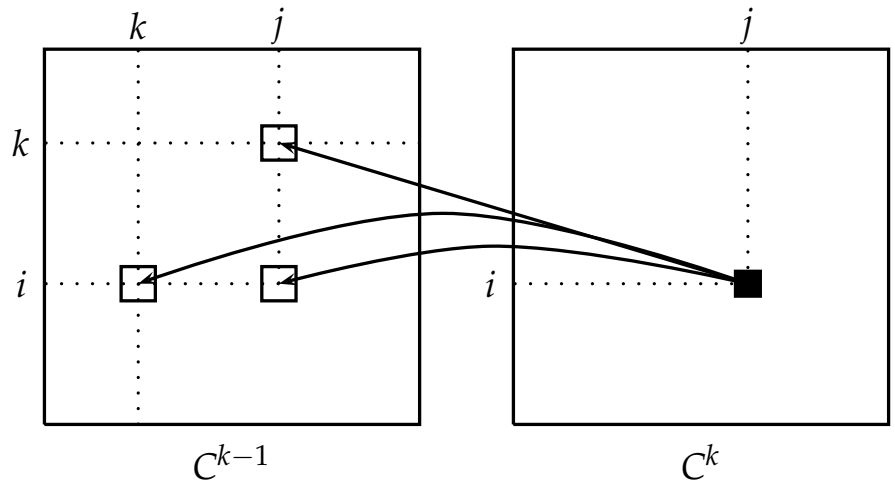
para $1 \leq k, i, j \leq n$.

Caso básico: $k = 0$,

$$C^0(i,j) = \begin{cases} G[i,j] & \text{si } i \neq j \\ 0 & \text{si } i = j \end{cases}$$

$C^n(i,j)$ = coste mínimo entre i y j.

Necesitamos $n+1$ matrices $n \times n \Rightarrow$ espacio adicional en $\Theta(n^3)$.



Para calcular C^k solo necesitamos C^{k-1}.

$$C^k(k,j) = \text{mín}\{C^{k-1}(k,j), C^{k-1}(k,k) + C^{k-1}(k,j)\} = C^{k-1}(k,j)$$

$$C^k(i,k) = \text{mín}\{C^{k-1}(i,k), C^{k-1}(i,k) + C^{k-1}(k,k)\} = C^{k-1}(i,k)$$

La fila k y la columna k no cambian cuando actualizamos de C^{k-1} a C^k.

Las actualizaciones se pueden ir realizando sobre la misma matriz: $C[1..n, 1..n]$.
Espacio adicional en $\Theta(1)$.

Algoritmo de Floyd: Implementación

```
fun Floyd(G : grafo-val[n]) dev ⟨ C[1..n,1..n] de real∞, camino[1..n,1..n] de 0..n ⟩
    { inicialización }
    C := G ;  camino[1..n,1..n] := [0]
    para i = 1 hasta n hacer C[i,i] := 0 fpara
    { actualizaciones de la matriz }
    para k = 1 hasta n hacer
        para i = 1 hasta n hacer
            para j = 1 hasta n hacer
                temp := C[i,k] + C[k,j]
                si temp < C[i,j] entonces
                    C[i,j] := temp
                    camino[i,j] := k
                fsi
            fpara
        fpara
    fpara
ffun
```

Coste

en tiempo $\Theta(n^3)$
en espacio $\Theta(1)$

Algoritmo de Floyd: Obtener caminos

```
proc imprimir-caminos(e C[1..n,1..n] de real_∞, e camino[1..n,1..n] de 0..n)
    para i = 1 hasta n hacer
        para j = 1 hasta n hacer
            si C[i,j] < +∞ entonces
                imprimir(camino de,i,a,j)
                imprimir(i)
                imp-camino-int(i,j,camino)
                imprimir(j)
            fsi
        fpara
    fpara
fproc

proc imp-camino-int(e i,j : 1..n, e camino[1..n,1..n] de 0..n)
    k := camino[i,j]
    si k > 0 entonces    { hay un camino no directo }
        imp-camino-int(i,k,camino)
        imprimir(k)
        imp-camino-int(k,j,camino)
    fsi
fproc
```

Cadena de productos de matrices

$A_{p\times q} \times B_{q\times r} = C_{p\times r}$, con

$$c_{ij} = \sum_{k=1}^{q} a_{ik} b_{kj}.$$

Coste: pqr multiplicaciones entre escalares.

Multiplicar una secuencia de matrices: $M_1 \times M_2 \times \cdots \times M_n$ $(M_i \rightsquigarrow d_{i-1} \times d_i)$.

Multiplicación de matrices: asociativa pero no conmutativa.
$\Rightarrow$ No se puede alterar el orden de las matrices, pero sí el de los productos.

Ejemplo: $A_{13\times 5}$, $B_{5\times 89}$, $C_{89\times 3}$, $D_{3\times 34}$

$$\underbrace{\underbrace{\underbrace{(\underbrace{(A \cdot B)}_{5785} \cdot C)}_{3471} \cdot D}_{1326}}_{} \quad \rightsquigarrow \quad 10582$$

$$\underbrace{\underbrace{\underbrace{(A \cdot \underbrace{(B \cdot C)}_{1335})}_{195} \cdot D}_{1326}}_{} \quad \rightsquigarrow \quad 2856$$

¿Cómo insertar paréntesis en la secuencia para minimizar el número total de multiplicaciones entre escalares?

Fuerza bruta: Probar todas las posibilidades.

$$\underbrace{(M_1 \cdot \ldots \cdot M_k)}_{X(k) \text{ formas}} \cdot \underbrace{(M_{k+1} \cdot \ldots \cdot M_n)}_{X(n-k) \text{ formas}}$$

$$X(1) = X(2) = 1,$$
$$X(n) = \sum_{k=1}^{n-1} X(k) \,.\, X(n-k),\ n > 2.$$

Números de Catalan: $X(n) = \frac{1}{n}\binom{2(n-1)}{n-1} \geq 2^{n-2}$.

Función recursiva:

$matrices(i,j)$ = mínimo número de multiplicaciones escalares para realizar el producto matricial $M_i \cdots M_j$, $(i \leq j)$.

$$matrices(i,i) = 0,$$
$$matrices(i,j) = \min_{i \leq k \leq j-1} \{matrices(i,k) + matrices(k+1,j) + d_{i-1} d_k d_j\},\ i < j$$

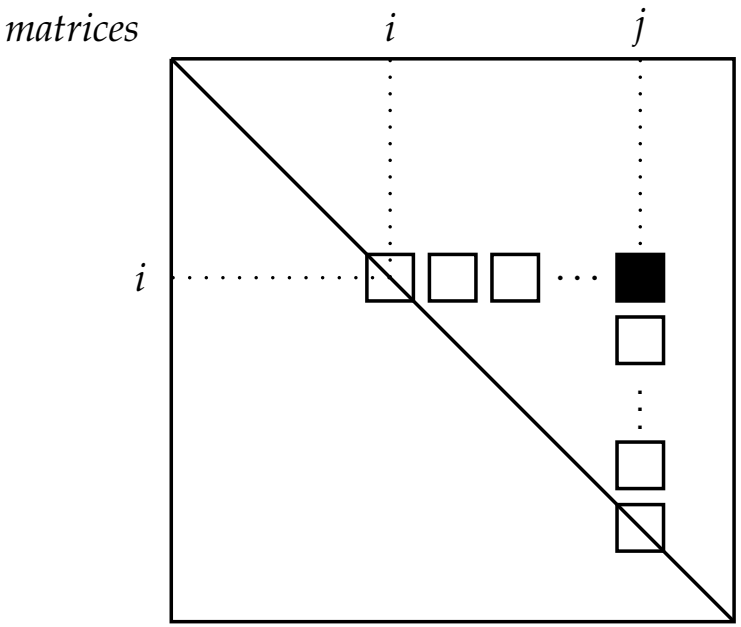
$$T(n) = 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k)) = 1 + 2\sum_{k=1}^{n-1} T(k) \in \Theta(3^n)$$

Cadena de productos de matrices

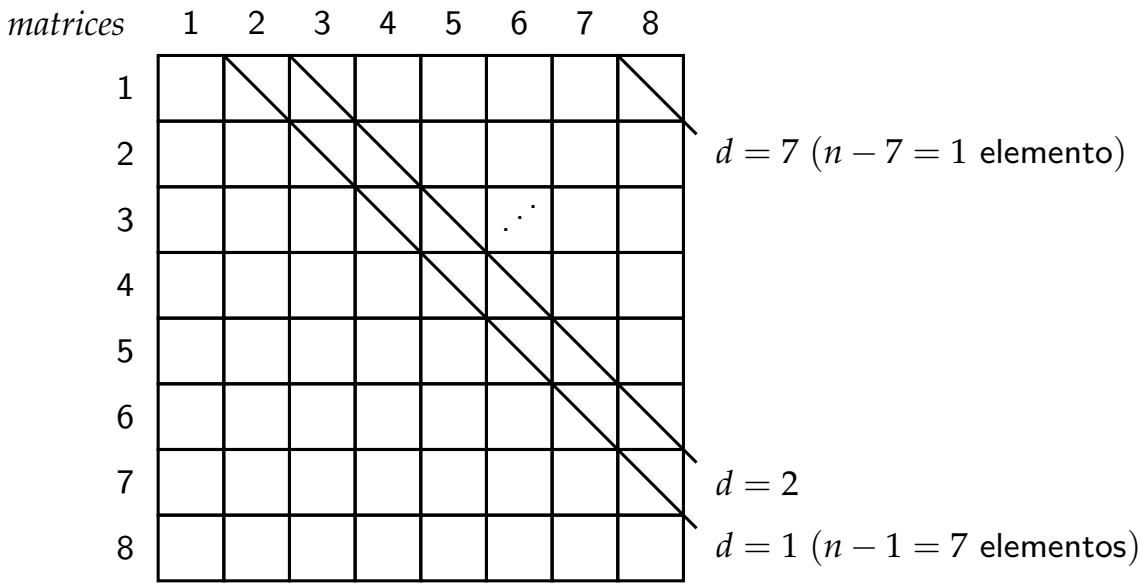
Se repiten demasiadas veces los subproblemas. Utilizar Programación dinámica.

Se cumple el principio de optimalidad: $\underbrace{(M_1 \cdot \ldots \cdot M_k)}_{\text{óptimo}} \cdot \underbrace{(M_{k+1} \cdot \ldots \cdot M_n)}_{\text{óptimo}}$

Tabla $matrices[1..n, 1..n]$, solo usamos la mitad superior a la diagonal principal.



Rellenar la matriz recorriéndola por diagonales.



- Numerar las diagonales desde $d = 1$ hasta $d = n - 1$ en el orden en el que tienen que recorrerse.
- Cada diagonal tiene $n - d$ elementos (numerados del $i = 1$ al $i = n - d$).
- Este índice sirve para conocer directamente la fila en la que se encuentra el elemento a calcular.
- La columna se puede calcular mediante $j = i + d$.

Cadena de productos de matrices: Implementación

```
fun multiplica-matrices(D[0..n] de nat+) dev ⟨núm-mín : nat, P[1..n,1..n] de 0..n⟩
var matrices[1..n,1..n] de nat∞
    para i = 1 hasta n hacer   { inicialización, diagonal principal }
        matrices[i,i] := 0 ;  P[i,i] := 0
    fpara
    { recorrido por diagonales }
    para d = 1 hasta n − 1 hacer { recorre diagonales }
        para i = 1 hasta n − d hacer { recorre elementos dentro de la diagonal }
            j := i + d
            { calcular mínimo }
            matrices[i,j] := +∞
            para k = i hasta j − 1 hacer
                temp := matrices[i,k] + matrices[k + 1,j] + D[i − 1] ∗ D[k] ∗ D[j]
                si temp < matrices[i,j] entonces
                    matrices[i,j] := temp ;  P[i,j] := k
                fsi
            fpara
        fpara
    fpara
    núm-mín := matrices[1,n]
ffun
```

Coste

en tiempo $\Theta(n^3)$
en espacio $\Theta(n^2)$

Cadena de productos de matrices: Imprimir orden

$\{ 1 \leq i \leq j \leq n \}$

```
proc escribir-paréntesis(e i,j : nat, e P[1..n,1..n] de nat)
    si i = j entonces imprimir("M_i")
    si no
        k := P[i,j]
        si k > i entonces
            imprimir(()
            escribir-paréntesis(i,k,P)
            imprimir())
        si no
            imprimir("M_i")
        fsi
        imprimir(*)
        si k+1 < j entonces
            imprimir(()
            escribir-paréntesis(k+1,j,P)
            imprimir())
        si no
            imprimir("M_j")
        fsi
    fsi
fproc
```

Llamada inicial

`escribir-paréntesis`$(1,n,P)$

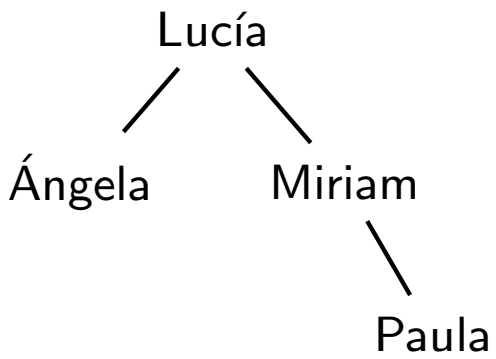
Coste en tiempo $\Theta(n)$

Árboles binarios de búsqueda óptimos

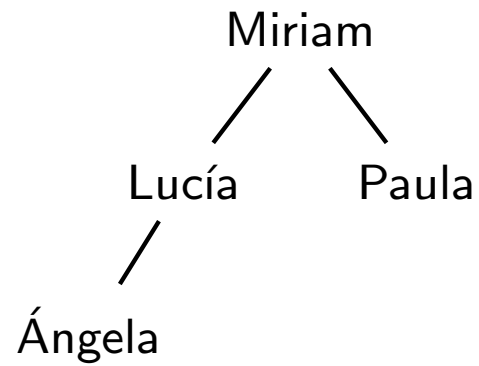
Sean $c_1 < c_2 < \cdots < c_n$ un conjunto de claves distintas ordenadas, y sea p_i la probabilidad con que se pide buscar la clave c_i y su información asociada.
Se tiene $\sum_{i=1}^{n} p_i = 1$.

Encontrar un árbol de búsqueda que minimice el número medio de comparaciones necesarias para realizar una búsqueda, suponiendo que todas las peticiones se refieren a claves que están en el árbol.

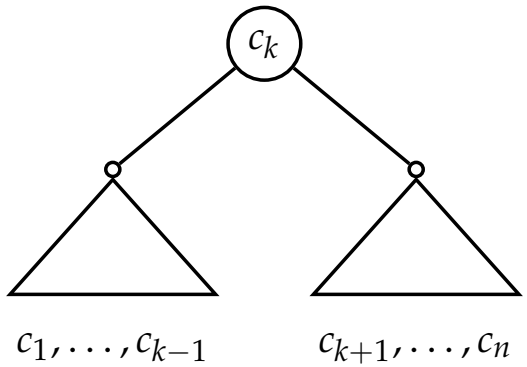
Ejemplo: $c_1 =$ Ángela, $p_1 = \frac{3}{8}$, $c_2 =$ Lucía, $p_2 = \frac{3}{8}$, $c_3 =$ Miriam, $p_3 = \frac{1}{8}$, $c_4 =$ Paula $p_4 = \frac{1}{8}$



$\frac{7}{4}$



$\frac{9}{4}$



Árbol de búsqueda a para las claves $\{c_1, \ldots, c_n\}$ con la clave c_k en la raíz.

$comp_a(1,n)$ = número medio de comparaciones para realizar una búsqueda en a en las condiciones del enunciado.

$$comp_a(1,n) = \sum_{l=1}^{n} p_l \, nivel_a(c_l) = p_k + \sum_{l=1}^{k-1} p_l \, nivel_a(c_l) + \sum_{l=k+1}^{n} p_l \, nivel_a(c_l)$$

$$= p_k + \sum_{l=1}^{k-1} p_l \, (nivel_{hi(a)}(c_l) + 1) + \sum_{l=k+1}^{n} p_l \, (nivel_{hd(a)}(c_l) + 1)$$

$$= p_k + \sum_{l=1}^{k-1} p_l + \sum_{l=k+1}^{n} p_l + \sum_{l=1}^{k-1} p_l \, nivel_{hi(a)}(c_l) + \sum_{l=k+1}^{n} p_l \, nivel_{hd(a)}(c_l)$$

$$= \sum_{l=1}^{n} p_l + comp_{hi(a)}(1, k-1) + comp_{hd(a)}(k+1, n)$$

Árboles binarios de búsqueda óptimos

$comp(i,j)$ = número medio mínimo de comparaciones en un árbol de búsqueda conteniendo las claves $c_{i+1}, \ldots, c_j$.

$$comp(i,i) = 0$$

para $0 \leq i \leq n$, que corresponde al árbol vacío;

$$comp(i,j) = \sum_{l=i+1}^{j} p_l + \min_{i+1 \leq k \leq j} \{comp(i,k-1) + comp(k,j)\}$$

para $0 \leq i < j \leq n$.

Se cumple el principio de optimalidad.

El caso que nos interesa, para un árbol con n claves, será $comp(0,n)$.

Árboles binarios de búsqueda óptimos

Recorrido por diagonales.

Matriz adicional $prob[0..n, 0..n]$:

$$prob[i,j] = \sum_{l=i+1}^{j} p_l,$$

se puede calcular la matriz mediante la fórmula

$$prob[i,j] = prob[i,j-1] + p_j,$$

caso básico: $prob[i,i] = 0$.

Guardamos en una tercera matriz $raíz[0..n, 0..n]$ las decisiones sobre las raíces de los árboles óptimos:

$raíz[i,j]$ = la raíz del árbol de búsqueda óptimo con las claves $c_{i+1}, \ldots, c_j$.

Árboles binarios de búsqueda óptimos: Implementación

$\{\ C[1] < \cdots < C[n] \ \wedge\ \forall i : 1 \leq i \leq n : 0 \leq P[i] \leq 1\ \}$
fun árbol-búsqueda-óptimo$(C[1..n]$ **de** *clave*$, P[1..n]$ **de** *real*$)$
  dev $\langle$ *núm-comp* : *real*$, raíz[0..n, 0..n]$ **de** $0..n\ \rangle$
var $comp[0..n, 0..n]$ **de** $real_\infty, prob[0..n, 0..n]$ **de** *real*
  para $i = 0$ **hasta** n **hacer**
    $comp[i,i] := 0$; $prob[i,i] := 0$; $raíz[i,i] := 0$
  fpara
  para $d = 1$ **hasta** n **hacer** { recorre diagonales }
    para $i = 0$ **hasta** $n - d$ **hacer** { recorre elementos dentro de la diagonal }
      $j := i + d$
      $prob[i,j] := prob[i, j-1] + P[j]$
      { calcular mínimo }
      $mínimo := +\infty$
      para $k = i + 1$ **hasta** j **hacer**
        $temp := comp[i, k-1] + comp[k, j]$
        si $temp < mínimo$ **entonces**
          $mínimo := temp$; $raíz[i,j] := k$
        fsi
      fpara
      $comp[i,j] := mínimo + prob[i,j]$
    fpara
  fpara
  $núm\text{-}comp := comp[0, n]$
ffun

Coste

en tiempo $\Theta(n^3)$
en espacio $\Theta(n^2)$

Árboles binarios de búsqueda óptimos: Construir árbol

$\{\ C[1] < \cdots < C[n] \ \wedge\ 0 \leq i \leq j \leq n\ \}$

fun `construir-árbol`$(C[1..n]$ **de** *clave*, $raíz[0..n, 0..n]$ **de** $0..n, i, j : 0..n)$
  dev *árbol* : *árbol-bb[clave]*
var iz, dr : *árbol-bb[clave]*
  si $raíz[i,j] = 0$ **entonces** *árbol* := `abb-vacío`$()$
  si no
    $k := raíz[i,j]$
    $iz :=$ `construir-árbol`$(C, raíz, i, k-1)$
    $dr :=$ `construir-árbol`$(C, raíz, k, j)$
    árbol := `plantar`$(iz, C[k], dr)$
  fsi
ffun

Llamada inicial `construir-árbol`$(C, raíz, 0, n)$.

Coste en tiempo $\Theta(n)$