Programación dinámica

Alberto Verdejo

Dpto. de Sistemas Informáticos y Computación

Universidad Complutense de Madrid

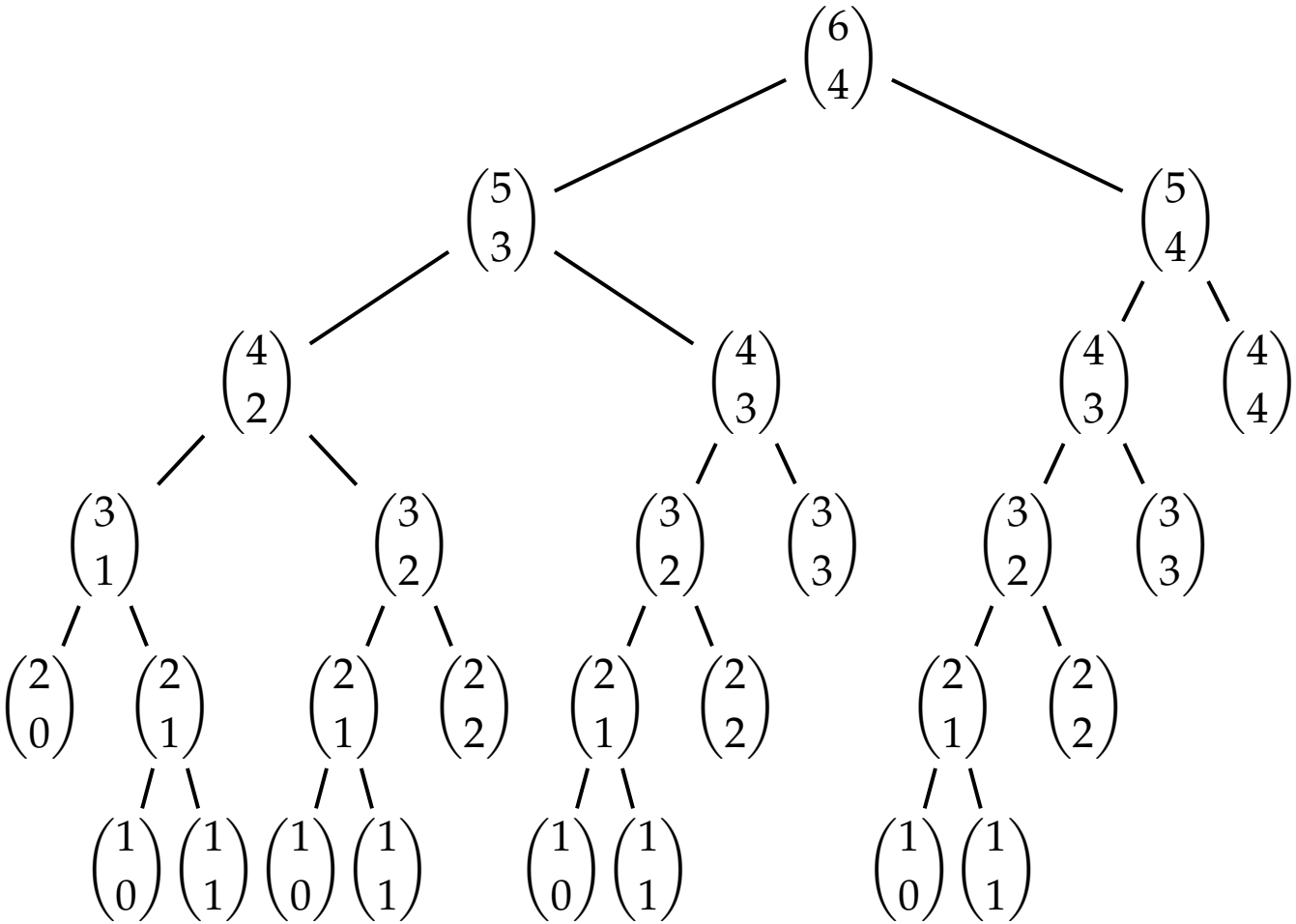
Noviembre 2015

Bibliografía

- R. Neapolitan. *Foundations of Algorithms*. Quinta edición. Jones and Bartlett Learning, 2015.
  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 A. Verdejo. *Estructuras de datos y métodos algorítmicos: 213 ejercicios resueltos*. Segunda edición, Garceta, 2013.
  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.

```
int num_combi(int i, int j, Matriz<int> & C) {
    if (j == 0 || j == i)
        return 1;
    else if (C[i][j] != -1)
        return C[i][j];
    else {
        int nc = num_combi(i-1, j-1, C);
               + num_combi(i-1, j, C);
        return C[i][j] = nc;
    }
}

Matriz<int> C(n+1, k+1, -1);

cout << num_combi(n, k, C) << "\n";
```

Inicialización con tiempo en $\Theta(1)$. ¿Es posible?

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.
- Posible optimización en espacio.

Números combinatorios

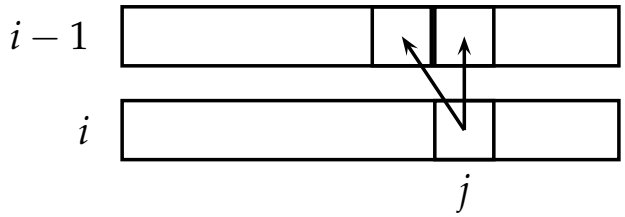
	0	1	2	...	r
0	1	0	0	...	0
1	1	1	0	...	0
2	1	2	1	...	0
⋮	⋮	⋮	⋮	⋱	⋮
n	1	n	$\binom{n}{2}$	...	$\binom{n}{r}$

```
int pascal(int n, int r) {
    Matriz<int> C(n+1,r+1,0);
    C[0][0] = 1;
    for (int i = 1; i <= n; ++i) {
        C[i][0] = 1;
        for (int j = 1; j <= r; ++j) {
            C[i][j] = C[i-1][j-1] + C[i-1][j];
        }
    }
    return C[n][r];
}
```

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

Números combinatorios, mejorar espacio adicional

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.




```
int pascal2(int n, int r) {
    vector<int> C(r+1,0);
    C[0] = 1;
    for (int i = 1; i <= n; ++i) {
        for (int j = r; j >= 1; --j) {
            C[j] = C[j-1] + C[j];
        }
    }
    return C[r];
}
```

Coste: en tiempo $\Theta(nr)$
en espacio $\Theta(r)$

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.

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

Se cumple el principio de optimalidad de Bellman según el cual para conseguir una solución óptima basta considerar subsoluciones óptimas.

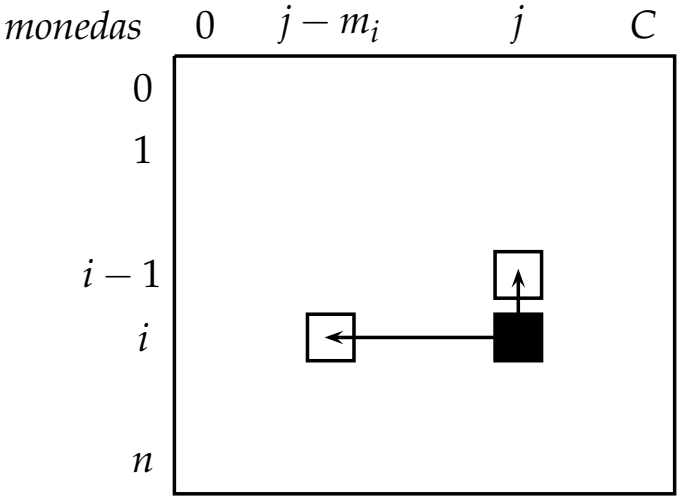
Definición recursiva:

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

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

Casos básicos:

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



```
double devolver_cambio1(vector<int> const& M, int C) {
    size_t n = M.size() - 1;
    Matriz<double> monedas(n+1, C+1, numeric_limits<double>::infinity());
    monedas[0][0] = 0;
    // rellenar la matriz
    for (int i = 1; i <= n; ++i) {
        monedas[i][0] = 0;
        for (int j = 1; j <= C; ++j) {
            if (M[i] > j) // no se puede considerar M[i] para pagar j
                monedas[i][j] = monedas[i-1][j];
            else
                monedas[i][j] = min(monedas[i-1][j], monedas[i][j - M[i]] + 1);
        }
    }
    return monedas[n][C];
}
```

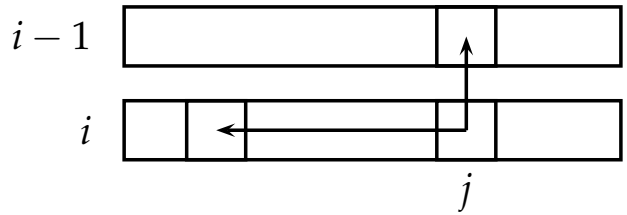
Coste: $\Theta(nC)$ tanto en tiempo como en espacio adicional.

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: mejorar espacio adicional



Rellenar de izquierda a derecha.

```
double devolver_cambio2(vector<int> const& M, int C) {
    size_t n = M.size() - 1;
    vector<double> monedas(C+1, numeric_limits<double>::infinity());
    monedas[0] = 0;
    // rellenar la matriz
    for (int i = 1; i <= n; ++i) {
        for (int j = M[i]; j <= C; ++j) {
            monedas[j] = min(monedas[j], monedas[j - M[i]] + 1);
        }
    }
    return monedas[C];
}
```

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

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] != infinito
vector<int> calcular_monedas1(vector<int> const& M, int C,
                              Matriz<double> const& monedas) {
    size_t n = M.size() - 1;
    vector<int> cuantas(n+1, 0);
    int i = n; int j = C;
    while (j > 0) {  // no se ha pagado todo
        if (M[i] <= j && monedas[i][j] != monedas[i-1][j]) {
            // tomamos una moneda de tipo i
            ++cuantas[i];
            j = j - M[i];
        } else // no tomamos más monedas de tipo i
            --i;
    }
    return cuantas;
}
```

¿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] != infinito
vector<int> calcular_monedas2(vector<int> const& M, int C,
                              vector<double> const& monedas) {
    size_t n = M.size() - 1;
    vector<int> cuantas(n+1, 0);
    int i = n; int j = C;
    while (j > 0) {  // no se ha pagado todo
        if (M[i] <= j && monedas[j] == monedas[j - M[i]] + 1) {
            // tomamos una moneda de tipo i
            ++cuantas[i];
            j = j - M[i];
        } else // no tomamos más monedas de tipo i
            --i;
    }
    return cuantas;
}
```

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

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

Definición recursiva

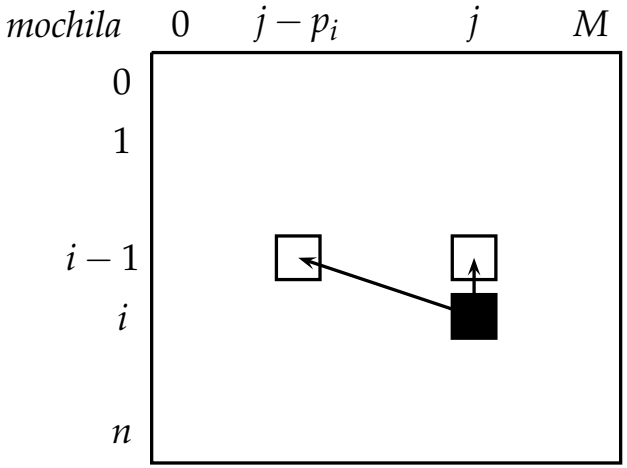
$$mochila(i,j) \quad = \quad \begin{cases} mochila(i-1,j) & \text{si } p_i > j \\ \max\{mochila(i-1,j), mochila(i-1,j-p_i)+v_i\} & \text{si } p_i \leq j \end{cases}$$

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

Los casos básicos son:

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

Problema inicial: $mochila(n,M)$.



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:

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

```
void mochila_pd(vector<int> const& P, vector<double> const& V, int M,
                double & valor, vector<bool> & cuales) {
    size_t n = P.size() - 1;
    Matriz<double> mochila(n+1, M+1, 0);
    // rellenar la matriz
    for (int i = 1; i <= n; ++i) {
        for (int j = 1; j <= M; ++j) {
            if (P[i] > j) // no se puede considerar M[i] para pagar j
                mochila[i][j] = mochila[i-1][j];
            else
                mochila[i][j] = max(mochila[i-1][j],
                                    mochila[i-1][j - P[i]] + V[i]);
        }
    }
    valor = mochila[n][M];
```

```
    // cálculo de los objetos
    int resto = M;
    for (size_t i = n; i >= 1; --i) {
        if (mochila[i][resto] == mochila[i-1][resto]) {
            // no cogemos objeto i
            cuales[i] = false;
        } else { // sí cogemos objeto i
            cuales[i] = true;
            resto = resto - P[i];
        }
    }
}
```

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

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

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

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

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

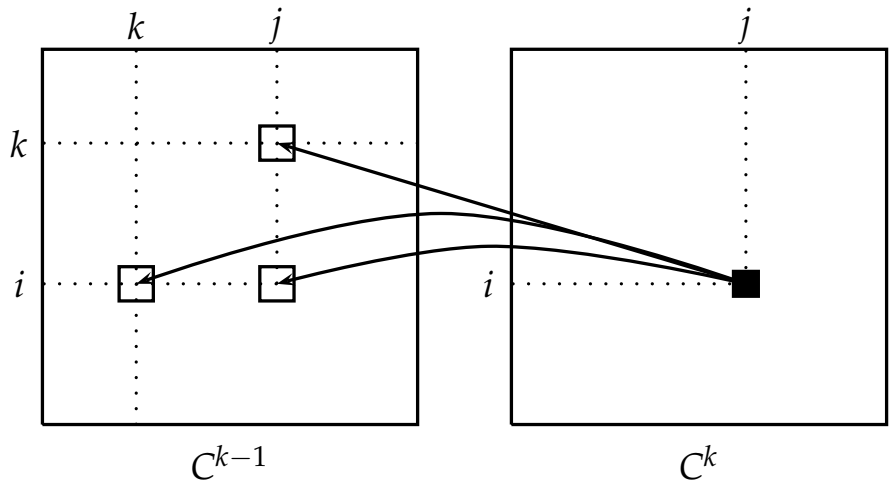
El caso básico se presenta cuando $k = 0$:

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

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.



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 $\Theta(1)$.

```
void Floyd(Matriz<double> const& G, Matriz<double> & C,
                                    Matriz<size_t> & camino) {
    size_t n = G.numfils() - 1;
    // inicialización
    C = G;
    camino = Matriz<size_t>(n+1,n+1,0);
    // actualizaciones de la matriz
    for (size_t k = 1; k <= n; ++k)
        for (size_t i = 1; i <= n; ++i)
            for (size_t j = 1; j <= n; ++j) {
                double temp = C[i][k] + C[k][j];
                if (temp < C[i][j]) { // es mejor pasar por k
                    C[i][j] = temp;
                    camino[i][j] = k;
                }
            }
}
```

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

```
void imprimir_caminos(Matriz<double> const& C,
                      Matriz<size_t> const& camino) {
    size_t n = C.numfils() - 1;
    for (size_t i = 1; i <= n; ++i)
        for (size_t j = 1; j <= n; ++j)
            if (i != j && C[i][j] < numeric_limits<double>::infinity()) {
                cout << "Camino de " << i << " a " << j << "\n";
                cout << "    Coste: " << C[i][j] << "\n";
                cout << "    " << i;
                imp_camino_int(i,j,camino);
                cout << " " << j << "\n";
            }
}

void imp_camino_int(size_t i, size_t j, Matriz<size_t> const& camino) {
    size_t k = camino[i][j];
    if (k > 0) {
        imp_camino_int(i, k, camino);
        cout << " " << k;
        imp_camino_int(k, j, camino);
    }
}
```

```
bool resuelveCaso() {
    size_t V;  cin >> V;

    if (!cin) return false;

    size_t E;  cin >> E;

    Matriz<double> grafo(V+1,V+1,numeric_limits<double>::infinity());
    for (size_t i = 1; i <= V; ++i)
        grafo[i][i] = 0;

    size_t u, v; double coste;
    for (size_t i = 1; i <= E; ++i) { // leer aristas
        cin >> u >> v >> coste;
        grafo[u][v] = coste;
    }

    Matriz<double> C(0,0); Matriz<size_t> camino(0,0);
    Floyd(grafo, C, camino);
    imprimir_caminos(C, camino);

    return true;
}
```

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

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

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

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

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:

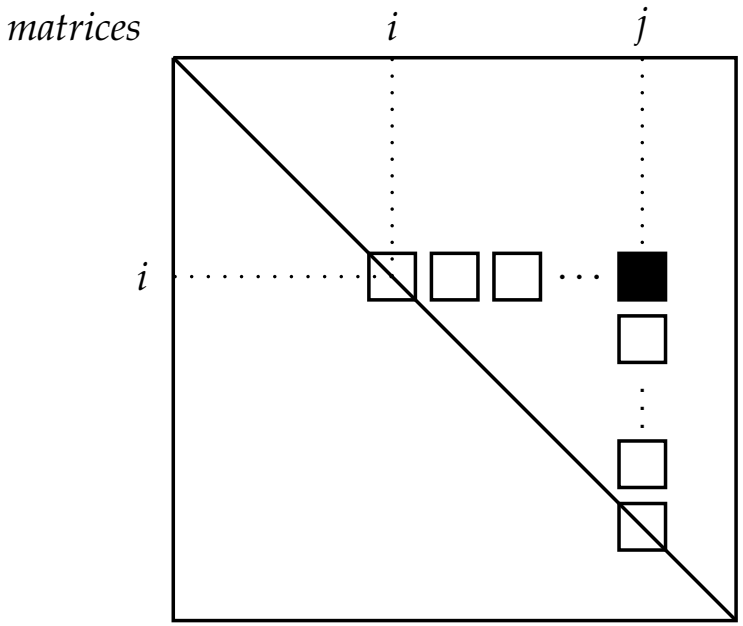
$$matrices(i,j) \quad = \quad \text{mín}_{i \leq k \leq j-1}\{matrices(i,k) + matrices(k+1,j) + d_{i-1}d_k d_j\}.$$

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

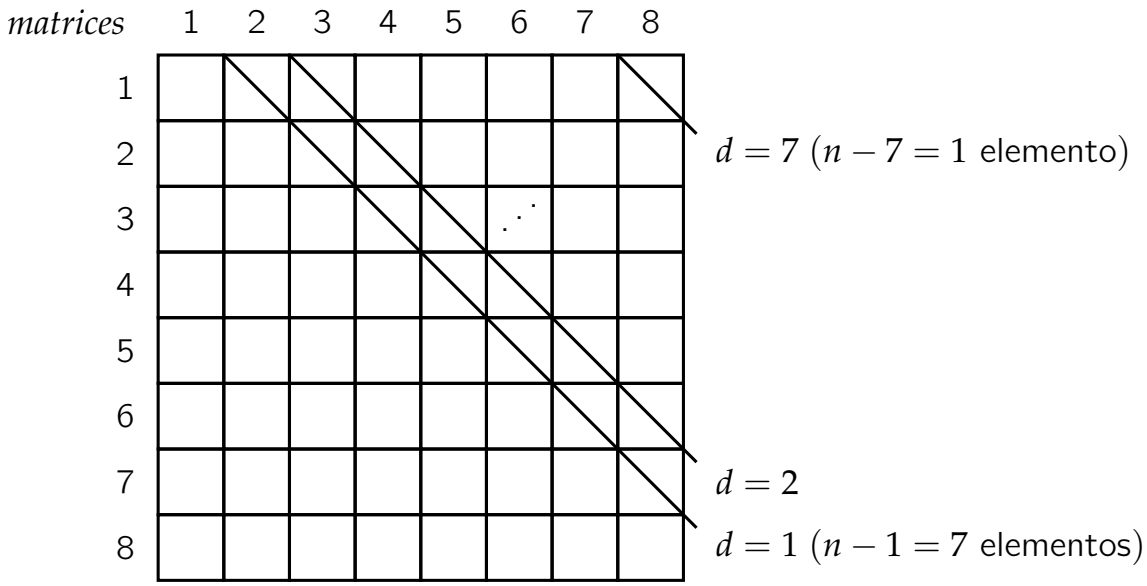
$$matrices(i,i) \quad = \quad 0.$$

Problema inicial: $matrices(1,n)$.

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 $j = i + d$.

```
int multiplica_matrices(vector<size_t> const& D, Matriz<size_t> & P) {
    size_t n = D.size() - 1;
    // inicialización
    Matriz<int> matrices(n+1,n+1,0);
    P = Matriz<size_t>(n+1,n+1,0);
    // recorrido por diagonales
    for (size_t d = 1; d <= n-1; ++d)  // recorre diagonales
        for (size_t i = 1; i <= n - d; ++i) {  // recorre elementos de diagonal
            size_t j = i + d;
            // calcular mínimo
            matrices[i][j] = INT_MAX;
            for (size_t k = i; k <= j-1; ++k) {
                int temp = matrices[i][k] + matrices[k+1][j] + D[i-1]*D[k]*D[j];
                if (temp < matrices[i][j]) { // es mejor pasar por k
                    matrices[i][j] = temp;
                    P[i][j] = k;
                }
            }
        }
    return matrices[1][n];
}
```

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

```
// 1 <= i <= j <= n
void escribir_paren(size_t i, size_t j, Matriz<size_t> const& P) {
    if (i == j) {
        cout << "M" << i;
    } else {
        size_t k = P[i][j];
        if (k > i) {
            cout << "("; escribir_paren(i, k, P); cout << ")";
        } else
            cout << "M" << i;
        cout << "*";
        if (k+1 < j) {
            cout << "("; escribir_paren(k+1, j, P); cout << ")";
        } else
            cout << "M" << j;
    }
}
```

Justificación de un texto

Dadas n palabras de longitudes $l_1, \ldots, l_n$, se desea colocarlas (sin partir y en el orden dado) en un párrafo con líneas de longitud L, de forma que entre cada dos palabras debe existir al menos un blanco. Si una línea contiene las palabras de la i a la j, siendo $i \leq j$, y se deja exactamente un blanco entre cada dos palablas, el número de blancos extra que hacen falta para rellenar la línea son $L - (j-i) - \sum_{k=i}^{j} l_k$. Añadir dichos blancos extra tiene la siguiente penalización:

$$penaliza(i,j) = \left(L - (j-i) - \sum_{k=i}^{j} l_k \right)^3$$

Queremos minimizar la suma de las penalizaciones de todas las líneas excepto la última.

Justificación de un texto

$$\textit{párrafo}(i) \quad = \quad \text{penalización } \textit{mínima} \text{ al formatear las palabras de la } i \text{ a la } n \text{ empezando en una línea en blanco.}$$

Los casos básidos corresponden a necesitar solamente una línea, por la que no se paga penalización.

$$\textit{párrafo}(i) \quad = \quad 0 \quad \text{si} \sum_{k=i}^{n} l_k + (n-i) \leq L$$

En otro caso, las posibilidades varían desde poner solamente una palabra en la primera línea hasta poner todas las que quepan, y continuar de forma óptima con el resto de palabras, comenzando una nueva línea.

$$\textit{párrafo}(i) \quad = \quad \min_{\substack{i \leq j \leq n \\ \sum_{k=i}^{j} l_k + (j-i) \leq L}} \left\{ \textit{penaliza}(i,j) + \textit{párrafo}(j+1) \right\}$$

Utilizamos un vector que va rellenándose de derecha a izquierda.

Justificación de un texto

```
int formatear_parrafo(vector<int> const& l, int L, vector<size_t> & decision) {
    size_t n = l.size() - 1;
    vector<int> parrafo(n+2, 0);
    decision = vector<size_t>(n+1);
    size_t i = n; int suma = l[n];
    while (suma + (n-i) <= L) {
        parrafo[i] = 0; decision[i] = n;
        --i; suma += l[i];
    }
    while (i >= 1) {
        size_t j = i; suma = l[i]; parrafo[i] = INT_MAX;
        while (j <= n && suma + (j-i) <= L) {
            int pen = L - (suma + (j-i));
            int nuevo = pen*pen*pen + parrafo[j+1];
            if (parrafo[i] > nuevo) {
                parrafo[i] = nuevo; decision[i] = j;
            }
            ++j; suma += l[j];
        }
        --i;
    }
    return parrafo[1];
}
```