Programación dinámica

Alberto Verdejo

Dpto. de Sistemas Informáticos y Computación

Universidad Complutense de Madrid

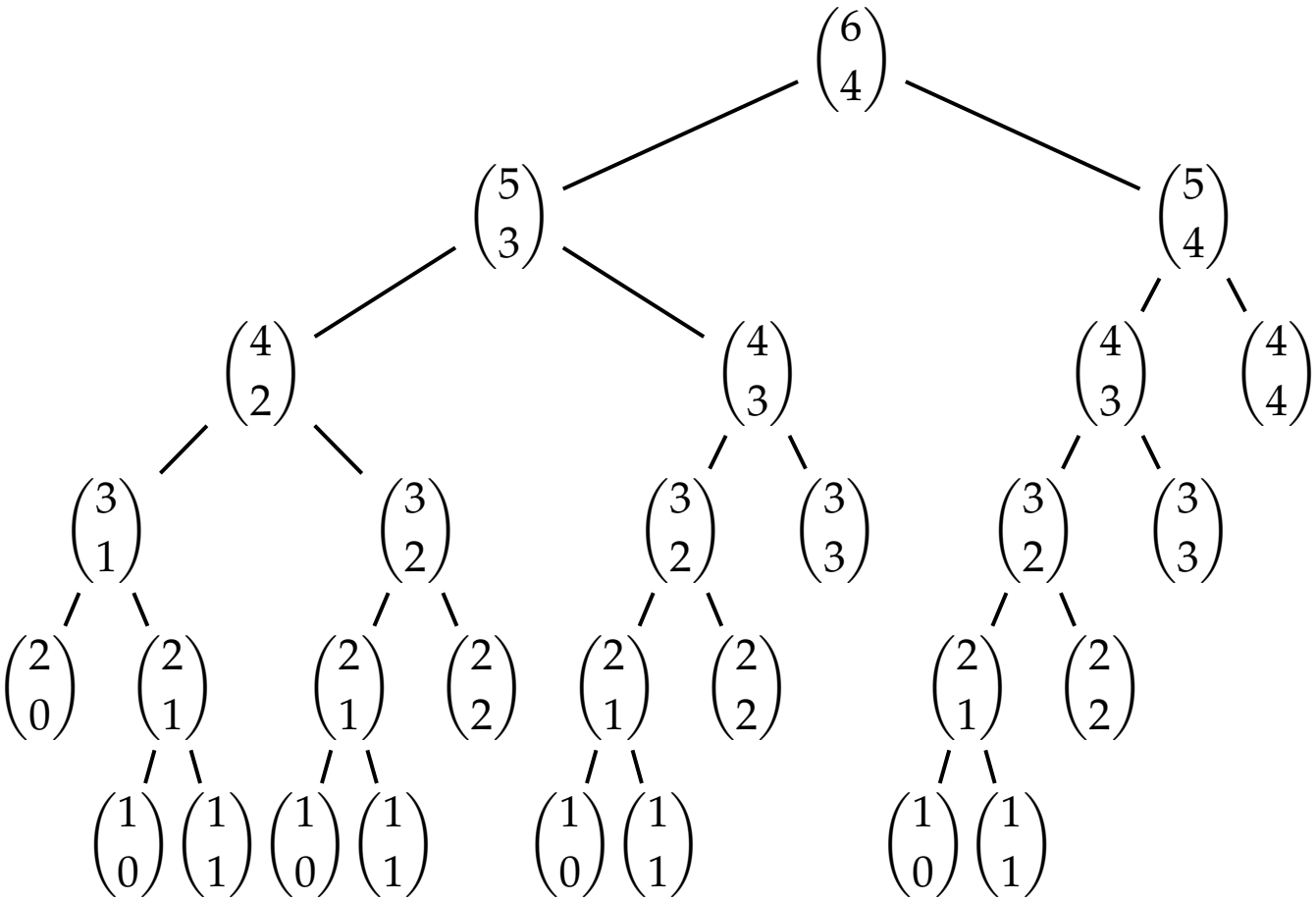
Noviembre 2008

Bibliografía

- R. Neapolitan y K. Naimipour. *Foundations of Algorithms using C++ pseudocode*. Tercera edición. Jones and Bartlett Publishers, 2004.
  Capítulo 3
- E. Horowitz, S. Sahni y S. Rajasekaran. *Computer Algorithms*. Tercera edición. Computer Science Press, 1998.
  Capítulo 5
- N. Martí Oliet, Y. Ortega Mallén y J. A. Verdejo López. *Estructuras de datos y métodos algorítmicos: ejercicios resueltos*. Colección Prentice Practica, Pearson/Prentice Hall, 2003.
  Capítulo 13

Motivación

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



Funciones con memoria

Añadir a la función un parámetro que es una tabla con los valores ya calculados.

La tabla está inicializada con valores diferentes, por ejemplo -1.

proc `núm-comb`$(n, r : nat, C[0..n, 0..r]$ **de** $ent, nc : nat)$
  si $r = 0 \vee r = n$ **entonces** $nc := 1$
  si no si $C[n, r] \neq -1$ **entonces** $nc := C[n, r]$
    si no
      `núm-comb`$(n-1, r-1, C, nc1)$
      `núm-comb`$(n-1, r, C, nc2)$
      $nc := nc1 + nc2$; $C[n, r] := nc$
    fsi
  fsi
fproc

$C[0..n, 0..r] := [-1]$
`núm-comb`(n, r, C, nc)

Inicialización con tiempo en $\Theta(1)$.

Método ascendente

Comenzar por resolver todos los subproblemas más pequeños que se puedan necesitar, para ir combinándolos hasta llegar a resolver el problema original.

Método ascendente

Comenzar por resolver todos los subproblemas más pequeños que se puedan necesitar, para ir combinándolos hasta llegar a resolver el problema original.

$$\begin{array}{ccccccc}
 & & & \binom{0}{0} & & & \\
 & & \binom{1}{0} & & \binom{1}{1} & & \\
 & \binom{2}{0} & & \binom{2}{1} & & \binom{2}{2} & \\
\binom{3}{0} & & \binom{3}{1} & & \binom{3}{2} & & \binom{3}{3} \\
\ddots & & \vdots & & & \ddots &
\end{array}$$

Método ascendente

Comenzar por resolver todos los subproblemas más pequeños que se puedan necesitar, para ir combinándolos hasta llegar a resolver el problema original.

$$\begin{array}{ccccccc} & & & \binom{0}{0} & & & \\ & & \binom{1}{0} & & \binom{1}{1} & & \\ & \binom{2}{0} & & \binom{2}{1} & & \binom{2}{2} & \\ \binom{3}{0} & & \binom{3}{1} & & \binom{3}{2} & & \binom{3}{3} \\ \ddots & & \vdots & & \ddots & & \end{array}$$

La base de la programación dinámica es el uso de una tabla para ir almacenando los resultados correspondientes a instancias más sencillas del problema a resolver.

Esquema de programación dinámica

Identificación

- Especificación de la función que representa el problema a resolver.
- Determinación de las ecuaciones recurrentes para calcular dicha función.
- Comprobación del alto coste de cálculo de dicha función debido a la repetición de subproblemas a resolver.

Construcción

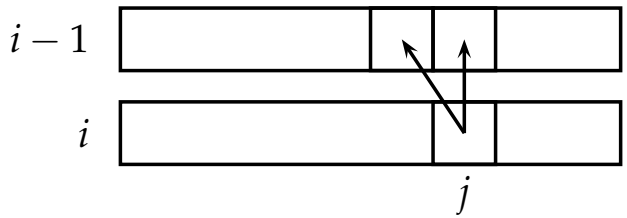
- Sustitución de la función por una tabla.
- Inicialización de la tabla según los casos base de la definición recursiva de la función.
- Sustitución, en las ecuaciones, de las llamadas recursivas por consultas a la tabla.
- Planificación del orden de llenado de la tabla, de forma que se respeten las necesidades de cada entrada de la tabla.

Números combinatorios

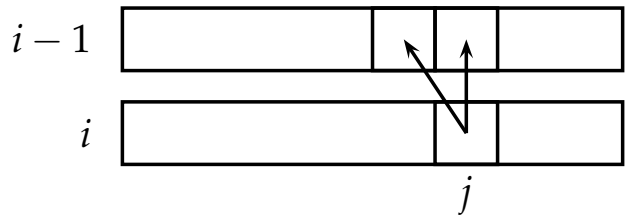
	0	1	2	...	r
0	1	0	0	...	0
1	1	1	0	...	0
2	1	2	1	...	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
n	1	n	$\binom{n}{2}$	...	$\binom{n}{r}$

fun pascal$(n, r : nat)$ **dev** $c : nat$
var $C[0..n, 0..r]$ **de** nat
  $C[0,0] := 1$
  $C[0, 1..r] := [0]$
  para $i = 1$ **hasta** n **hacer**
    $C[i, 0] := 1$
    para $j = 1$ **hasta** r **hacer**
      $C[i, j] := C[i-1, j-1] + C[i-1, j]$
    fpara
  fpara
  $c := C[n, r]$
ffun

Dejando aparte los casos básicos, para calcular cada entrada (i,j) en la tabla se necesitan las entradas $(i-1,j-1)$ e $(i-1,j)$ de la fila anterior, por lo que el espacio adicional se puede reducir a un vector que se rellena de derecha a izquierda.



Dejando aparte los casos básicos, para calcular cada entrada (i,j) en la tabla se necesitan las entradas $(i-1,j-1)$ e $(i-1,j)$ de la fila anterior, por lo que el espacio adicional se puede reducir a un vector que se rellena de derecha a izquierda.



fun pascal2$(n,r : nat)$ **dev** $c : nat$
var $C[0..r]$ **de** *nat*
    $C[0] := 1$; $C[1..r] := [0]$
    para $i = 1$ **hasta** n **hacer**
        para $k = r$ **hasta** 1 **paso** -1 **hacer**
            $C[k] := C[k] + C[k-1]$
        fpara
    fpara
    $c := C[r]$
ffun

El campeonato mundial

- Competición entre dos equipos, A y B, en la que el ganador es el primer equipo que consiga n victorias.
- No hay empates, los resultados de todos los partidos son independientes, y para cualquier partido dado hay una probabilidad constante p de que lo gane el equipo A (y por tanto una probabilidad constante $q = 1 - p$ de que lo gane el equipo B).
- Queremos calcular la probabilidad que tiene el equipo A de ganar la competición, antes del primer partido.

El campeonato mundial

- Competición entre dos equipos, A y B, en la que el ganador es el primer equipo que consiga n victorias.
- No hay empates, los resultados de todos los partidos son independientes, y para cualquier partido dado hay una probabilidad constante p de que lo gane el equipo A (y por tanto una probabilidad constante $q = 1 - p$ de que lo gane el equipo B).
- Queremos calcular la probabilidad que tiene el equipo A de ganar la competición, antes del primer partido.

Definimos la función

$P(i,j)$ = probabilidad de que A gane la competición si a A le faltan i victorias para ganar y a B le faltan j victorias.

El campeonato mundial

- Competición entre dos equipos, A y B, en la que el ganador es el primer equipo que consiga n victorias.
- No hay empates, los resultados de todos los partidos son independientes, y para cualquier partido dado hay una probabilidad constante p de que lo gane el equipo A (y por tanto una probabilidad constante $q = 1 - p$ de que lo gane el equipo B).
- Queremos calcular la probabilidad que tiene el equipo A de ganar la competición, antes del primer partido.

Definimos la función

$P(i,j)$ = probabilidad de que A gane la competición si a A le faltan i victorias para ganar y a B le faltan j victorias.

El valor que nos interesa calcular es $P(n,n)$.

Definición recursiva:

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

El campeonato mundial

- Competición entre dos equipos, A y B, en la que el ganador es el primer equipo que consiga n victorias.
- No hay empates, los resultados de todos los partidos son independientes, y para cualquier partido dado hay una probabilidad constante p de que lo gane el equipo A (y por tanto una probabilidad constante $q = 1 - p$ de que lo gane el equipo B).
- Queremos calcular la probabilidad que tiene el equipo A de ganar la competición, antes del primer partido.

Definimos la función

$P(i,j)$ = probabilidad de que A gane la competición si a A le faltan i victorias para ganar y a B le faltan j victorias.

El valor que nos interesa calcular es $P(n,n)$.

Definición recursiva:

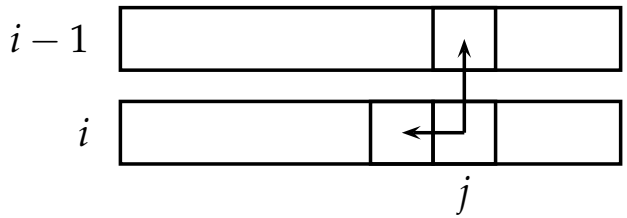
$$P(i,j) = pP(i-1,j) + qP(i,j-1)$$

Los casos básicos son

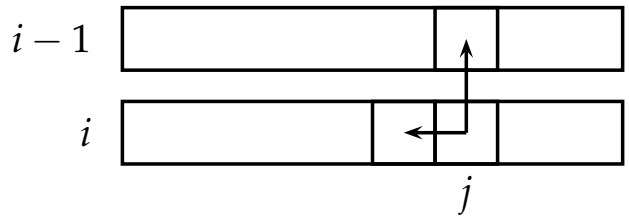
$$\begin{aligned} P(0,j) &= 1 \qquad 1 \leq j \leq n \\ P(i,0) &= 0 \qquad 1 \leq i \leq n \end{aligned}$$

```
fun competición(n : nat+, p : real) dev probabilidad-A : real    { Θ(n²) }
var P[0..n, 0..n] : real
    P[0, 1..n] := [1]
    P[1..n, 0] := [0]
    para i = 1 hasta n hacer
        para j = 1 hasta n hacer
            P[i, j] := p * P[i − 1, j] + (1 − p) * P[i, j − 1]
        fpara
    fpara
    probabilidad-A := P[n]
ffun
```

¿Podemos reducir el espacio adicional utilizado?



¿Podemos reducir el espacio adicional utilizado?



Rellenar de izquierda a derecha.

fun competición2$(n : nat^+, p : real)$ **dev** *probabilidad-A* : *real* $\{\ \Theta(n)\ \}$
var $P[0..n] : real$
$P[0] := 0$
$P[1..n] := [1]$
para $i = 1$ **hasta** n **hacer**
para $j = 1$ **hasta** n **hacer**
$P[j] := p * P[j] + (1 - p) * P[j-1]$
fpara
fpara
$probabilidad\text{-}A := P[n]$
ffun

El campeonato mundial, generalización

¿Qué ocurre cuando existe una probabilidad p de que A gane un partido, q de que lo pierda, y r (con $p+q+r=1$) de que haya empate?

Un empate no supone una victoria para ningún equipo y siguen siendo necesarias n victorias para ganar la competición.

El campeonato mundial, generalización

¿Qué ocurre cuando existe una probabilidad p de que A gane un partido, q de que lo pierda, y r (con $p+q+r=1$) de que haya empate?

Un empate no supone una victoria para ningún equipo y siguen siendo necesarias n victorias para ganar la competición.

Hay que modificar la fórmula anterior como sigue:

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

donde el tercer caso indica que al haber empatado, tanto a A como a B les faltan las mismas victorias que antes de jugar ese partido.

El campeonato mundial, generalización

¿Qué ocurre cuando existe una probabilidad p de que A gane un partido, q de que lo pierda, y r (con $p+q+r=1$) de que haya empate?

Un empate no supone una victoria para ningún equipo y siguen siendo necesarias n victorias para ganar la competición.

Hay que modificar la fórmula anterior como sigue:

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

donde el tercer caso indica que al haber empatado, tanto a A como a B les faltan las mismas victorias que antes de jugar ese partido.

Podemos despejar $P(i,j)$ obteniendo

$$P(i,j) = \frac{p}{1-r}P(i-1,j) + \frac{q}{1-r}P(i,j-1),$$

que se puede resolver exactamente de la misma forma que en el caso anterior.

Problema del cambio

- Se dispone de un conjunto finito $M = \{m_1, m_2, \ldots, m_n\}$ de tipos de monedas, donde cada m_i es un número natural.
- Existe una cantidad ilimitada de monedas de cada valor.
- Se quiere pagar una cantidad $C > 0$ utilizando el menor número posible de monedas.

Problema del cambio

- Se dispone de un conjunto finito $M = \{m_1, m_2, \ldots, m_n\}$ de tipos de monedas, donde cada m_i es un número natural.
- Existe una cantidad ilimitada de monedas de cada valor.
- Se quiere pagar una cantidad $C > 0$ utilizando el menor número posible de monedas.

$monedas(n, C)$ = número *mínimo* de monedas para pagar la cantidad C considerando los tipos de monedas del 1 al n.

Problema del cambio

- Se dispone de un conjunto finito $M = \{m_1, m_2, \ldots, m_n\}$ de tipos de monedas, donde cada m_i es un número natural.
- Existe una cantidad ilimitada de monedas de cada valor.
- Se quiere pagar una cantidad $C > 0$ utilizando el menor número posible de monedas.

$$monedas(n, C) \quad = \quad \text{número } \textit{mínimo} \text{ de monedas para pagar la cantidad } C \text{ considerando los tipos de monedas del } 1 \text{ al } n.$$

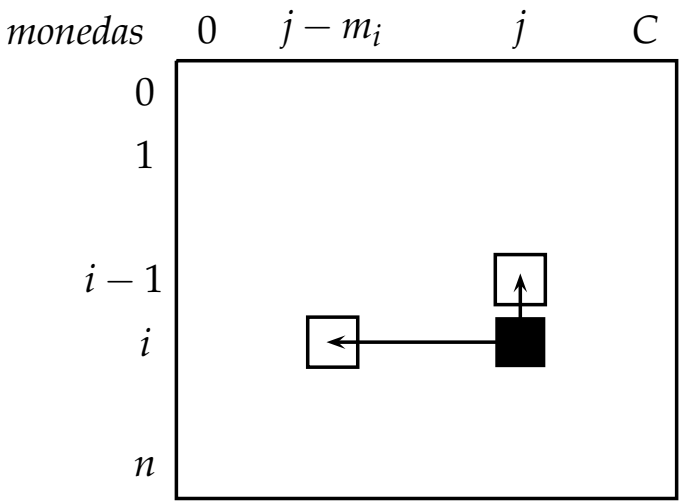
Definición recursiva:

$$monedas(i,j) \quad = \quad \begin{cases} monedas(i-1,j) & \text{si } m_i > j \\ \text{mín}\{monedas(i-1,j), monedas(i,j-m_i)+1\} & \text{si } m_i \leq j \end{cases}$$

donde $1 \leq i \leq n$ y $1 \leq j \leq C$.

Casos básicos:

$$\begin{array}{lcll} monedas(i,0) & = & 0 & 0 \leq i \leq n \\ monedas(0,j) & = & +\infty & 1 \leq j \leq C \end{array}$$



fun `devolver-cambio1`$(M[1..n]$ **de** $nat^+, C : nat)$ **dev** $número : nat_\infty$
{ $número$ es la cantidad de monedas en la solución óptima }
{ $número$ es $+\infty$ cuando no existe solución }
var $monedas[0..n, 0..C]$ **de** nat_∞
    { inicialización }
    $monedas[0, 1..C] := [+\infty]$
    $monedas[0..n, 0] := [0]$
    { rellenar la matriz }
    para $i = 1$ **hasta** n **hacer**
        para $j = 1$ **hasta** C **hacer**
            si $M[i] > j$ **entonces** $monedas[i, j] := monedas[i-1, j]$
            si no $monedas[i, j] :=$ `mín`$(monedas[i-1, j], monedas[i, j-M[i]]+1)$
            fsi
        fpara
    fpara
    $número := monedas[n, C]$
ffun

Coste: $\Theta(nC)$ tanto en tiempo como en espacio adicional.

Problema del cambio: Ejemplo

$C = 8$, $n = 3$, $m_1 = 1$, $m_2 = 4$ y $m_3 = 6$.

	0	1	2	3	4	5	6	7	8
0	0	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
1	0	1	2	3	4	5	6	7	8
2	0	1	2	3	1	2	3	4	2
3	0	1	2	3	1	2	1	2	2

Problema del cambio: cálculo de la solución óptima

Además del número total de monedas en la solución óptima, queremos conocer también dicha solución, es decir, cuántas monedas de cada tipo la forman.

$$monedas(i,j) = \text{mín}\{\underbrace{monedas(i-1,j)}_{\substack{\text{no cogemos} \\ \text{moneda } m_i}}, \underbrace{monedas(i,j-m_i)+1}_{\substack{\text{sí cogemos} \\ \text{moneda } m_i}}\}$$

Problema del cambio: cálculo de la solución óptima

Además del número total de monedas en la solución óptima, queremos conocer también dicha solución, es decir, cuántas monedas de cada tipo la forman.

$$monedas(i,j) = \text{mín}\{\underbrace{monedas(i-1,j)}_{\substack{\text{no cogemos}\\ \text{moneda } m_i}}, \underbrace{monedas(i,j-m_i)+1}_{\substack{\text{sí cogemos}\\ \text{moneda } m_i}}\}$$

Por tanto, sabemos que si

$$monedas(i,j) = monedas(i-1,j)$$

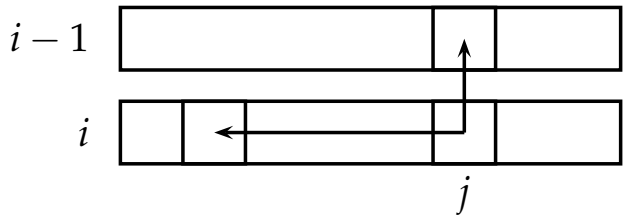
es porque podemos no coger ninguna moneda de tipo i para pagar la cantidad j, mientras que si

$$monedas(i,j) \neq monedas(i-1,j)$$

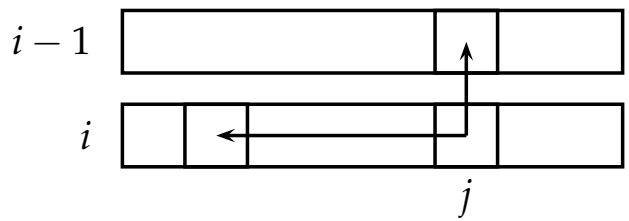
debemos coger al menos una moneda de tipo i para pagar la cantidad j.

$\{\ monedas[n,C] \neq +\infty\ \}$
fun `calcular-monedas1`$(M[1..n]$ **de** $nat^+, monedas[0..n,0..C]$ **de** $nat_\infty)$ $\quad \{\ \Theta(C)\ \}$
dev $cuántas[1..n]$ **de** nat
$\{\ cuántas[i]$ es el número de monedas de tipo i cogidas $\}$
$cuántas[1..n] := [0]$
$i := n\ ;\ \ j := C$
mientras $j > 0$ **hacer** $\{$ no hemos pagado todo $\}$
si $M[i] \leq j \ \wedge\ monedas[i,j] \neq monedas[i-1,j]$ **entonces**
$\{$ tomamos una moneda de tipo i $\}$
$cuántas[i] := cuántas[i] + 1$
$j := j - M[i]$
si no $\{$ no tomamos más monedas de tipo i $\}$
$i := i - 1$
fsi
fmientras
ffun

Problema del cambio: mejorar espacio adicional



Problema del cambio: mejorar espacio adicional



fun `devolver-cambio2`$(M[1..n]$ **de** $nat^+, C : nat)$ **dev** $número : nat_\infty$
var $monedas[0..C]$ **de** nat_∞
    { inicialización }
    $monedas[0] := 0$
    $monedas[1..C] := [+\infty]$
    { hacer las actualizaciones }
    para $i = 1$ **hasta** n **hacer**
        para $j = M[i]$ **hasta** C **hacer**
            $monedas[j] :=$ `mín`$(monedas[j], monedas[j - M[i]] + 1)$
        fpara
    fpara
    $número := monedas[C]$
ffun

Coste: $\Theta(nC)$ en tiempo y $\Theta(C)$ en espacio adicional.

¿Podemos optimizar en espacio y seguir calculando la solución óptima?

¿Podemos optimizar en espacio y seguir calculando la solución óptima?

La última fila contiene la información sobre el número de monedas mínimo para cada cantidad, con el sistema monetario completo.

Si

$$monedas(n, j) = monedas(n, j - m_i) + 1$$

para algún j y algún i, sabemos que podemos coger una moneda de tipo i para conseguir una solución óptima para pagar j.

Además, como el número de monedas de cada tipo es ilimitado, el sistema monetario no cambia y se puede iterar el proceso para $j - m_i$ haciendo comparaciones de nuevo en la última fila, es decir, el vector.

Para implementar este proceso, al principio j vale C e i vale n; la cantidad j decrece tal y como se van cogiendo monedas y, cuando la ecuación anterior no es cierta y por tanto no se puede coger ninguna moneda más de tipo i, se decrementa i pasando a considerar el tipo de moneda anterior $i - 1$.

$\{\ monedas[C] \neq +\infty\ \}$
fun `calcular-monedas2`$(M[1..n]$ **de** $nat^+, monedas[0..C]$ **de** $nat_\infty)$ $\quad \{\ \Theta(C)\ \}$
dev $cuántas[1..n]$ **de** nat
$\{\ cuántas[i]$ es el número de monedas de tipo i cogidas $\}$
$cuántas[1..n] := [0]$
$i := n\ ;\ \ j := C$
mientras $j > 0$ **hacer** $\{$ no hemos pagado todo $\}$
si $M[i] \leq j\ \wedge_c\ monedas[j] = monedas[j - M[i]] + 1$ **entonces**
$\{$ tomamos una moneda de tipo i $\}$
$cuántas[i] := cuántas[i] + 1$
$j := j - M[i]$
si no $\{$ no tomamos más monedas de tipo i $\}$
$i := i - 1$
fsi
fmientras
ffun

Problema de la mochila (versión entera)

Cuando Alí-Babá consigue por fin entrar en la Cueva de los Cuarenta Ladrones encuentra allí gran cantidad de objetos muy valiosos. A pesar de su pobreza, Alí-Babá conoce muy bien el peso y valor de cada uno de los objetos en la cueva.

Debido a los peligros que tiene que afrontar en su camino de vuelta, solo puede llevar consigo aquellas riquezas que quepan en su pequeña mochila, que soporta un peso máximo conocido.

Suponiendo que los objetos no se pueden fraccionar y que los pesos son números naturales positivos, determinar qué objetos debe elegir Alí-Babá para maximizar el valor total de lo que pueda llevarse en su mochila.

En la cueva hay n objetos, cada uno con un peso (natural) $p_i > 0$ y un valor (real) $v_i > 0$ para todo i entre 1 y n.

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

El problema consiste en 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.

Definimos una función

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

Definimos una función

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

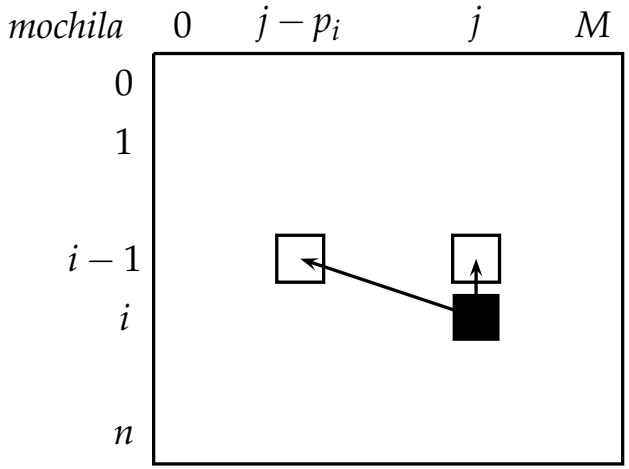
Definición recursiva

$$mochila(i,j) \quad = \quad \begin{cases} mochila(i-1,j) & \text{si } p_i > j \\ \max\{mochila(i-1,j), mochila(i-1,j-p_i)+v_i\} & \text{si } p_i \leq j \end{cases}$$

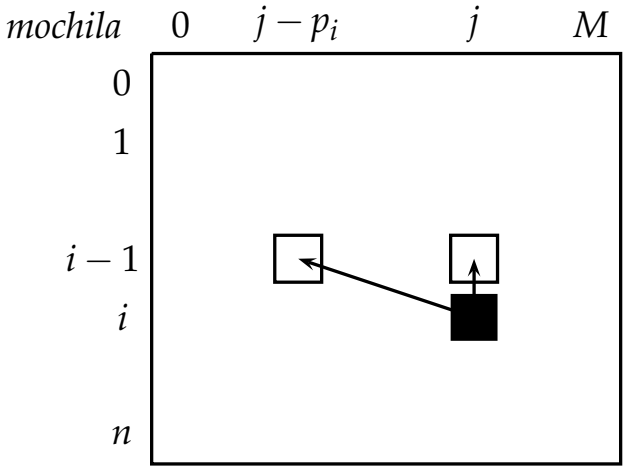
con $1 \leq i \leq n$ y $1 \leq j \leq M$.

Los casos básicos son:

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

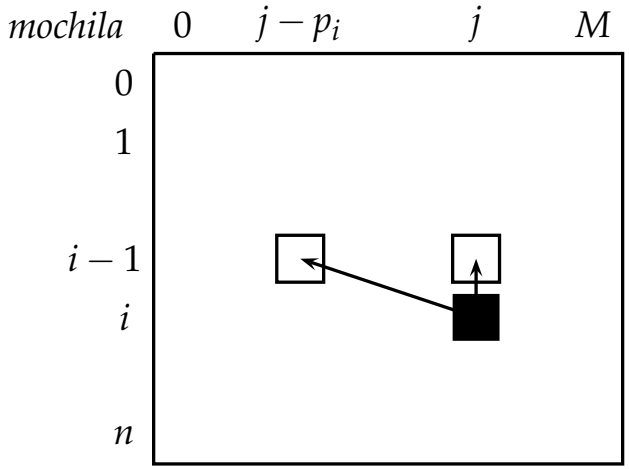


Recorrer la matriz por filas de arriba abajo y cada fila de izquierda a derecha.



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

Si solo quisiéramos el valor máximo alcanzable, podríamos optimizar el espacio adicional utilizando solo un vector que recorreríamos de derecha a izquierda.



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

Si solo quisiéramos el valor máximo alcanzable, podríamos optimizar el espacio adicional utilizando solo un vector que recorreríamos de derecha a izquierda.

Si queremos devolver los objetos que forman parte de la solución óptima no interesa optimizar, porque en ese caso las comparaciones que hacemos para llenar una posición siempre se refieren a posiciones de la fila anterior:

$$mochila(i,j) = \text{máx}\{\underbrace{mochila(i-1,j)}_{\substack{\text{no cogemos}\\ \text{el objeto } i}}, \underbrace{mochila(i-1,j-p_i)+v_i}_{\substack{\text{sí cogemos}\\ \text{el objeto } i}}\}$$

```
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
```

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

Formas de poner sellos

El país de Fanfanisflán emite n sellos diferentes de valores naturales positivos $s_1, s_2, \ldots, s_n$.

Se quiere enviar una carta y se sabe que la correspondiente tarifa postal es T.

¿De cuántas formas diferentes se puede franquear exactamente la carta, si el orden de los sellos no importa?

Formas de poner sellos

El país de Fanfanisflán emite n sellos diferentes de valores naturales positivos $s_1, s_2, \ldots, s_n$.

Se quiere enviar una carta y se sabe que la correspondiente tarifa postal es T.

¿De cuántas formas diferentes se puede franquear exactamente la carta, si el orden de los sellos no importa?

$formas(n, T)$ = número de formas de franquear T con n tipos de sellos.

$$formas(i,j) \quad = \quad \begin{cases} formas(i-1,j) & \text{si } s_i > j \\ formas(i-1,j) + formas(i,j-s_i) & \text{si } s_i \leq j \end{cases}$$

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

$$\begin{aligned} formas(0,j) &= 0 \qquad 1 \leq j \leq T \\ formas(i,0) &= 1 \qquad 0 \leq i \leq n \end{aligned}$$

```
fun sellos(S[1..n] de nat+, T : nat+) dev núm-formas : nat
var formas[0..T] de nat
    { inicialización }
    formas[0] := 1
    formas[1..T] := [0]
    { actualizaciones del vector }
    para i = 1 hasta n hacer
        para j = S[i] hasta T hacer
            formas[j] := formas[j] + formas[j − S[i]]
        fpara
    fpara
    núm-formas := formas[T]
ffun
```

Coste: $\Theta(nT)$ en tiempo y $\Theta(T)$ en espacio adicional.

Reparto del botín

El Maqui y el Popeye acaban de desvalijar la reserva nacional de oro. Los lingotes están empaquetados en n cajas de diferentes pesos naturales positivos p_i para i entre 1 y n.

Como no tienen tiempo de desempaquetarlos para dividir el botín, deciden utilizar los pesos de cada una de las cajas para distribuir el botín a medias.

Al cabo de un buen rato todavía no han conseguido repartirse el botín, por lo cual acuden al Teclas para saber si el botín se puede dividir en dos partes iguales sin desempaquetar las cajas con los lingotes.

Considerando $P = \sum_{i=1}^{n} p_i$, el problema es equivalente a comprobar si es posible, sumando algunos de los pesos, obtener $P/2$.

Considerando $P = \sum_{i=1}^{n} p_i$, el problema es equivalente a comprobar si es posible, sumando algunos de los pesos, obtener $P/2$.

$$\textit{se-puede}(i,j) \quad = \quad \text{booleano que indica si es posible sumar la cantidad } j \text{ eligiendo algunas de las } i \text{ primeras cajas.}$$

Considerando $P = \sum_{i=1}^{n} p_i$, el problema es equivalente a comprobar si es posible, sumando algunos de los pesos, obtener $P/2$.

$$\textit{se-puede}(i,j) \quad = \quad \text{booleano que indica si es posible sumar la cantidad } j \text{ eligiendo algunas de las } i \text{ primeras cajas.}$$

$$\textit{se-puede}(i,j) \quad = \quad \begin{cases} \textit{se-puede}(i-1,j) & \text{si } p_i > j \\ \textit{se-puede}(i-1,j) \ \vee \ \textit{se-puede}(i-1,j-p_i) & \text{si } p_i \leq j \end{cases}$$

con $1 \leq i \leq n$ y $1 \leq j \leq P/2$

Considerando $P = \sum_{i=1}^{n} p_i$, el problema es equivalente a comprobar si es posible, sumando algunos de los pesos, obtener $P/2$.

$$\textit{se-puede}(i,j) \quad = \quad \text{booleano que indica si es posible sumar la cantidad } j \text{ eligiendo algunas de las } i \text{ primeras cajas.}$$

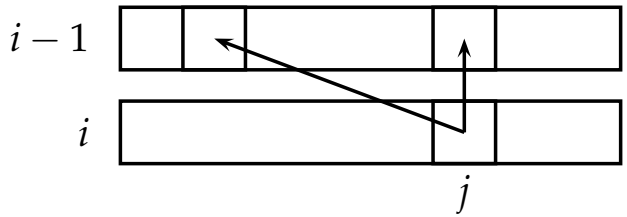
$$\textit{se-puede}(i,j) \quad = \quad \begin{cases} \textit{se-puede}(i-1,j) & \text{si } p_i > j \\ \textit{se-puede}(i-1,j) \ \vee \ \textit{se-puede}(i-1,j-p_i) & \text{si } p_i \leq j \end{cases}$$

con $1 \leq i \leq n$ y $1 \leq j \leq P/2$

$$\begin{array}{lcll} \textit{se-puede}(0,j) & = & \texttt{falso} & 1 \leq j \leq P/2 \\ \textit{se-puede}(i,0) & = & \texttt{cierto} & 0 \leq i \leq n. \end{array}$$

Utilizamos una matriz $\textit{se-puede}[0..n, 0..P/2]$ para calcular los valores de la recurrencia.

Si solo queremos saber si se puede o no sumar la cantidad $P/2$, y no los objetos necesarios para sumar tal cantidad, podemos optimizar el espacio adicional utilizando solo un vector, $\textit{se-puede}[0..P/2]$. Ya que las dos posiciones necesarias para calcular $\textit{se-puede}(i, j)$ se encuentran en la fila anterior,



el vector tiene que recorrerse de derecha a izquierda, para no perder valores de la fila anterior que se necesitan después.

$\{\ P' = P/2\ \}$
fun `repartir-botín-par-pd`$(peso[1..n]$ **de** $nat^+, P' : nat^+)$ **dev** $respuesta : bool$
var $se\text{-}puede[0..P']$ **de** $bool$
    $\{$ inicialización $\}$
    $se\text{-}puede[0] := \texttt{cierto}$
    $se\text{-}puede[1..P'] := [\texttt{falso}]$
    $\{$ actualizaciones del vector $\}$
    para $i = 1$ **hasta** n **hacer**
        para $j = P'$ **hasta** $peso[i]$ **paso** -1 **hacer**
            $se\text{-}puede[j] := se\text{-}puede[j] \vee se\text{-}puede[j - peso[i]]$
        fpara
    fpara
    $respuesta := se\text{-}puede[P']$
ffun

$\{\ P' = P/2\ \}$
fun `repartir-botín-par-pd`$(peso[1..n]$ **de** $nat^+, P' : nat^+)$ **dev** $respuesta : bool$
var $se\text{-}puede[0..P']$ **de** $bool$
    { inicialización }
    $se\text{-}puede[0] := $ `cierto`
    $se\text{-}puede[1..P'] := [$`falso`$]$
    { actualizaciones del vector }
    para $i = 1$ **hasta** n **hacer**
        para $j = P'$ **hasta** $peso[i]$ **paso** -1 **hacer**
            $se\text{-}puede[j] := se\text{-}puede[j] \vee se\text{-}puede[j - peso[i]]$
        fpara
    fpara
    $respuesta := se\text{-}puede[P']$
ffun

$\{\ P = \sum_{i=1}^{n} peso[i]\ \}$
fun `repartir-botín-pd`$(peso[1..n]$ **de** $nat^+, P : nat^+)$ **dev** $respuesta : bool$
    si `impar`(P) **entonces** $respuesta :=$ `falso`
    si no $respuesta :=$ `repartir-botín-par-pd`$(peso, P \text{ div } 2)$
    fsi
ffun

Caminos mínimos: algoritmo de Floyd

Dado un grafo dirigido cuyas aristas están valoradas con números positivos, calcular el *coste (del camino) mínimo* entre cada par de vértices del grafo.

Caminos mínimos: algoritmo de Floyd

Dado un grafo dirigido cuyas aristas están valoradas con números positivos, calcular el *coste (del camino) mínimo* entre cada par de vértices del grafo.

El grafo viene dado por su matriz de adyacencia $G[1..n, 1..n]$

$$G[i,j] = \begin{cases} 0 & \text{si } i = j \\ coste & \text{si hay arista de } i \text{ a } j \\ +\infty & \text{si no hay arista de } i \text{ a } j \end{cases}$$

Caminos mínimos: algoritmo de Floyd

Dado un grafo dirigido cuyas aristas están valoradas con números positivos, calcular el *coste (del camino) mínimo* entre cada par de vértices del grafo.

El grafo viene dado por su matriz de adyacencia $G[1..n, 1..n]$

$$G[i,j] = \begin{cases} 0 & \text{si } i = j \\ coste & \text{si hay arista de } i \text{ a } j \\ +\infty & \text{si no hay arista de } i \text{ a } j \end{cases}$$

Definición de la función

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

Caminos mínimos: algoritmo de Floyd

Dado un grafo dirigido cuyas aristas están valoradas con números positivos, calcular el *coste (del camino) mínimo* entre cada par de vértices del grafo.

El grafo viene dado por su matriz de adyacencia $G[1..n, 1..n]$

$$G[i,j] = \begin{cases} 0 & \text{si } i = j \\ coste & \text{si hay arista de } i \text{ a } j \\ +\infty & \text{si no hay arista de } i \text{ a } j \end{cases}$$

Definición de la función

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

La recurrencia, con $1 \leq k, i, j \leq n$, es

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

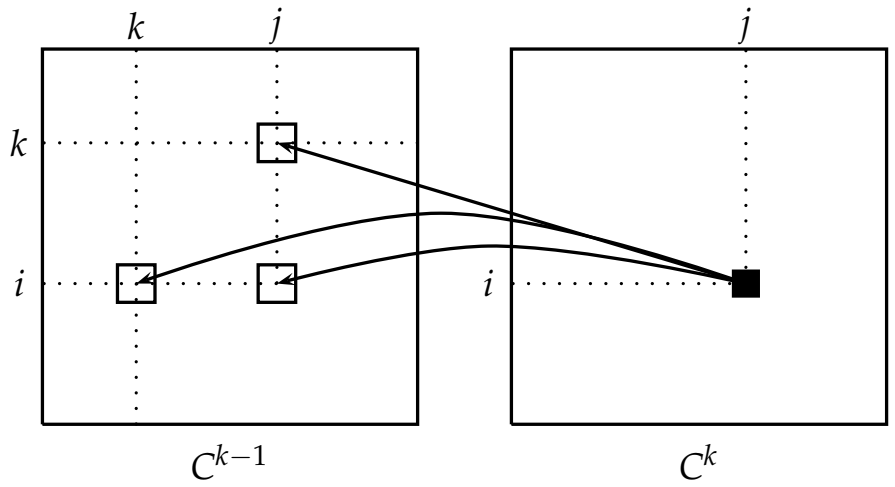
El caso básico se presenta cuando $k = 0$:

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

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

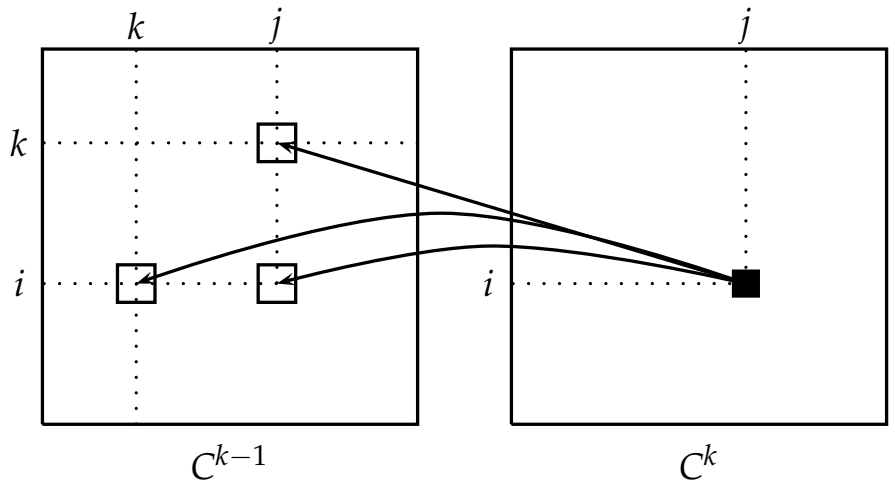
En principio necesitamos $n+1$ matrices $n \times n$, con un espacio adicional en $\Theta(n^3)$. Pero se puede mejorar.

Para calcular la matriz C^k solo necesitamos la matriz C^{k-1} y las actualizaciones se pueden ir realizando sobre la misma matriz.



En principio necesitamos $n+1$ matrices $n \times n$, con un espacio adicional en $\Theta(n^3)$. Pero se puede mejorar.

Para calcular la matriz C^k solo necesitamos la matriz C^{k-1} y las actualizaciones se pueden ir realizando sobre la misma matriz.



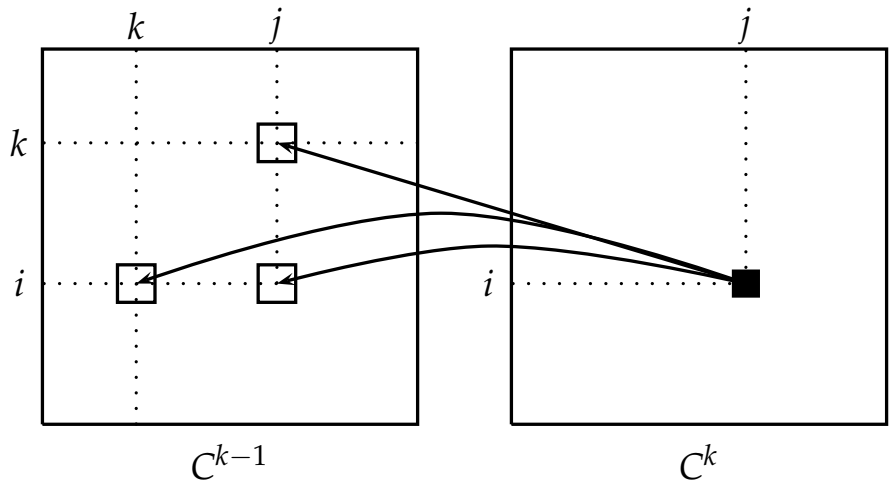
Pero la fila k y la columna k no cambian cuando actualizamos de C^{k-1} a C^k. Para la fila k tenemos

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

ya que $C^{k-1}(k,k) = 0$. Y de igual forma $C^k(i,k) = C^{k-1}(i,k)$ para la columna k.

En principio necesitamos $n+1$ matrices $n \times n$, con un espacio adicional en $\Theta(n^3)$. Pero se puede mejorar.

Para calcular la matriz C^k solo necesitamos la matriz C^{k-1} y las actualizaciones se pueden ir realizando sobre la misma matriz.



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

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

ya que $C^{k-1}(k,k) = 0$. Y de igual forma $C^k(i,k) = C^{k-1}(i,k)$ para la columna k.

Solo utilizamos una matriz $C[1..n, 1..n]$, en la que finalmente se devuelve la solución, de modo que el coste en espacio adicional está en $\Theta(1)$.

```
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: $\Theta(n^3)$ en tiempo y $\Theta(1)$ en espacio adicional.

```
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

El producto de una matriz $A_{p\times q}$ y una matriz $B_{q\times r}$ es una matriz $C_{p\times r}$ cuyos elementos son

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

Se necesitan pqr multiplicaciones entre escalares para calcular C.

Cadena de productos de matrices

El producto de una matriz $A_{p\times q}$ y una matriz $B_{q\times r}$ es una matriz $C_{p\times r}$ cuyos elementos son

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

Se necesitan pqr multiplicaciones entre escalares para calcular C.

Para multiplicar una secuencia de matrices $M_1 M_2 \cdots M_n$ (M_i tiene dimensiones $d_{i-1} \times d_i$) el orden de las matrices no se puede alterar pero sí el de los productos a realizar, ya que la multiplicación de matrices es asociativa.

Cadena de productos de matrices

El producto de una matriz $A_{p\times q}$ y una matriz $B_{q\times r}$ es una matriz $C_{p\times r}$ cuyos elementos son

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

Se necesitan pqr multiplicaciones entre escalares para calcular C.

Para multiplicar una secuencia de matrices $M_1 M_2 \cdots M_n$ (M_i tiene dimensiones $d_{i-1} \times d_i$) el orden de las matrices no se puede alterar pero sí el de los productos a realizar, ya que la multiplicación de matrices es asociativa.

¿Cuál es la mejor forma de insertar paréntesis en la secuencia de matrices de forma que el número total de multiplicaciones entre escalares sea *mínimo*?

Cadena de productos de matrices

El producto de una matriz $A_{p\times q}$ y una matriz $B_{q\times r}$ es una matriz $C_{p\times r}$ cuyos elementos son

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

Se necesitan pqr multiplicaciones entre escalares para calcular C.

Para multiplicar una secuencia de matrices $M_1 M_2 \cdots M_n$ (M_i tiene dimensiones $d_{i-1} \times d_i$) el orden de las matrices no se puede alterar pero sí el de los productos a realizar, ya que la multiplicación de matrices es asociativa.

¿Cuál es la mejor forma de insertar paréntesis en la secuencia de matrices de forma que el número total de multiplicaciones entre escalares sea *mínimo*?

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

$$\underbrace{\underbrace{\underbrace{(\underbrace{(A \cdot B)}_{5785} \cdot C)}_{3471} \cdot D}_{1324}}_{} \quad \rightsquigarrow \quad 10582$$

Cadena de productos de matrices

El producto de una matriz $A_{p\times q}$ y una matriz $B_{q\times r}$ es una matriz $C_{p\times r}$ cuyos elementos son

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

Se necesitan pqr multiplicaciones entre escalares para calcular C.

Para multiplicar una secuencia de matrices $M_1 M_2 \cdots M_n$ (M_i tiene dimensiones $d_{i-1} \times d_i$) el orden de las matrices no se puede alterar pero sí el de los productos a realizar, ya que la multiplicación de matrices es asociativa.

¿Cuál es la mejor forma de insertar paréntesis en la secuencia de matrices de forma que el número total de multiplicaciones entre escalares sea *mínimo*?

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

$$\underbrace{\underbrace{\underbrace{(\underbrace{(A\cdot B)}_{5785}\cdot C)}_{3471}\cdot D}_{1324}}_{10582} \qquad \underbrace{\underbrace{\underbrace{(A\cdot \underbrace{(B\cdot C)}_{1335})}_{195}\cdot D}_{1326}}_{2856}$$

$$\underbrace{(M_1 \cdot \ldots \cdot M_k)}_{d_0 \times d_k} \cdot \underbrace{(M_{k+1} \cdot \ldots \cdot M_n)}_{d_k \times d_n}$$

$$\underbrace{(M_1 \cdot \ldots \cdot M_k)}_{d_0 \times d_k} \cdot \underbrace{(M_{k+1} \cdot \ldots \cdot M_n)}_{d_k \times d_n}$$

Utilizamos la función

$$matrices(i,j) \quad = \quad \text{número } \textit{mínimo} \text{ de multiplicaciones escalares para realizar el producto matricial } M_i \cdots M_j.$$

$$\underbrace{(M_1 \cdot \ldots \cdot M_k)}_{d_0 \times d_k} \cdot \underbrace{(M_{k+1} \cdot \ldots \cdot M_n)}_{d_k \times d_n}$$

Utilizamos la función

$$matrices(i,j) \quad = \quad \text{número } \textit{mínimo} \text{ de multiplicaciones escalares para realizar el producto matricial } M_i \cdots M_j.$$

La recurrencia solo tiene sentido cuando $i \leq j$. El caso recursivo, $i < j$, se define de la siguiente manera:

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

$$\underbrace{(M_1 \cdot \ldots \cdot M_k)}_{d_0 \times d_k} \cdot \underbrace{(M_{k+1} \cdot \ldots \cdot M_n)}_{d_k \times d_n}$$

Utilizamos la función

$$matrices(i,j) \quad = \quad \text{número } \textit{mínimo} \text{ de multiplicaciones escalares para realizar el producto matricial } M_i \cdots M_j.$$

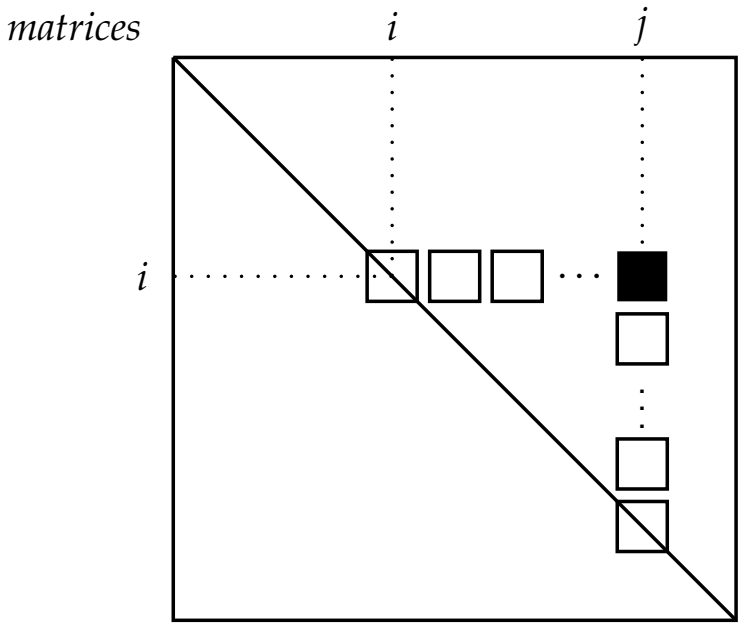
La recurrencia solo tiene sentido cuando $i \leq j$. El caso recursivo, $i < j$, se define de la siguiente manera:

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

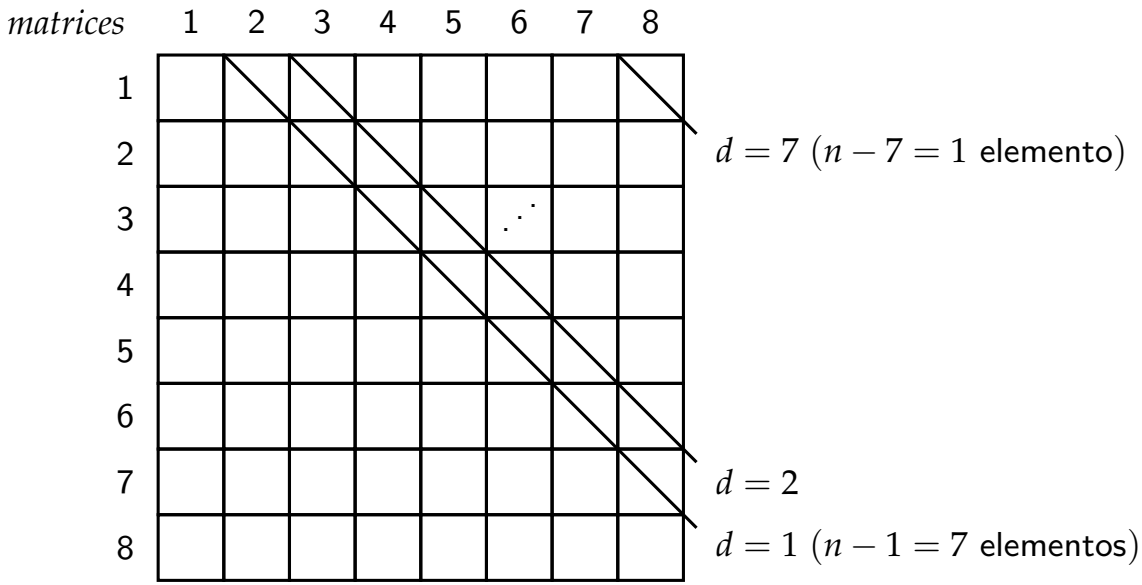
El caso básico se presenta cuando solo tenemos una matriz, esto es, $i = j$:

$$matrices(i,i) \quad = \quad 0.$$

Utilizaremos una tabla $matrices[1..n, 1..n]$, de la cual solo necesitaremos la mitad superior por encima de la diagonal principal.



Rellenar la matriz recorriéndola por diagonales.



- Las diagonales se numeran desde $d = 1$ hasta $d = n - 1$ en el orden en el que tienen que recorrerse.
- Cada diagonal tiene $n - d$ elementos que numeraremos del $i = 1$ al $i = n - d$.
- Así este índice nos sirve directamente para conocer la fila en la que se encuentra el elemento por el que vamos.
- La columna podemos calcularla mediante $j = i + d$.

```
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
```

```
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 $\Theta(n^2)$

```
{ 1 ≤ i ≤ j ≤ n }
proc escribir-parén(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-parén(i,k,P); imprimir(")")
        si no imprimir("M_i")
        fsi
        imprimir("*")
        si k+1 < j entonces imprimir("("); escribir-parén(k+1,j,P); imprimir(")")
        si no imprimir("M_j")
        fsi
    fsi
fproc
```

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.

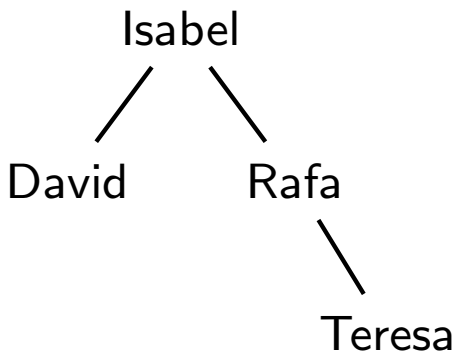
Queremos encontrar un árbol de búsqueda que minimice el número medio de comparaciones necesarias para realizar una búsqueda, suponiendo que $\sum_{i=1}^{n} p_i = 1$, es decir, que todas las peticiones se refieren a claves que están en el árbol.

Árboles binarios de búsqueda óptimos

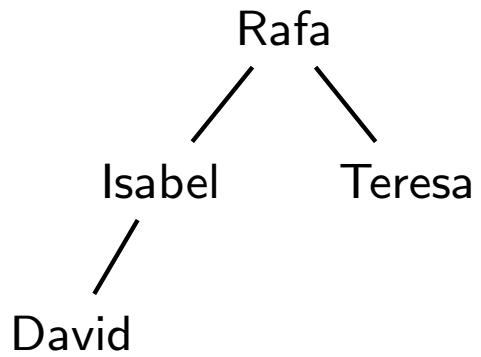
Sean $c_1 < c_2 < \cdots < c_n$ un conjunto de claves distintas ordenadas, y sea p_i la probabilidad con que se pide buscar la clave c_i y su información asociada.

Queremos encontrar un árbol de búsqueda que minimice el número medio de comparaciones necesarias para realizar una búsqueda, suponiendo que $\sum_{i=1}^{n} p_i = 1$, es decir, que todas las peticiones se refieren a claves que están en el árbol.

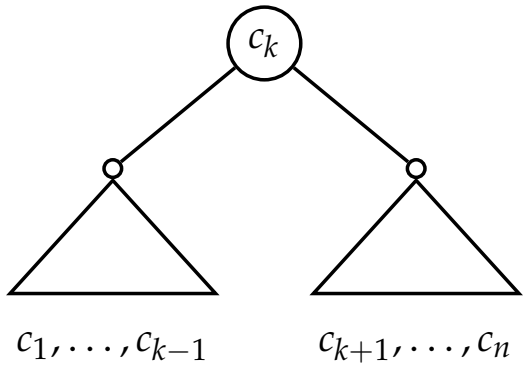
Ejemplo: $c_1 = \text{David}$, $c_2 = \text{Isabel}$, $c_3 = \text{Rafa}$, $c_4 = \text{Teresa}$
$c_1 = \frac{3}{8}$, $p_2 = \frac{3}{8}$, $p_3 = \frac{1}{8}$, $p_4 = \frac{1}{8}$



$\frac{7}{4}$



$\frac{18}{8}$



Consideremos un árbol de búsqueda a para las claves $\{c_1, \ldots, c_n\}$ en el que la clave c_k está en la raíz.

Utilizamos la función

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

El número medio de comparaciones viene dado por:

$$
\begin{aligned}
comp_a(1,n) &= \sum_{l=1}^{n} p_l \, nivel_a(c_l) \\
&= p_k + \sum_{l=1}^{k-1} p_l \, nivel_a(c_l) + \sum_{l=k+1}^{n} p_l \, nivel_a(c_l) \\
&= p_k + \sum_{l=1}^{k-1} p_l \, (nivel_{hi(a)}(c_l) + 1) + \sum_{l=k+1}^{n} p_l \, (nivel_{hd(a)}(c_l) + 1) \\
&= p_k + \sum_{l=1}^{k-1} p_l + \sum_{l=k+1}^{n} p_l + \sum_{l=1}^{k-1} p_l \, nivel_{hi(a)}(c_l) + \sum_{l=k+1}^{n} p_l \, nivel_{hd(a)}(c_l) \\
&= \sum_{l=1}^{n} p_l + comp_{hi(a)}(1,k-1) + comp_{hd(a)}(k+1,n)
\end{aligned}
$$

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

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

$$comp(0,n) \quad = \quad \sum_{l=1}^{n} p_l + \min_{1 \leq k \leq n} \{comp(0,k-1) + comp(k,n)\}.$$

En el caso general, la recurrencia se define para $0 \leq i < j \leq n$:

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

El caso básico es $comp(i,i) = 0$, donde $0 \leq i \leq n$, que corresponde al árbol vacío.

Recorrido por diagonales.

Matriz adicional $prob[0..n, 0..n]$. Con la definición

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

se puede calcular la matriz mediante la fórmula

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

teniendo como caso básico $prob[i,i] = 0$.

Recorrido por diagonales.

Matriz adicional $prob[0..n, 0..n]$. Con la definición

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

se puede calcular la matriz mediante la fórmula

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

teniendo como 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

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

$\{\ C[1] < \cdots < C[n]\ \wedge\ \forall i : 1 \leq i \leq n : 0 \leq P[i] \leq 1\ \}$
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

$\{\ C[1] < \cdots < C[n]\ \wedge\ \forall i : 1 \leq i \leq n : 0 \leq P[i] \leq 1\ \}$
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 $\Theta(n^2)$

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