Algorítmica

Tema 4. Algoritmos basados en programación dinámica

Francisco Javier Cabrerizo Lorite

Curso académico 2023-2024

ETS de Ingenierías Informática y de Telecomunicación. Universidad de Granada

Índice

1. Motivación

2. Programación dinámica

3. El problema del cambio de monedas

4. El problema de la mochila 0/1

5. Multiplicación encadenada de matrices

6. Caminos mínimos

Objetivos

- Ser capaz de proponer diferentes soluciones para un determinado problema y evaluar la calidad de estas.
- Comprender la técnica de resolución de problemas de programación dinámica e identificar las diferencias con «divide y vencerás» y con *greedy*.
- Saber identificar los problemas que cumplen el principio de optimalidad y qué se necesita para poder aplicar esta técnica.
- Conocer los criterios de aplicación de cada una de las distintas técnicas de diseño de algoritmos.

Motivación

Motivación

Fibonacci

- La sucesión de Fibonacci es:

$$F_n = \begin{cases} n & n = 0, 1 \\ F_{n-1} + F_{n-2} & n > 1 \end{cases}$$

- Algoritmo recursivo:

```
1 int fibonacci(int n) {
2   if (n <= 1)
3     return n;
4   else
5     return fibonacci(n-1) + fibonacci(n-2);
6 }
```

- El tiempo de ejecución es $O(\Phi^n)$.

Motivación

Algoritmo recursivo de Fibonacci

- Es ineficiente porque se repiten muchos cálculos:

$$\begin{aligned} F_6 &= F_5 + F_4 = F_4 + F_3 + F_3 + F_2 \\ &= F_3 + F_2 + F_2 + F_1 + F_2 + F_1 + F_2 \\ &= F_2 + F_1 + F_2 + F_2 + F_1 + F_2 + F_1 + F_2 \end{aligned}$$

- Los subproblemas se **solapan**.
- Se podría mejorar si se almacenan los resultados y se calcula cada F_i solo una vez.

Algoritmo lineal de Fibonacci

- Se puede obtener un algoritmo más eficiente usando una tabla para **almacenar** los cálculos si estos se **organizan** de forma ordenada.

```
1  int fibonacci_lineal(int n) {
2    int tabla[n+1];
3    if (n <= 1)
4      return n;
5    else {
6      tabla[0] = 0;
7      tabla[1] = 1;
8      for (int i = 2; i <= n; i++)
9        tabla[i] = tabla[i-1] + tabla[i-2];
10     return tabla[n];
11   }
12 }
```

- El tiempo de ejecución es $O(n)$ y el coste espacial es lineal.

Algoritmo de Fibonacci eficiente en espacio

- Como el término F_n depende solo de F_{n-1} y F_{n-2}, no es necesario almacenar toda la tabla.

```
1  int fibonacci_eficiente(int n) {
2    int t1 = 1, t2 = 0, tn;
3    if (n <= 1)
4      return n;
5    else {
6      for (int i = 2; i <= n; i++) {
7        tn = t1 + t2;
8        t2 = t1;
9        t1 = tn;
10     }
11     return tn;
12   }
13 }
```

- El tiempo de ejecución es $O(n)$ y el coste espacial es constante.

Números combinatorios

- Calcula el número de subconjuntos de tamaño k que se pueden formar con n elementos distintos:

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

- Se puede calcular recursivamente:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

- Algoritmo recursivo

```
1 int combinatorio(int n, int k) {
2   if (k == 0 || k == n)
3     return 1;
4   else
5     return combinatorio(n-1, k-1) + combinatorio(n-1,k);
6 }
```

Algoritmo recursivo de los números combinatorios

- Es ineficiente porque muchos valores $C(i,j)$, con $i < n$, $j < k$, se calculan varias veces:

$$
\begin{aligned}
C(5,3) &= C(4,2) + C(4,3) \\
&= C(3,1) + C(3,2) + C(3,2) + C(3,3) \\
&= C(2,0) + C(2,1) + C(2,1) + C(2,2) + C(2,1) + C(2,2) + 1 \\
&= 1 + C(1,0) + C(1,1) + C(1,0) + C(1,1) + 1 + C(1,0) + C(1,1) + 1 + 1 \\
&= 10
\end{aligned}
$$

Motivación

Números combinatorios

- Si se usa una tabla de resultados intermedios $C[i, j]$, $i = 1, \ldots, n$, $j = 0, \ldots, k$ (triángulo de Pascal) se obtiene un resultado más eficiente.
- Se rellena la tabla línea por línea:

	0	1	2	3	4	5	$\cdots$
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1	4	6	4	1		
5	1	5	10	10	5	1	
$\vdots$			$\vdots$				$\ddots$

Números combinatorios

- Algoritmo más eficiente:

```
1  int combinatorio_eficiente(int n, int k) {
2    int tabla[n+1][k+1];
3
4    for (int i = 0; i <= n; i++) {
5      for (int j = 0; j <= min(i,k); j++) {
6        if (j == 0 || j == i)
7          tabla[i][j] = 1;
8        else
9          tabla[i][j] = tabla[i-1][j-1] + tabla[i-1][j];
10     }
11   }
12   return tabla[n][k];
13 }
```

- El tiempo de ejecución es $O(n \cdot k)$ y el coste espacial es $n \cdot k$.

Números combinatorios

- Algoritmo más eficiente en espacio:

```
1  int combinatorio_eficiente_espacio(int n, int k) {
2    int tabla[n+1];
3    tabla[0] = 1;
4
5    for (int i = 1; i <= n; i++) {
6      tabla[i] = 1;
7      for (int j = i-1; j > 0; j--) {
8        tabla[j] += tabla[j-1];
9      }
10   }
11   return tabla[k];
12 }
```

- El tiempo de ejecución es $O(n \cdot k)$ y el coste espacial es n.

Motivación

¿Qué se ha visto?

- La eficiencia disminuye mucho cuando un problema se divide recursivamente en subproblemas que se solapan.
- Se pueden mantener en memoria los subproblemas resueltos para no repetir cálculos y mejorar la eficiencia.

«Aquellos que no recuerdan su pasado
están condenados a repetirlo»

Programación dinámica

Características de los problemas (1/2)

- Suelen ser problemas de **optimización**.
- Deben poder resolverse **por etapas**.
- Deben poder modelarse mediante una **ecuación recurrente**.
- Deben cumplir el **Principio de Optimalidad de Bellman**.

Características de los problemas (2/2)

- Requieren un alto coste computacional (posiblemente exponencial) donde:
  - **Subproblemas optimales**. La solución óptima a un problema se puede definir en función de soluciones óptimas a subproblemas de tamaño menor, generalmente de forma recursiva.
  - **Solapamiento entre subproblemas**. Al plantear la solución recursiva, un mismo problema se resuelve más de una vez.

Características de la técnica (1/2)

- Introducida por Richard Bellman en los años 50.
- No tiene nada que ver con la programación (es más bien «planificación»).
- Inspirada en la teoría de control: se obtiene la política óptima para un problema de control con n etapas basándose en una política óptima para un problema similar, pero de $n - 1$ etapas).
- Resuelve un problema por etapas (como *greedy* o *backtracking*).
- Divide el problema en subproblemas (como «divide y vencerás»).

Características de la técnica (2/2)

- Suele ser una técnica **ascendente** (*bottom-up*):
  - Para obtener la solución, primero calcula las soluciones óptimas a problemas de tamaño pequeño.
  - Usando tales soluciones, encuentra soluciones a problemas de mayor tamaño.
- Para evitar cálculos repetidos, retiene en memoria las soluciones de los subproblemas (**memoización**).
- Devuelve la **solución óptima** (Principio de Optimalidad de Bellman).

Programación dinámica

Idea general

- Encontrar una formulación recursiva de la solución de un problema mayor en función de soluciones a problemas menores.
- Resolver cada instancia de tamaño menor una única vez y guardarla en una tabla.
- Obtener la solución de la instancia inicial mediante las soluciones almacenadas.

Análisis de la eficiencia

- Depende del problema, aunque suele ser **polinomial**.
- En general, al usar una tabla, será $n \cdot m$:
  - n es el tamaño de la tabla.
  - m es el tiempo para rellenar cada casilla.
- Puede dar problemas al necesitar mucha memoria (para rellenar la tabla).

Comparación con «divide y vencerás» (1/2)

- Ambos combinan soluciones de subproblemas.
- Ambos obtienen la solución óptima.
- Dependencia de problemas:
  - «Divide y vencerás» si los subproblemas son independientes.
  - Programación dinámica si los subproblemas se solapan.
- Recursividad e iteratividad:
  - «Divide y vencerás» usa recurrencias (más tiempo, menos memoria).
  - Programación dinámica intenta evitar recurrencias, usa memoria para la iteración (menos tiempo, más memoria).

Comparación con «divide y vencerás» (2/2)

- Repetición de cálculos:
  - «Divide y vencerás» repite muchos cálculos.
  - Programación dinámica mantiene los resultados en memoria para evitar repetir cálculos.
- Método empleado:
  - «Divide y vencerás» usa un método descendente.
  - Programación dinámica usa un método ascendente (normalmente).

Comparación con *greedy*

- Ambos resuelven el problema por etapas, pero:
  - *Greedy* selecciona un elemento y solo genera una solución, en cada etapa.
  - Programación dinámica selecciona un elemento en cada paso, pero genera múltiples caminos de etapas a seguir, eligiendo la óptima entre estas.
- Optimalidad:
  - *Greedy* no asegura optimalidad (miope).
  - Programación dinámica asegura optimalidad (Principio de Optimalidad de Bellman).
- Eficiencia:
  - *Greedy* es eficiente en tiempo y memoria.
  - Programación dinámica es eficiente en tiempo pero no en memoria.

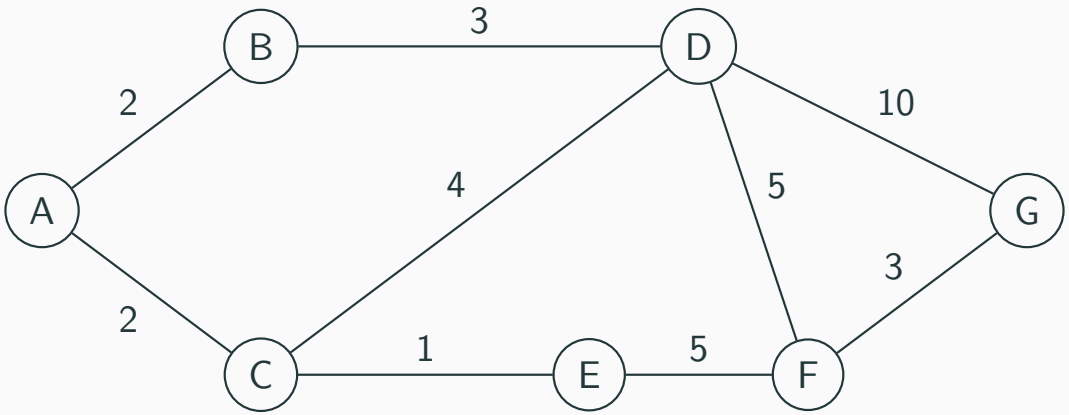
Programación dinámica

Principio de Optimalidad de Bellman

Una secuencia óptima de decisiones que resuelve un problema debe cumplir la propiedad de que cualquier subsecuencia de decisiones debe ser también óptima respecto al subproblema que resuelve.

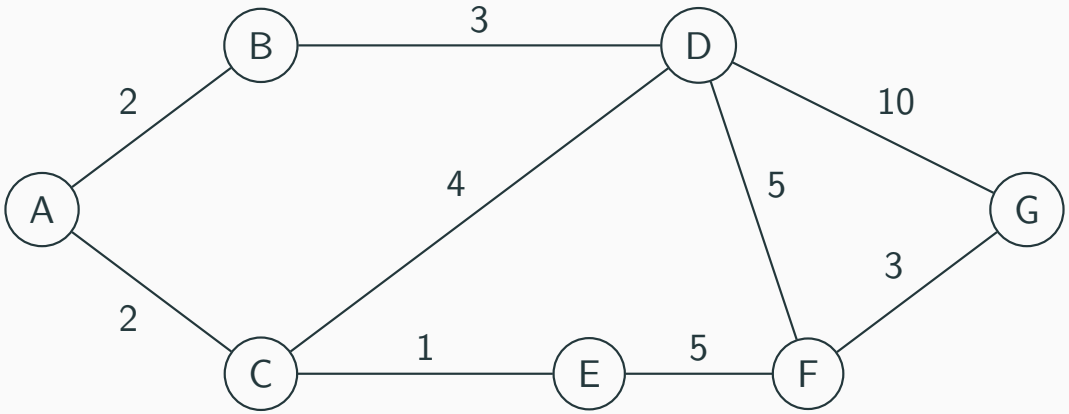
- La solución óptima a cualquier caso no trivial de un problema es una combinación de soluciones óptimas de algunos de los sub-casos.
- Si $d_1, d_2, d_3, \ldots, d_n$ es óptima para $P[1, n]$:
  - Entonces, $d_1, d_2, d_3, \ldots, d_i$ es óptima para $P[1, i]$.
  - Y, $d_{i+1}, \ldots, d_n$ es óptima para $P[i + 1, n]$.

Principio de Optimalidad: caminos mínimos



- Camino mínimo de A a G es ACEFG con valor 11.
- Otros caminos de A a G:
  - ACDFG (14).
  - ACDG (16).
  - ABDG (15).
  - ABDFG (13).

Principio de Optimalidad: caminos mínimos



- **Subcaminos del camino mínimo también son mínimos** para el subproblema correspondiente:
  - ACE (3) es el camino mínimo de A a E.
  - EFG (8) es el camino mínimo de E a G.
- **La composición de caminos mínimos** ABD y DFG **no es el camino mínimo** de A a G:
  - Camino mínimo de A a D es ABD (5).
  - Camino mínimo de D a G es DFG (8).

Principio de Optimalidad

- La dificultad de aplicar este principio está en que no suele ser evidente cuáles son los sub-casos relevantes para el caso considerado.
- Esto impide usar una aproximación similar a «divide y vencerás», comenzando en el caso original y buscando recursivamente soluciones óptimas para los sub-casos relevantes y solo para estos.
- En su lugar, la programación dinámica resuelve todos los sub-casos para determinar los que realmente son relevantes.
- Entonces, se combinan en una solución óptima para el caso original.

Esquema de diseño (1/2)

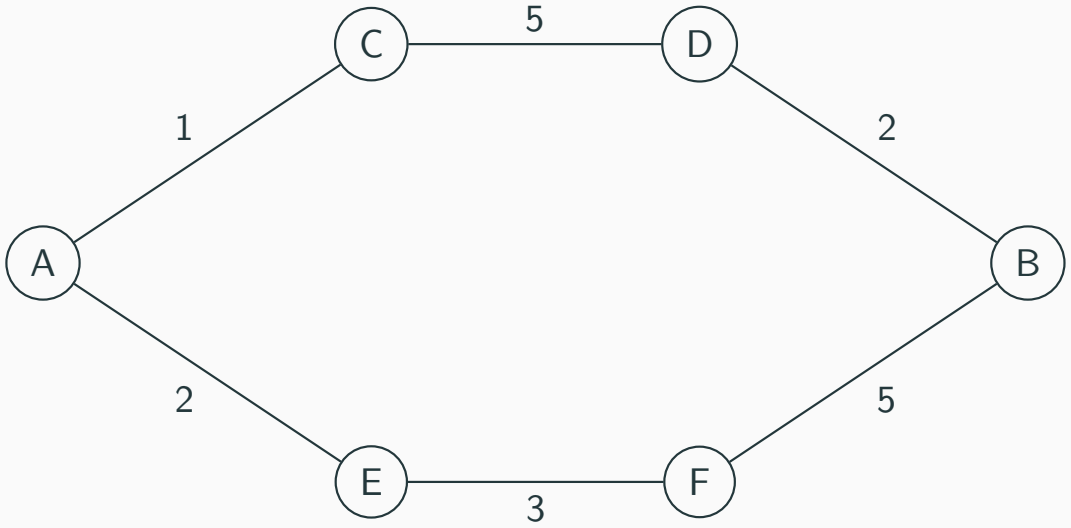
- Planteamiento de la solución como una sucesión de decisiones y verificación de que se cumple el Principio de Optimalidad:
  - Encontrar una estructura de la solución.
  - Dividir el problema en subproblemas y determinar si se puede aplicar el Principio de Optimalidad.
- Definición recursiva de la solución óptima:
  - Definir el valor de la solución óptima en función de valores de soluciones para subproblemas de tamaño menor.

Esquema de diseño (2/2)

- Calcular el valor de la solución óptima mediante un enfoque ascendente:
  - Determinar el conjunto de subproblemas distintos a resolver (tamaño de la tabla).
  - Identificar los subproblemas con solución trivial.
  - Obtener los valores con un enfoque ascendente y almacenar los que vamos calculando en la tabla.
  - En etapas posteriores, se usarán los valores previamente calculados.
- Determinar la solución óptima a partir de la información calculada previamente.

Incumplimiento del Principio de Optimalidad

- Cálculo del camino simple (sin ciclos) más largo entre dos vértices de un grafo:



- El camino más largo de A a B es AEFB (10), pero el subcamino AEF (5) no es el camino más largo de A a F (es ACDBF (13)).

El problema del cambio de monedas

El problema del cambio de monedas

Enunciado

- Suponiendo que el sistema monetario de un país está formado por monedas de valores $v_1, v_2, \ldots, v_n$, el problema del cambio de dinero consiste en descomponer cualquier cantidad dada M en monedas de ese país usando el menor número posible de monedas.

Ejemplo

- Un artículo de una tienda cuesta 5.37 euros.
- El cliente que lo compra paga con un billete de 10 euros.
- Hay que devolverle 4.63 euros.

¿Cómo se devuelve el cambio
con el menor número de monedas?

El problema del cambio de monedas

Algoritmo *greedy*

- Eficiente.
- No era capaz de encontrar el óptimo en todos los casos.

Ejemplo

- Existen monedas de 1, 4 y 6 céntimos.
- Hay que devolver 8 céntimos.
- El algoritmo *greedy* devuelve:
  - Una moneda de 6 céntimos.
  - Dos monedas de 1 céntimo.
- La solución óptima consiste en dos monedas de 4 céntimos.

Se puede mejorar con programación dinámica

El problema del cambio de monedas

Consideraciones

- Hay monedas de n valores diferentes.
- Las monedas de tipo i tiene un valor de $v_i > 0$ unidades.
- Hay un suministro ilimitado de monedas.
- Al cliente hay que devolverle un cambio de valor igual a M.
- Hay que utilizar el menor número de monedas.

El problema del cambio de monedas

Planteamiento de la solución como una sucesión de decisiones

- Se puede plantear la solución como una secuencia de decisiones $x_1, x_2, \ldots, x_n$:
  - Cuántas monedas de tipo 1.
  - Cuántas monedas de tipo 2.
  - ...
- Sea $C(i, j)$, $1 \leq i \leq n$, $0 \leq j \leq M$, el número mínimo de monedas para obtener la cantidad j restringiéndose a los tipos de moneda desde 1 hasta i.
- La solución al problema consiste en calcular $C(n, M)$.

Verificación del Principio de Optimalidad

- Si $x_1, \ldots, x_n$ es óptima para $C(n, M)$, entonces hay que demostrar que $x_1, \ldots, x_{n-1}$ es óptima para $C(n-1, M - x_n \cdot v_n)$.

El problema del cambio de monedas

Demostración por contradicción

- Si no fuese así, entonces existe $y_1, \ldots, y_{n-1}$ tal que:

$$\sum_{i=1}^{n-1} y_i < \sum_{i=1}^{n-1} x_i \quad \text{y} \quad \sum_{i=1}^{n-1} y_i \cdot v_i = M - x_n \cdot v_n$$

- Pero entonces, haciendo $y_n = x_n$ tenemos que:

$$\sum_{i=1}^{n} y_i \cdot v_i = M, \quad \text{luego } y_1, \ldots, y_n \text{ es una solución para } C(n, M).$$

Y, además:

$$\sum_{i=1}^{n} y_i < \sum_{i=1}^{n} x_i, \quad \text{luego } x_1, \ldots, x_n \text{ no sería óptima.}$$

El problema del cambio de monedas

Definición recursiva (1/2)

- Si no se puede conseguir $C(i, j)$, entonces $C(i, j) = \infty$.
- En cada paso, existen dos opciones:
  - No incluir ninguna moneda del tipo i. Esto supone que el valor de $C(i, j)$ va a coincidir con el de $C(i - 1, j)$. Por tanto:

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

  - Sí incluirla. Al hacerlo, el número de monedas global coincide con el número óptimo de monedas para una cantidad $(j - v_i)$ más esta moneda i que se incluye. Es decir:

$$C(i, j) = 1 + C(i, j - v_i)$$

- El cálculo de $C(i, j)$ óptimo tomará la solución más favorable (el menor valor de ambas).

El problema del cambio de monedas

Definición recursiva (2/2)

$$C(i,j) = \begin{cases} \infty, & \text{si } i = 1 \text{ y } 1 \leq j < v_i \\ 0, & \text{si } j = 0 \\ 1 + C(i, j - v_i), & \text{si } i = 1 \text{ y } j \geq v_i \\ C(i-1, j), & \text{si } i > 1 \text{ y } j < v_i \\ \min(C(i-1,j), 1 + C(i, j - v_i)), & \text{en otro caso} \end{cases}$$

- Se requiere una tabla para almacenar los resultados de los subproblemas.
- Se rellena la tabla de arriba a abajo y de izquierda a derecha.

El problema del cambio de monedas

Ejemplo

- La cantidad a devolver es $M = 8$ unidades.
- Existen $n = 3$ tipos de monedas.
- El valor de cada moneda es $v_1 = 1$, $v_2 = 4$, y $v_3 = 6$.
- Caso base: $C(i, 0) = 0$.

	0	1	2	3	4	5	6	7	8
$v_1 = 1$	0								
$v_2 = 4$	0								
$v_3 = 6$	0								

El problema del cambio de monedas

Ejemplo

- La cantidad a devolver es $M = 8$ unidades.
- Existen $n = 3$ tipos de monedas.
- El valor de cada moneda es $v_1 = 1$, $v_2 = 4$, y $v_3 = 6$.
- $C(1, j) = 1 + C(1, j - 1)$.

	0	1	2	3	4	5	6	7	8
$v_1 = 1$	0	1	2	3	4	5	6	7	8
$v_2 = 4$	0								
$v_3 = 6$	0								

El problema del cambio de monedas

Ejemplo

- La cantidad a devolver es $M = 8$ unidades.
- Existen $n = 3$ tipos de monedas.
- El valor de cada moneda es $v_1 = 1$, $v_2 = 4$, y $v_3 = 6$.
- $C(2, j) = \min(C(1, j), 1 + C(2, j - 4)$.

	0	1	2	3	4	5	6	7	8
$v_1 = 1$	0	1	2	3	4	5	6	7	8
$v_2 = 4$	0	1	2	3	1	2	3	4	2
$v_3 = 6$	0								

El problema del cambio de monedas

Ejemplo

- La cantidad a devolver es $M = 8$ unidades.
- Existen $n = 3$ tipos de monedas.
- El valor de cada moneda es $v_1 = 1$, $v_2 = 4$, y $v_3 = 6$.
- $C(3, j) = \min(C(2, j), 1 + C(3, j - 6)$.

	0	1	2	3	4	5	6	7	8
$v_1 = 1$	0	1	2	3	4	5	6	7	8
$v_2 = 4$	0	1	2	3	1	2	3	4	2
$v_3 = 6$	0	1	2	3	1	2	1	2	2

El problema del cambio de monedas

Ejemplo

- La cantidad a devolver es $M = 8$ unidades.
- Existen $n = 3$ tipos de monedas.
- El valor de cada moneda es $v_1 = 1$, $v_2 = 4$, y $v_3 = 6$.
- La solución óptima es $C(3, 8) = 2$ monedas.

	0	1	2	3	4	5	6	7	8
$v_1 = 1$	0	1	2	3	4	5	6	7	8
$v_2 = 4$	0	1	2	3	1	2	3	4	2
$v_3 = 6$	0	1	2	3	1	2	1	2	2

El problema del cambio de monedas

Algoritmo de programación dinámica

```
1  int cambio_monedas(int *v, int n, int m){
2    int c[n][m+1];
3    for (int i = 0; i < n; i++)
4      c[i][0] = 0;
5    for (int i = 0; i < n; i++){
6      for (int j = 1; j <= m; j++){
7        if (i == 0 && j < v[i])
8          c[i][j] = INT_MAX;
9        else if (i == 0)
10         c[i][j] = 1 + c[i][j - v[0]];
11       else if (j < v[i])
12         c[i][j] = c[i-1][j];
13       else
14         c[i][j] = min(c[i-1][j], 1+c[i][j-v[i]]);
15     }
16   }
17   return c[n-1][m];
```

- El tiempo de ejecución es $O(n \cdot M)$.

El problema del cambio de monedas

¿Cuántas monedas de cada tipo?

- Se empieza con $C(n, M)$: $i = n$, $j = M$.
- Si $C(i, j) = C(i-1, j)$, no se escoge una moneda de tipo i y se pasa a analizar $C(i-1, j)$: $i = i - 1$.
- Si $C(i, j) = 1 + C(i, j - v_i)$, se escoge una moneda de tipo i y se pasa a estudiar $C(i, j - v_i)$: `monedas[i]++`.
- Se continua hasta que se llega a algún $C(i, 0)$ y ya no quede nada por pagar.

	0	1	2	3	4	5	6	7	8
$v_1 = 1$	0	1	2	3	4	5	6	7	8
$v_2 = 4$	0	1	2	3	1	2	3	4	2
$v_3 = 6$	0	1	2	3	1	2	1	2	2

El problema de la mochila 0/1

El problema de la mochila 0/1

Enunciado

- Considérese una mochila capaz de albergar un peso máximo M y n elementos con pesos $p_1, p_2, \ldots, p_n$ y beneficios $b_1, b_2, \ldots, b_n$.
- Se trata de seleccionar de los n elementos, $x = (x_1, x_2, \ldots, x_n)$, con $x_i \in \{0, 1\}$, aquellos que deberán introducirse en la mochila para maximizar el beneficio.
- Se buscan los valores $x_1, x_2, \ldots, x_n$, con $x_i \in \{0, 1\}$, que maximizan la siguiente función objetivo:

$$f(x) = \sum_{i=1}^{n} b_i \cdot x_i$$

con la restricción:

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

El problema de la mochila 0/1

Consideraciones

- La capacidad máxima de la mochila es M.
- Hay n objetos.
- El beneficio de llevar cada objeto es b_i, $i = 1, \ldots, n$.
- El peso de cada objeto es p_i, $i = 1, \ldots, n$.
- Hay que maximizar $\sum_{i=1}^{n} x_i \cdot b_i$ sujeto a $\sum_{i=1}^{n} x_i \cdot p_i \leq M$.

El problema de la mochila 0/1

Planteamiento de la solución como una sucesión de decisiones

- Se puede plantear la solución como una secuencia de decisiones $x_1, x_2, \ldots, x_n$:
  - Seleccionar, o no, el objeto 1.
  - Seleccionar, o no, el objeto 2.
  - ...
- Sea $B(i, p)$, con $1 \leq i \leq n$ y $0 \leq p \leq M$, el valor máximo de la mochila con capacidad p cuando consideramos i objetos.
- La solución viene dada por $B(n, M)$.

Verificación del Principio de Optimalidad

- Si $x_1, \ldots, x_n$ es óptima para $B(n, M)$, entonces hay que demostrar que $x_1, \ldots, x_{n-1}$ es óptima para:
  - $B(n-1, M)$ si $x_n = 0$.
  - $B(n-1, M - p_i)$ si $x_n = 1$.

Demostración por contradicción (1/2)

- Caso $x_n = 0$. Si no fuese así, entonces existe $y_1, \ldots, y_{n-1}$ tal que:

$$\sum_{i=1}^{n-1} y_i \cdot b_i > \sum_{i=1}^{n-1} x_i \cdot b_i \qquad \text{y} \qquad \sum_{i=1}^{n-1} y_i \cdot p_i \leq M$$

- Pero entonces, haciendo $y_n = x_n = 0$, tenemos que:

$$\sum_{i=1}^{n} y_i \cdot p_i \leq M, \quad \text{luego } y_1, \ldots, y_n \text{ es una solución para } B(n, M)$$

Y, además:

$$\sum_{i=1}^{n} y_i \cdot b_i > \sum_{i=1}^{n} x_i \cdot b_i, \quad \text{luego } x_1, \ldots, x_n \text{ no sería óptima.}$$

Demostración por contradicción (2/2)

- Caso $x_n = 1$. Si $x_1, \ldots, x_{n-1}$ no es óptima para $B(n-1, M-p_i)$, entonces existe $y_1, \ldots, y_{n-1}$ tal que:

$$\sum_{i=1}^{n-1} y_i \cdot b_i > \sum_{i=1}^{n-1} x_i \cdot b_i \qquad \text{y} \qquad \sum_{i=1}^{n-1} y_i \cdot p_i \leq M - p_n$$

- Pero entonces, haciendo $y_n = x_n = 1$, tenemos que:

$$\sum_{i=1}^{n-1} y_i \cdot p_i + p_n \leq M, \quad \text{luego } y_1, \ldots, y_n \text{ es una solución para } B(n, M)$$

Y, además:

$$\sum_{i=1}^{n-1} y_i \cdot b_i + b_n > \sum_{i=1}^{n-1} x_i \cdot b_i + b_n, \quad \text{luego } x_1, \ldots, x_n \text{ no sería óptima.}$$

El problema de la mochila 0/1

Definición recursiva (1/2)

- La mejor selección de elementos del conjunto $1, 2, \ldots, i$ para una mochila de tamaño M se puede definir en función de selecciones de elementos de $1, 2, \ldots, i-1$, para mochilas de tamaño menor:
  - O bien es la ganancia para la mejor selección de elementos de $1, 2, \ldots, i-1$ con peso máximo p (no selecciono el objeto i).
  - O bien es la ganancia de la mejor selección de elementos de $1, 2, \ldots, i-1$ con peso máximo $p - p_i$ más la ganancia del elemento i, b_i (sí selecciono el objeto i).

El problema de la mochila 0/1

Definición recursiva (2/2)

$$B(i,p) = \begin{cases} 0, & \text{si } i = 0 \text{ o } p = 0 \\ B(i-1,p), & \text{si } p_i > p \\ \max(B(i-1,p), B(i-1,p-p_i) + b_i), & \text{en otro caso} \end{cases}$$

- Estos valores se van almacenando en una tabla de tamaño $(n+1) \cdot (M+1)$.

El problema de la mochila 0/1

Ejemplo

- Supongamos el siguiente ejemplo ($n = 4$, $M = 5$):

	1	2	3	4
p	2	1	3	2
b	12	10	20	15

	0	1	2	3	4	5
0						
1						
2						
3						
4						?

El problema de la mochila 0/1

Ejemplo

- Supongamos el siguiente ejemplo ($n = 4$, $M = 5$):

	1	2	3	4
p	2	1	3	2
b	12	10	20	15

- Casos base: $B(0, p) = B(i, 0) = 0$.

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					?

El problema de la mochila 0/1

Ejemplo

- Supongamos el siguiente ejemplo ($n = 4$, $M = 5$):

	1	2	3	4
p	2	1	3	2
b	12	10	20	15

- $B(1,1) = B(0,1) = 0$.

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0				
2	0					
3	0					
4	0					?

El problema de la mochila 0/1

Ejemplo

- Supongamos el siguiente ejemplo ($n = 4$, $M = 5$):

	1	2	3	4
p	2	1	3	2
b	12	10	20	15

- $B(1,2) = \max(B(0,2), 12 + B(0,2-2)) = \max(0, 12+0) = 12$.

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12			
2	0					
3	0					
4	0					?

El problema de la mochila 0/1

Ejemplo

- Supongamos el siguiente ejemplo ($n = 4$, $M = 5$):

	1	2	3	4
p	2	1	3	2
b	12	10	20	15

- $B(1,3) = \max(B(0,3), 12 + B(0,3-2)) = \max(0, 12+0) = 12$.

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12		
2	0					
3	0					
4	0					?

El problema de la mochila 0/1

Ejemplo

- Supongamos el siguiente ejemplo ($n = 4$, $M = 5$):

	1	2	3	4
p	2	1	3	2
b	12	10	20	15

- $B(1,4) = \max(B(0,4), 12 + B(0,4-2)) = \max(0, 12+0) = 12$.

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	
2	0					
3	0					
4	0					?

El problema de la mochila 0/1

Ejemplo

- Supongamos el siguiente ejemplo ($n = 4$, $M = 5$):

	1	2	3	4
p	2	1	3	2
b	12	10	20	15

- $B(1,5) = \max(B(0,5), 12 + B(0,5-2)) = \max(0, 12+0) = 12$.

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0					
3	0					
4	0					?

El problema de la mochila 0/1

Ejemplo

- Supongamos el siguiente ejemplo ($n = 4$, $M = 5$):

	1	2	3	4
p	2	1	3	2
b	12	10	20	15

- $B(2,1) = \max(B(1,1), 10 + B(1,1-1)) = \max(0, 10+0) = 10$.

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10				
3	0					
4	0					?

El problema de la mochila 0/1

Ejemplo

- Supongamos el siguiente ejemplo ($n = 4$, $M = 5$):

	1	2	3	4
p	2	1	3	2
b	12	10	20	15

- $B(2,2) = \max(B(1,2), 10 + B(1,2-1)) = \max(12, 10+0) = 12$.

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12			
3	0					
4	0					?

El problema de la mochila 0/1

Ejemplo

- Supongamos el siguiente ejemplo ($n = 4$, $M = 5$):

	1	2	3	4
p	2	1	3	2
b	12	10	20	15

- $B(2,3) = \max(B(1,3), 10 + B(1,3-1)) = \max(12, 10+12) = 22$.

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22		
3	0					
4	0					?

El problema de la mochila 0/1

Ejemplo

- Supongamos el siguiente ejemplo ($n = 4$, $M = 5$):

	1	2	3	4
p	2	1	3	2
b	12	10	20	15

- $B(2,4) = \max(B(1,4), 10 + B(1,4-1)) = \max(12, 10+12) = 22$.

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	
3	0					
4	0					?

El problema de la mochila 0/1

Ejemplo

- Supongamos el siguiente ejemplo ($n = 4$, $M = 5$):

	1	2	3	4
p	2	1	3	2
b	12	10	20	15

- $B(2,5) = \max(B(1,5), 10 + B(1,5-1)) = \max(12, 10+12) = 22$.

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0					
4	0					?

El problema de la mochila 0/1

Ejemplo

- Supongamos el siguiente ejemplo ($n = 4$, $M = 5$):

	1	2	3	4
p	2	1	3	2
b	12	10	20	15

- $B(3,1) = B(2,1) = 10$.

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10				
4	0					?

El problema de la mochila 0/1

Ejemplo

- Supongamos el siguiente ejemplo ($n = 4$, $M = 5$):

	1	2	3	4
p	2	1	3	2
b	12	10	20	15

- $B(3, 2) = B(2, 2) = 12$.

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12			
4	0					?

El problema de la mochila 0/1

Ejemplo

- Supongamos el siguiente ejemplo ($n = 4$, $M = 5$):

	1	2	3	4
p	2	1	3	2
b	12	10	20	15

- $B(3,3) = \max(B(2,3), 20 + B(2,3-3)) = \max(22, 20+0) = 22$.

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22		
4	0					?

El problema de la mochila 0/1

Ejemplo

- Supongamos el siguiente ejemplo ($n = 4$, $M = 5$):

	1	2	3	4
p	2	1	3	2
b	12	10	20	15

- $B(3,4) = \max(B(2,4), 20 + B(2,4-3)) = \max(22, 20+10) = 30$.

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	
4	0					?

El problema de la mochila 0/1

Ejemplo

- Supongamos el siguiente ejemplo ($n = 4$, $M = 5$):

	1	2	3	4
p	2	1	3	2
b	12	10	20	15

- $B(3,5) = \max(B(2,5), 20 + B(2,5-3)) = \max(22, 20+12) = 32$.

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0					?

El problema de la mochila 0/1

Ejemplo

- Supongamos el siguiente ejemplo ($n = 4$, $M = 5$):

	1	2	3	4
p	2	1	3	2
b	12	10	20	15

- $B(4, 1) = B(3, 1) = 10$.

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10				?

El problema de la mochila 0/1

Ejemplo

- Supongamos el siguiente ejemplo ($n = 4$, $M = 5$):

	1	2	3	4
p	2	1	3	2
b	12	10	20	15

- $B(4,2) = \max(B(3,2), 15 + B(3,2-2)) = \max(12, 15+0) = 15$.

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15			?

El problema de la mochila 0/1

Ejemplo

- Supongamos el siguiente ejemplo ($n = 4$, $M = 5$):

	1	2	3	4
p	2	1	3	2
b	12	10	20	15

- $B(4,3) = \max(B(3,3), 15 + B(3,3-2)) = \max(22, 15+10) = 25$.

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25		?

El problema de la mochila 0/1

Ejemplo

- Supongamos el siguiente ejemplo ($n = 4$, $M = 5$):

	1	2	3	4
p	2	1	3	2
b	12	10	20	15

- $B(4,4) = \max(B(3,4), 15 + B(3,4-2)) = \max(30, 15+12) = 30$.

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	?

El problema de la mochila 0/1

Ejemplo

- Supongamos el siguiente ejemplo ($n = 4$, $M = 5$):

	1	2	3	4
p	2	1	3	2
b	12	10	20	15

- $B(4,5) = \max(B(3,5), 15 + B(3,5-2)) = \max(32, 15+22) = 37$.

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

El problema de la mochila 0/1

Algoritmo de programación dinámica

```
1  int mochila(int *b, int *p, int n, int m){
2    int B[n+1][m+1];
3    for (int i = 0; i <= m; i++)
4      B[0][i] = 0;
5    for (int i = 0; i <= n; i++)
6      B[i][0] = 0;
7    for (int i = 1; i <= n; i++){
8      for (int j = 1; j <= m; j++){
9        if (p[i-1] <= j)
10         B[i][j] = max(B[i-1][j-p[i-1]]+b[i-1], B[i-1][j]);
11       else
12         B[i][j] = B[i-1][j];
13     }
14   }
15   return B[n][m];
16 }
```

El problema de la mochila 0/1

Búsqueda de la solución óptima

- Si $B(i, p) = B(i-1, p)$, entonces el objeto i no se selecciona y se pasa a estudiar el objeto $i-1$ para una mochila de capacidad p: Problema $B(i-1, p)$.
- Si $B(i, p) \neq B(i-1, p)$, entonces el objeto i se selecciona y se pasa a estudiar el objeto $i-1$ para una mochila de capacidad $p - p_i$: Problema $B(i-1, p-p_i)$.

El problema de la mochila 0/1

Ejemplo

- Supongamos el siguiente ejemplo ($n = 4$, $M = 5$):

	1	2	3	4
p	2	1	3	2
b	12	10	20	15

- Se seleccionan los objetos 4, 2 y 1.

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

Multiplicación encadenada de matrices

Multiplicación encadenada de matrices

Enunciado

- Se quiere calcular la matriz producto M de n matrices dadas, $M = M_1 \cdot M_2 \cdot \ldots \cdot M_n$.
- Por ser asociativa la multiplicación de matrices, existen muchas formas posibles de realizar esa operación, cada una con un coste asociado en términos del número de multiplicaciones escalares.
- Si cada M_i es de dimensión $d_{i-1} \times d_i$ $(1 \leq i \leq n)$, multiplicar $M_i \cdot M_{i+1}$ requiere $d_{i-1} \cdot d_i \cdot d_{i+1}$ operaciones.
- El problema consiste en encontrar el mínimo número de operaciones necesario para calcular el producto M.

Multiplicación encadenada de matrices

Ejemplo

- Para $n = 4$ y para las matrices M_1, M_2, M_3 y M_4, cuyos órdenes son:
  - M_1 es 30×1.
  - M_2 es 1×40.
  - M_3 es 40×10.
  - M_4 es 10×25.
- Hay cinco formas distintas de multiplicarlas, cuyos costes son:

$$
\begin{aligned}
((M_1 \cdot M_2) \cdot M_3) \cdot M_4 &= 30 \cdot 1 \cdot 40 + 30 \cdot 40 \cdot 10 + 30 \cdot 10 \cdot 25 &&= 20\,700 \\
M_1 \cdot (M_2 \cdot (M_3 \cdot M_4)) &= 40 \cdot 10 \cdot 25 + 1 \cdot 40 \cdot 25 + 30 \cdot 1 \cdot 25 &&= 11\,750 \\
(M_1 \cdot M_2) \cdot (M_3 \cdot M_4) &= 30 \cdot 1 \cdot 40 + 40 \cdot 10 \cdot 25 + 30 \cdot 40 \cdot 25 &&= 41\,200 \\
M_1 \cdot ((M_2 \cdot M_3) \cdot M_4) &= 1 \cdot 40 \cdot 10 + 1 \cdot 10 \cdot 25 + 30 \cdot 1 \cdot 25 &&= 1\,400 \\
(M_1 \cdot (M_2 \cdot M_3)) \cdot M_4 &= 1 \cdot 40 \cdot 10 + 30 \cdot 1 \cdot 10 + 30 \cdot 10 \cdot 25 &&= 8\,200
\end{aligned}
$$

Multiplicación encadenada de matrices

Fuerza bruta

- La enumeración de todas las parentizaciones posibles no proporciona un método eficiente.
- Sea P_n el número de parentizaciones de una sucesión de n matrices.
- Se puede dividir una sucesión de n matrices en dos (las i primeras y las $n-i$ siguientes) para cualquier $i = 1, 2, \ldots, n-1$. Entonces, al parentizar las dos subsucesiones resultantes independientemente, obtenemos:

$$P_n = \begin{cases} 1, & \text{si } n = 1 \\ \sum_{i=1}^{n-1} P_i \cdot P_{n-i}, & \text{si } n \geq 2 \end{cases}$$

Multiplicación encadenada de matrices

Fuerza bruta

- La solución de la ecuación recurrente anterior es la sucesión de los números de Catalan.
- Estos forman una secuencia de números naturales que aparece en varios problemas de conteo que habitualmente son recursivos (por ejemplo, el número de árboles binarios con $n+1$ hojas).

$$P_n = C_{n-1}$$

- Donde:

$$C_n = \frac{1}{n+1} \cdot \binom{2 \cdot n}{n} = \frac{(2 \cdot n)!}{(n+1)! \cdot n!} \quad \text{con } n \geq 0$$

- Asintóticamente, los números de Catalan crecen como $\frac{4^n}{n^{3/2} \cdot \sqrt{\pi}}$.
- Por tanto, el método de la fuerza bruta es una pobre estrategia para determinar la parentización óptima de una cadena de matrices.

Multiplicación encadenada de matrices

Planteamiento de la solución como una sucesión de decisiones

- Se puede plantear la solución como una secuencia de decisiones $x_1, x_2, \ldots, x_n$:
  - Dónde colocar el primer paréntesis.
  - Dónde colocar el segundo paréntesis.
  - ...
- Sea $N(i, j)$ el mínimo número de operaciones escalares para multiplicar las matrices $M_i \cdot \ldots \cdot M_j$.
- La solución que se busca es $N(1, n)$.

Multiplicación encadenada de matrices

Verificación del Principio de Optimalidad

- La solución óptima se puede definir en términos de soluciones óptimas a problemas de tamaño menor.
- Necesariamente tiene que haber una multiplicación final (la de mayor nivel) en el cálculo de la solución óptima.
- Si $(M_1 \cdot (M_2 \cdot M_3)) \cdot M_4$ es óptima para $M_1 \cdot M_2 \cdot M_3 \cdot M_4$, entonces $(M_1 \cdot (M_2 \cdot M_3))$ es óptima para $M_1 \cdot M_2 \cdot M_3$.
- Si hubiera una solución mejor para el subproblema, se podría haber usado en lugar de la anterior, lo que sería una contradicción sobre la optimalidad de $(M_1 \cdot (M_2 \cdot M_3)) \cdot M_4$.

Multiplicación encadenada de matrices

Definición recursiva (1/2)

- Sea $d_{i-1} \times d_i$ la dimensión de la matriz M_i.
- Se quiere multiplicar $(M_1 \cdot M_2 \cdot \ldots \cdot M_i \cdot M_{i+1} \cdot M_{i+2} \cdot \ldots \cdot M_n)$.
- Supongamos que parentizamos en i:

$$(M_1 \cdot M_2 \cdot \ldots \cdot M_i) \cdot (M_{i+1} \cdot M_{i+2} \cdot \ldots \cdot M_n)$$

- Si $N(i, j)$ el número de operaciones necesarias para multiplicar $M_i \cdot M_{i+1} \cdot \ldots \cdot M_j$, entonces:

$$N(1, n) = N(1, i) + N(i + 1, n) + d_0 \cdot d_i \cdot d_n$$

Definición recursiva (2/2)

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

- A partir de la definición recursiva se puede ver cómo se produce el solapamiento de los subproblemas.
- Primero se solucionan los subproblemas triviales (tamaño 0) para en cada paso ir resolviendo subproblemas de tamaño $1, 2, 3, \ldots$

Multiplicación encadenada de matrices

Ejemplo

- El número de matrices es $n = 4$.
- M_1 es 10×20.
- M_2 es 20×50.
- M_3 es 50×1.
- M_4 es 1×100.
- Caso base: $N(i, i) = 0$.

	1	2	3	4
1	0			
2		0		
3			0	
4				0

Multiplicación encadenada de matrices

Ejemplo

- El número de matrices es $n = 4$.
- M_1 es 10×20.
- M_2 es 20×50.
- M_3 es 50×1.
- M_4 es 1×100.
- $N(1,2) = N(1,1) + N(2,2) + d_0 \cdot d_1 \cdot d_2 = 0 + 0 + 10\,000 = 10\,000$.

	1	2	3	4
1	0	10 000		
2		0		
3			0	
4				0

Multiplicación encadenada de matrices

Ejemplo

- El número de matrices es $n = 4$.
- M_1 es 10×20.
- M_2 es 20×50.
- M_3 es 50×1.
- M_4 es 1×100.
- $N(2,3) = N(2,2) + N(3,3) + d_1 \cdot d_2 \cdot d_3 = 0 + 0 + 1\,000 = 1\,000$.

	1	2	3	4
1	0	10 000		
2		0	1 000	
3			0	
4				0

Multiplicación encadenada de matrices

Ejemplo

- El número de matrices es $n = 4$.
- M_1 es 10×20.
- M_2 es 20×50.
- M_3 es 50×1.
- M_4 es 1×100.
- $N(3,4) = N(3,3) + N(4,4) + d_2 \cdot d_3 \cdot d_4 = 0 + 0 + 5\,000 = 5\,000$.

	1	2	3	4
1	0	10 000		
2		0	1 000	
3			0	5 000
4				0

Multiplicación encadenada de matrices

Ejemplo

- El número de matrices es $n = 4$.
- M_1 es 10×20.
- M_2 es 20×50.
- M_3 es 50×1.
- M_4 es 1×100.
- $N(1,3) = \min(N(1,1) + N(2,3) + d_0 \cdot d_1 \cdot d_3, N(1,2) + N(3,3) + d_0 \cdot d_2 \cdot d_3) = \min(0 + 1\,000 + 200, 10\,000 + 0 + 500) = 1\,200$.

	1	2	3	4
1	0	10 000	1 200	
2		0	1 000	
3			0	5 000
4				0

Multiplicación encadenada de matrices

Ejemplo

- El número de matrices es $n = 4$.
- M_1 es 10×20.
- M_2 es 20×50.
- M_3 es 50×1.
- M_4 es 1×100.
- $N(2,4) = \min(N(2,2) + N(3,4) + d_1 \cdot d_2 \cdot d_4, N(2,3) + N(4,4) + d_1 \cdot d_3 \cdot d_4) = \min(0 + 5\,000 + 100\,000, 1\,000 + 0 + 2\,000) = 3\,000$.

	1	2	3	4
1	0	10 000	1 200	
2		0	1 000	3 000
3			0	5 000
4				0

Multiplicación encadenada de matrices

Ejemplo

- El número de matrices es $n = 4$.
- M_1 es 10×20.
- M_2 es 20×50.
- M_3 es 50×1.
- M_4 es 1×100.
- $N(1,4) = \min(N(1,1) + N(2,4) + d_0 \cdot d_1 \cdot d_4, N(1,2) + N(3,4) + d_0 \cdot d_2 \cdot d_4, N(1,3) + N(4,4) + d_0 \cdot d_3 \cdot d_4) = 2\,200$.

	1	2	3	4
1	0	10 000	1 200	2 200
2		0	1 000	3 000
3			0	5 000
4				0

Multiplicación encadenada de matrices

Algoritmo de programación dinámica

```
1  int matrices(int n, int *d){
2    int N[n][n], j, aux;
3    for (int i = 0; i < n; i++)
4      N[i][i] = 0;
5    for (int l = 1; l < n; l++){
6      for (int i = 0; i < n-l; i++){
7        j = i+l;
8        N[i][j] = INT_MAX;
9        for (int k = i; k < j; k++){
10         aux = N[i][k] + N[k+1][j] + d[i]*d[k+1]*d[j+1];
11         if (aux < N[i][j])
12           N[i][j] = aux;
13       }
14     }
15   }
16   return N[0][n-1];
17 }
```

Búsqueda de la ordenación óptima (1/2)

- En la posición $N(1, n)$ está el número mínimo de multiplicaciones escalares necesario. Se necesita calcular cuál es la ordenación óptima asociada.
- Usar una matriz auxiliar, `Mejor_i(1..n, 1..n)`, en la que se almacene el índice donde se alcanzó el mínimo (mejor valor de i) para cada subproblema.
- En el ejemplo anterior:

	1	2	3	4
1	-	1	1	3
2		-	2	3
3			-	3
4				-

Búsqueda de la ordenación óptima (1/2)

	1	2	3	4
1	-	1	1	3
2		-	2	3
3			-	3
4				-

- `Mejor_i(1,4) = 3`, luego los subproblemas son $M_{1..3}$ y $M_{4..4}$. Es decir, $(M_1 \cdot M_2 \cdot M_3) \cdot M_4$.
- `Mejor_i(1,3) = 1`, luego tenemos $M_{1..1}$ y $M_{2..3}$. Es decir, $M_1 \cdot (M_2 \cdot M_3)$.
- La parametrización óptima es $((M_1 \cdot (M_2 \cdot M_3)) \cdot M_4)$.

Caminos mínimos

Caminos mínimos

Enunciado

- Sea $G = (V, A)$ un grafo dirigido y ponderado con pesos no negativos. Se quiere calcular el camino más corto que une cada par de vértices del grafo.

Posibles soluciones

- Fuerza bruta (orden exponencial).
- Aplicar el algoritmo de Dijkstra para cada vértice.
- Algoritmo de Floyd (programación dinámica).

Planteamiento de la solución como una sucesión de decisiones

- Se puede plantear la solución como una secuencia de decisiones $x_1, x_2, \ldots, x_n$:
  - Cuál es el primer vértice del camino.
  - Cuál es el segundo vértice del camino.
  - ...
- Sea $D_k(i, j)$ el coste del camino más corto del vértice i al vértice j usando solo los k primeros vértices del grafo, $\{1, 2, \ldots, k\}$, como vértices intermedios. Si $k = 0$, no hay vértices intermedios.
- La solución al problema consiste en calcular $D_n(i, j)$, siendo n el número de vértices del grafo.

Verificación del Principio de Optimalidad

- Si el camino de coste mínimo del vértice i al vértice j pasa por el vértice k, entonces los caminos de coste mínimo de i a k y de k a j son también caminos de coste mínimo.
- Si no lo fueran, entonces se encontraría un camino de i a j mejor que el mínimo.
- Por tanto, se cumple el Principio de Optimalidad de Bellman.

Definición recursiva (1/2)

- Si el camino de coste mínimo del vértice i al vértice j usando solo los k primeros vértices no pasa por el vértice k:

$$D_k(i,j) = D_{k-1}(i,j)$$

- Si el camino de coste mínimo del vértice i al vértice j usando solo los k primeros vértices sí pasa por el vértice k:

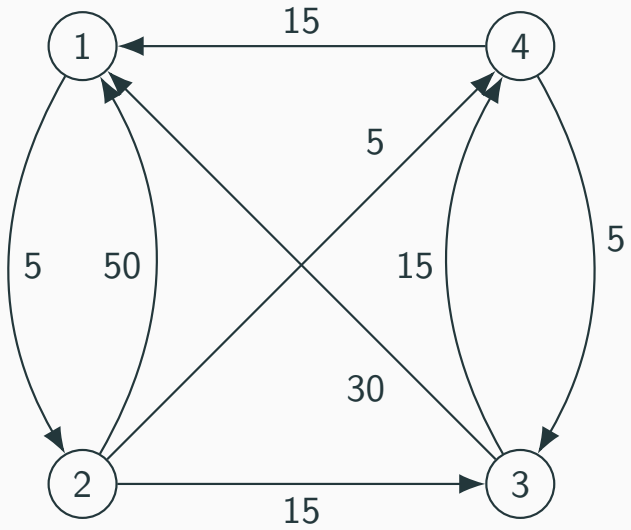
$$D_k(i,j) = D_{k-1}(i,k) + D_{k-1}(k,j)$$

Definición recursiva (2/2)

$$D_k(i,j) = \begin{cases} c_{ij}, & \text{si } k = 0 \\ \min(D_{k-1}(i,j), D_{k-1}(i,k) + D_{k-1}(k,j)), & \text{en otro caso} \end{cases}$$

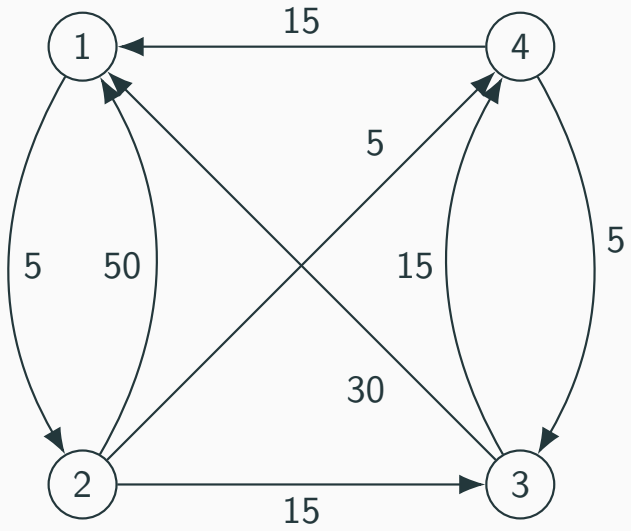
- Donde $C = (c_{ij})$ es la matriz de adyacencia del grafo, con $c_{ij} = \infty$ si no hay arco del vértice i al vértice j.

Ejemplo



$$D_0 = C = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & \infty & 0 & 15 \\ 15 & \infty & 5 & 0 \end{pmatrix}$$

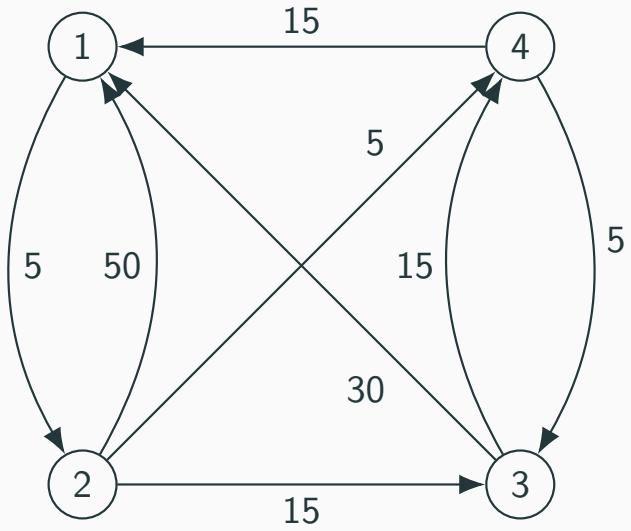
Ejemplo $k = 1$



- $D_1(3,2) = \min(D_0(3,2), D_0(3,1) + D_0(1,2)) = 35$
- $D_1(4,2) = \min(D_0(4,2), D_0(4,1) + D_0(1,2)) = 20$
- $D_1(4,3) = \min(D_0(4,3), D_0(4,1) + D_0(1,3)) = 5$

$$D_1 = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

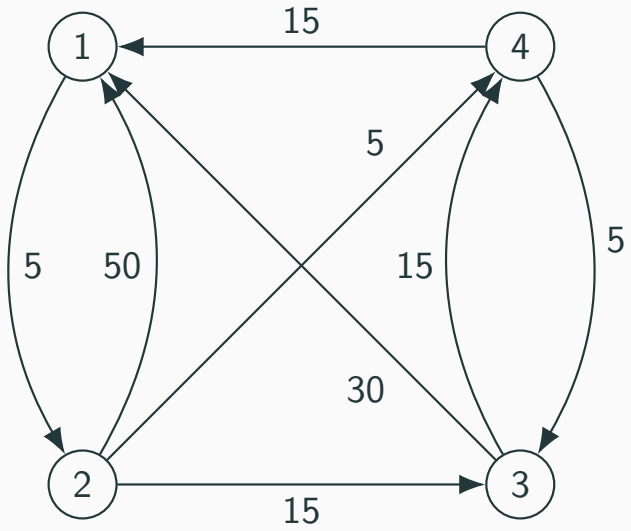
Ejemplo $k = 2$



- $D_2(1,3) = \min(D_1(1,3), D_1(1,2) + D_1(2,3)) = 20$
- $D_2(1,4) = \min(D_1(1,4), D_1(1,2) + D_1(2,4)) = 10$
- $D_2(4,3) = \min(D_1(4,3), D_1(4,2) + D_1(2,3)) = 5$

$$D_2 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

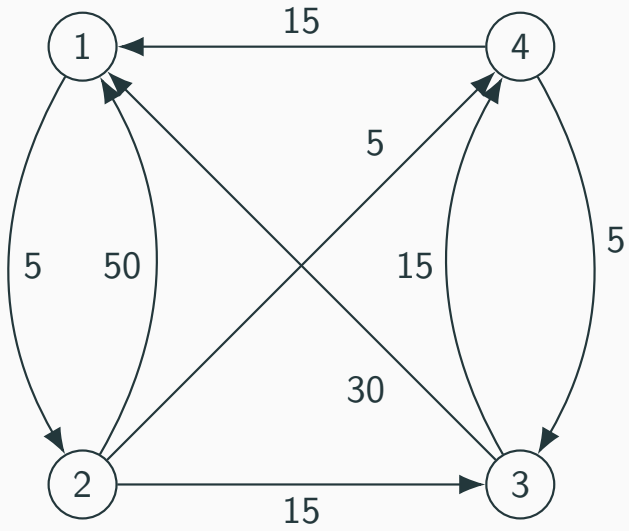
Ejemplo $k = 3$



- $D_3(2,1) = \min(D_2(2,1), D_2(2,3) + D_2(3,1)) = 45$

$$D_3 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ 45 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

Ejemplo $k = 4$



- $D_4(1,3) = \min(D_3(1,3), D_3(1,4) + D_3(4,3)) = 15$
- $D_4(2,1) = \min(D_3(2,1), D_3(2,4) + D_3(4,1)) = 20$
- $D_4(2,3) = \min(D_3(2,3), D_3(2,4) + D_3(4,3)) = 10$

$$D_4 = \begin{pmatrix} 0 & 5 & 15 & 10 \\ 20 & 0 & 10 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

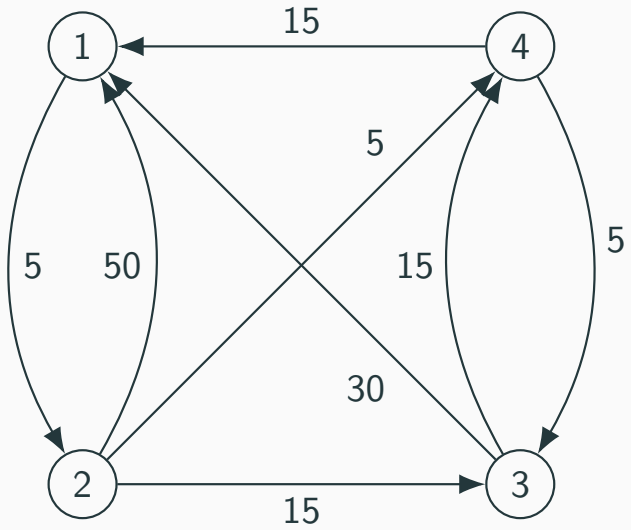
Algoritmo de programación dinámica

```
1  void floyd(int C[][N], int D[][N]){
2    for (int i = 0; i < N; i++)
3      for (int j = 0; j < N; j++)
4        D[i][j] = C[i][j];
5    for (int k = 0; k < N; k++)
6      for (int i = 0; i < N; i++)
7        for (int j = 0; j < N; j++)
8          if (D[i][k] + D[k][j] < D[i][j])
9            D[i][j] = D[i][k] + D[k][j];
10 }
```

Búsqueda de la solución óptima (1/2)

- Si además de conocer el coste del camino mínimo queremos conocer el camino en sí, se emplea otra matriz P y cuando se cumpla que $D(i,j) + D(k,j) < D(i,j)$, entonces $P(i,j) = k$.
- Si al terminar $P(i,j) = 0$, el camino es el directo de i a j.
- Si $P(i,j) = k$, entonces el camino pasa por k. Se analizan $P(i,k)$ y $P(k,j)$.

Búsqueda de la solución óptima (2/2)



- Camino de 1 a 3 es $P(1,3) = 4$.
- Camino de 1 a 4 y de 4 a 3: $P(1,4) = 2$ y $P(4,3) = 0$.
- Camino de 1 a 2 y de 2 a 4: $P(1,2) = 0$ y $P(2,4) = 0$.
- Camino mínimo de 1 a 3 es: 1-2-4-3.

$$P = \begin{pmatrix} 0 & 0 & 4 & 2 \\ 4 & 0 & 4 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

Bibliografía

Bibliografía

Aclaración

- El contenido de las diapositivas es esquemático y representa un apoyo para las clases teóricas.
- Se recomienda completar los contenidos del tema 1 con apuntes propios tomados en clase y con la bibliografía principal de la asignatura.

Por ejemplo


G. Brassard and P. Bratley.
***Fundamentals of Algorithmics*.**
Prentice Hall, Englewwod Cliffs, New Jersey, 1996.


J. L. Verdegay.
***Lecciones de Algorítmica*.**
Editorial Técnica AVICAM, 2017.