



UNIVERSIDADE DA CORUÑA

Algoritmos: Diseño de algoritmos por inducción

Alberto Valderruten

LFCIA - Departamento de Computación

Facultad de Informática

Universidad de A Coruña, España

www.lfcia.org/alg

www.fi.udc.es

Contenido

- Divide y Vencerás
- Programación Dinámica
 - Sucesión de Fibonacci
 - Coeficientes binomiales
 - Devolver el cambio
 - Problema de la mochila
 - Principio de optimalidad

Divide y Vencerás (1)

- **Descomponer** el caso a resolver en subcasos del mismo problema, resolverlos, independientemente entre sí (recursivamente), y **combinar** las soluciones de los subcasos para obtener la solución del caso original.
- **Ejemplos vistos:**
suma de la subsecuencia máxima (algoritmo *ssm3*), mergesort, quicksort, búsqueda binaria.
- **Esquema para la técnica:**
Consideremos:
 - un problema (ejemplo: ordenación)
 - un algoritmo *ad-hoc*, sencillo, capaz de resolver ese problema y *eficiente para casos pequeños del problema* (ejemplo: Inserción)Esquema: función Divide y Vencerás
- **Ejercicio:** contrastarla con los ejemplos vistos

Función Divide y Vencerás

```
función Divide y Vencerás (x) : solución
  si x es suficientemente pequeño entonces devolver ad-hoc(x)
  sino
    descomponer x en casos más pequeños  $x_1, x_2, \dots, x_a$ ;
    para  $i := 1$  hasta a hacer
       $y_i :=$  Divide y Vencerás ( $x_i$ );
    combinar los  $y_i$  para obtener una solución  $y$  de  $x$ ;
    devolver  $y$ 
fin función
```



Divide y Vencerás (2)

- Características de a (nº de subcasos):
 - pequeño
 - independiente de la entrada
- Caso particular: $a = 1 \Rightarrow$ *algoritmo de reducción*
 - el paso de recursivo a iterativo supondrá un ahorro en tiempo (constante) y en espacio (pila de ejecución, $\Omega(n)$).
 - Ejemplo: búsqueda binaria
- **Principales problemas:**
 - la descomposición y la combinación: ¿son posibles? ¿son eficientes?
 - subcasos: en lo posible del mismo tamaño: n/b , donde b constante $\neq a$
 - ¿umbral a partir del cual hay que utilizar el algoritmo ad-hoc?
- **Análisis:**
 - Reglas \Rightarrow ¿relación de recurrencia?
 - Comprobar las condiciones de aplicación del teorema Divide y Vencerás.

Programación Dinámica (1)

- **Divide y Vencerás** → riesgo de llegar a tener un gran número de subcasos idénticos \equiv ineficiencia!
- Ejemplo: **La sucesión de Fibonacci** [1202] - Leonardo de Pisa [1170-1240]

La sucesión se define inductivamente del modo siguiente:

$$\begin{cases} fib(0) = 0 \\ fib(1) = 1 \\ fib(n) = fib(n-1) + fib(n-2) \quad \text{si } n \geq 2 \end{cases}$$

\leftrightarrow sección áurea: $s_1 \geq s_2, s = s_1 + s_2$

s_1 : segmento áureo de $s \equiv s_1^2 = s \cdot s_2$

$\Rightarrow s_2$: segmento áureo de $s_1 \equiv s_2^2 = (s_1 - s_2) s_1$

\rightarrow ley de armonía (arquitectura, arte)...

Fibonacci 1

■ Algoritmo Fibonacci 1:

```
función fib1(n)
    si n < 2 entonces devolver n
    sino devolver fib1(n-1) + fib1(n-2)
fin función
```

- $T(n) = \Theta(\Phi^n)$, donde $\Phi = (1 + \sqrt{5})/2$
(Cf. resolución de recurrencias)
- $\rightarrow fib1(5)$ produce 3 llamadas a $fib1(0)$, 4 llamadas a $fib1(1)$,...
en total, 15 llamadas.

Programación Dinámica (2)

- **Programación Dinámica:** resolver cada subcaso una sólo vez, guardando las soluciones en una *tabla de resultados*, que se va completando hasta alcanzar la solución buscada.

⇒ Técnica ascendente, opuesta a la descendente de Divide y Vencerás.

- Ejemplo: **Algoritmo Fibonacci 2**

```
función fib2 (n)
  i := 1; j := 0;
  para k := 1 hasta n hacer
    j := i+j; i := j-i;
  devolver j
fin función
```

- $T(n) = \Theta(n)$ y espacio en $\Theta(1)$.

Fibonacci 3

- Algoritmo Fibonacci 3: $T(n) = O(\log n)$

función fib3 (n)

 i := 1; j := 0; k := 0; h := 1;

 mientras n > 0 hacer

 si n es impar entonces

 t := j*h;

 j := i*h + j*k + t;

 i := i*k + t

 t := h²;

 h := 2*k*h + t;

 k := k² + t;

 n := n div 2

 devolver j

fin función

$$\begin{pmatrix} fib(n+2) \\ fib(n+1) \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \times \begin{pmatrix} fib(n+1) \\ fib(n) \end{pmatrix}$$

$$\Rightarrow \begin{pmatrix} fib(n+1) \\ fib(n) \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n \times \begin{pmatrix} fib(1) \\ fib(0) \end{pmatrix}$$

Coeficientes binomiales (1)

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

- Ejemplo: Teorema de Newton

$$(1+x)^n = 1 + \binom{n}{1}x + \binom{n}{2}x^2 + \dots + \binom{n}{n-1}x^{n-1} + x^n$$

- **Problema:** Dados $0 \leq k \leq n \rightarrow i \binom{n}{k} ?$

- **Divide y Vencerás:**
función C(n, k): valor
 si k = 0 ó k = n entonces devolver 1
 sino devolver C(n-1, k-1) + C(n-1, k)
fin función

→ muchos cálculos se repiten \equiv suma de 1's (como en fib1) $\rightarrow \Omega\left(\binom{n}{k}\right)$

Coeficientes binomiales (2)

- Programación Dinámica:

→ Tabla de resultados intermedios: *triángulo de Pascal*

	0	1	2	...	$k - 1$	k
0	1					
1	1	1				
2	1	2	1			
...						
$n - 1$					$\binom{n - 1}{k - 1}$	$\binom{n - 1}{k}$
n						$\binom{n}{k}$

- $T(n) = \Theta(nk)$ y la complejidad espacial también.

- ¿Mejora? La complejidad espacial puede ser $\Theta(k)$
↔ manejar una sólo línea del triángulo de Pascal.

- Ejercicio: escribir el pseudocódigo.



Devolver el cambio (1)

- **Problema:** el algoritmo voraz es eficiente pero no “funciona” *siempre*: $M = \{1, 4, 6\}, n = 8 \Rightarrow \text{¿}S = \{4, 4\}$?
- Dado $M = \{v_1, v_2, \dots, v_m\}, v_i > 0$: denominaciones de las monedas
Objetivo: pagar exactamente n unidades de valor, con $|S|$ mínimo
Hipótesis: \exists suministro ilimitado de monedas
- Programación Dinámica \leftrightarrow tabla $c[1..m, 0..n]$
 $c[i, j] =$ n° mínimo de monedas para pagar j unidades de valor ($0 \leq j \leq n$)
utilizando monedas de denominación $v_1..v_i$ ($1 \leq i \leq m$).
- $|S| = c[m, n]$
- Construcción de la tabla:
 $c[i, 0] = 0$
 $c[i, j] = \min \begin{cases} c[i-1, j] & : \text{no utilizar una moneda más de } v_i & \leftrightarrow i > 1 \\ 1 + c[i, j - v_i] & : \text{utilizar una moneda más de } v_i & \leftrightarrow j \geq v_i \end{cases}$
Caso particular: $i = 1 \wedge j < v_1 \Rightarrow c[i, j] = +\infty \equiv$ no existe solución

Devolver el cambio (2)

```
función monedas (n): número de monedas
  const v[1..m]=[1,4,6]           {denominaciones de las monedas}
  {se construye una tabla c[1..m, 0..n]}
  para i := 1 hasta m hacer c [i,0] := 0;
  para i := 1 hasta m hacer
    para j := 1 hasta n hacer
      si i = 1 y j < v[i] entonces c[1,j] := infinito
      sino si i = 1 entonces c[1,j] := 1 + c[1, j-v[1] ]
      sino si j < v[i] entonces c[i,j] := c[i-1,j]
      sino c[i,j] := min ( c[i-1, j], 1 + c[i, j-v[i] ] )
  devolver c[m,n]
fin función
```

Devolver el cambio (3)

- **Ejemplo:** $M = \{1, 4, 6\}$, ¿c[3, 8]?

n	0	1	2	3	4	5	6	7	8
{1}	0	1	2	3	4	5	6	7	8
{1, 4}	0	1	2	3	1	2	3	4	2
{1, 4, 6}	0	1	2	3	1	2	1	2	2

- **Análisis:** $T(n) = \Theta(mn)$

- **Problema:** ¿Conjunto de monedas?

⇒ Algoritmo voraz sobre c : camino $c[m, n] \rightarrow c[0, 0]$

m “pasos” hacia arriba \equiv “no utilizar más v_i ”

$+c[m, n]$ “saltos” hacia la izquierda \equiv “utilizar una v_i más”

En total, $\Theta(m + c[m, n])$:

trabajo adicional a la construcción de la tabla

El problema de la mochila II (1)

- Versión II: los objetos **no** se pueden fraccionar $\equiv x_i \in \begin{cases} 0 & \equiv \text{dejar} \\ 1 & \equiv \text{tomar} \end{cases}$
- ¿Qué pasa con el algoritmo voraz?

Ejemplo: $n = 3, W = 9$:

Objetos	1	2	3	
v_i	9	7	5	
w_i	6	5	4	
v_i/w_i	1,5	1,4	1,25	Objetivo ($\sum_{i=1}^n x_i v_i$)
x_i (voraz)	1	0	0	9
x_i (óptimo)	0	1	1	12

\Rightarrow ¡Ha dejado de funcionar!

El problema de la mochila II (2)

- **Programación Dinámica** \leftrightarrow tabla $v[1..n, 0..W]$
 $v[i, j]$ = valor de la carga máxima para la capacidad j ($0 \leq j \leq W$)
considerando los objetos $1..i$ ($1 \leq i \leq n$).
- Construcción de la tabla:
$$v[i, j] = \max \begin{cases} v[i - 1, j] & : \text{no añadir el objeto } i \\ v[i - 1, j - w_i] + v_i & : \text{añadir el objeto } i \end{cases}$$
- **Observación:** a diferencia del caso de las monedas, cada objeto sólo se puede incluir “una vez” en la carga de la mochila.
- **Ejercicio:** ¿algoritmo?

El problema de la mochila II (3)

- **Ejemplo:**

	v_i	w_i	0	1	2	3	4	5	6	7	8	9
{1}	9	6	0	0	0	0	0	0	9	9	9	9
{1, 2}	7	5	0	0	0	0	0	7	9	9	9	9
{1, 2, 3}	5	4	0	0	0	0	5	7	9	9	9	12

- **Análisis:** $T(n) = \Theta(nW)$

- **Problema:** ¿Composición de la carga?

⇒ Recorrido sobre v : camino $v[n, W] \rightarrow v[0, 0]$

máximo n “pasos” hacia arriba \equiv “no incluir el objeto i ”

+ máximo W “saltos” hacia arriba y a la izquierda

\equiv “incluir el objeto i ”

En total, $\Theta(n + W)$: trabajo adicional a la construcción de la tabla

Programación Dinámica (conclusión)

- **Principio de optimalidad:**

La Programación Dinámica se utiliza para resolver *problemas de optimización* que satisfacen el principio de optimalidad:

“En una secuencia óptima de decisiones toda subsecuencia ha de ser también óptima”

- ¡No siempre es aplicable!

Ejemplo: Hallar el camino simple más largo entre dos nodos.



UNIVERSIDADE DA CORUÑA

Algoritmos: Diseño de algoritmos por inducción

Alberto Valderruten

LFCIA - Departamento de Computación

Facultad de Informática

Universidad de A Coruña, España

www.lfcia.org/alg

www.fi.udc.es