Programación dinámica

Alejandro Anzola Ávila

2024-2

Algoritmos y Estructuras de Datos – Grupo 4

Problema de *rod-cutting*

Formulación

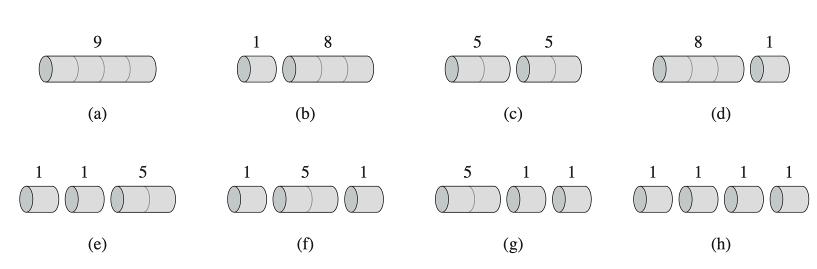
Dada una vara de longitud n y una tabla de precios p_i para $i = 1, 2, \ldots, n$, determine la máxima ganancia r_n obtenida de cortar la vara y vender las partes.

longitud i	1	2	3	4	5	6	7	8	9	10
precio p_i	1	5	8	9	10	17	17	20	24	30

Diseño

Existen 2^{n-1} maneras de cortar una vara de longitud n, podemos optar por cortar o *no* cortar, a una distancia i desde el extremo izquierdo.

Para una vara de longitud $n = 4$, tenemos $2^{n-1} = 2^{4-1} = 2^3 = 8$ maneras de cortar una vara.



Diseño: Notación

Denotemos la descomposición de la vara como:

$$n = i_1 + i_2 + \cdots + i_k$$

Para cortes en la longitud $i_1, i_2, \ldots, i_k$ respectivamente, donde $1 \leq k \leq n$.
Ejemplo: $7 = 2 + 2 + 3$, indica tres cortes: dos piezas de 2 unidades de longitud, y una pieza de longitud 3.

Para $n = i_1 + i_2 + \cdots + i_k$ si consideramos a r_n como la maxima ganancia, tenemos:

$$r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k}$$

longitud i	1	2	3	4	5	6	7	8	9	10
precio p_i	1	5	8	9	10	17	17	20	24	30

Para este ejemplo, encontremos los valores óptimos de r_i

$$
\begin{aligned}
r_1 &= 1 && \text{de la solución} && 1 = 1 \text{ (sin cortes)} \\
r_2 &= 5 && \text{de la solución} && 2 = 2 \text{ (sin cortes)} \\
r_3 &= 8 && \text{de la solución} && 3 = 3 \text{ (sin cortes)} \\
r_4 &= 10 && \text{de la solución} && 4 = 2 + 2 \\
r_5 &= 13 && \text{de la solución} && 5 = 2 + 3 \\
r_6 &= 17 && \text{de la solución} && 6 = 6 \text{ (sin cortes)} \\
r_7 &= 18 && \text{de la solución} && 7 = 1 + 6 \text{ ó } 7 = 2 + 2 + 3 \\
r_8 &= 22 && \text{de la solución} && 8 = 2 + 6 \\
r_9 &= 25 && \text{de la solución} && 9 = 3 + 6 \\
r_{10} &= 30 && \text{de la solución} && 10 = 10 \text{ (sin cortes)}
\end{aligned}
$$

Mas generalmente tenemos que

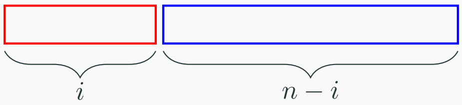
$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1)$$

Mas generalmente tenemos que

$$r_n = \text{máx}(\, p_n,\, r_1 + r_{n-1},\, r_2 + r_{n-2},\, \ldots,\, r_{n-1} + r_1 \,)$$

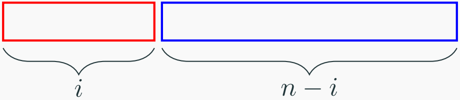
¿Existe una mejor forma de representar esto?

Podemos optar por simplificar el problema a solo resolver un subproblema:



Para un precio fijo p_i, y caso recursivo de r_{n-i}.

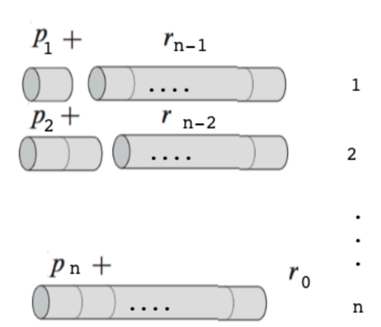
Podemos optar por simplificar el problema a solo resolver un subproblema:



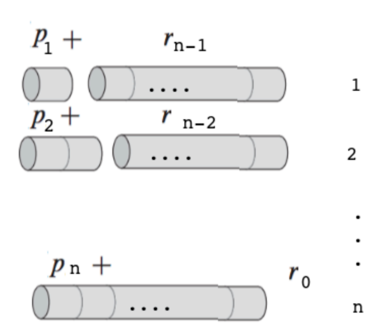
Para un precio fijo p_i, y caso recursivo de r_{n-i}.

¿Que estrategia de solución es esta?

$$r_n = \begin{cases} 0 & \text{si } n = 0 \\ \text{máx}_{1 \le i \le n}(p_i + r_{n-i}) & \text{si } n > 0 \end{cases}$$



$$r_n = \begin{cases} 0 & \text{si } n = 0 \\ \max_{1 \le i \le n}(p_i + r_{n-i}) & \text{si } n > 0 \end{cases}$$



Solo computar esto para $n = 40$ nos tomaría varios minutos encontrar la solución, ya que debemos verificar las 2^{n-1} posibilidades.

De formula a pseudo-código

$$r_n = \begin{cases} 0 & \text{si } n = 0 \\ \max_{1 \leq i \leq n}(p_i + r_{n-i}) & \text{si } n > 0 \end{cases}$$

CUT-ROD(p, n)

if n == 0
    return 0
$q = -\infty$
for $i = 1$ **to** n
    $q = \max(q, p[i] +$ CUT-ROD$(p, n - i))$
return q

CUT-ROD computa la máxima ganancia para una varilla de longitud n, con los precios $p[1 \,..\, N]$, $n \leq N$.

De formula a pseudo-código

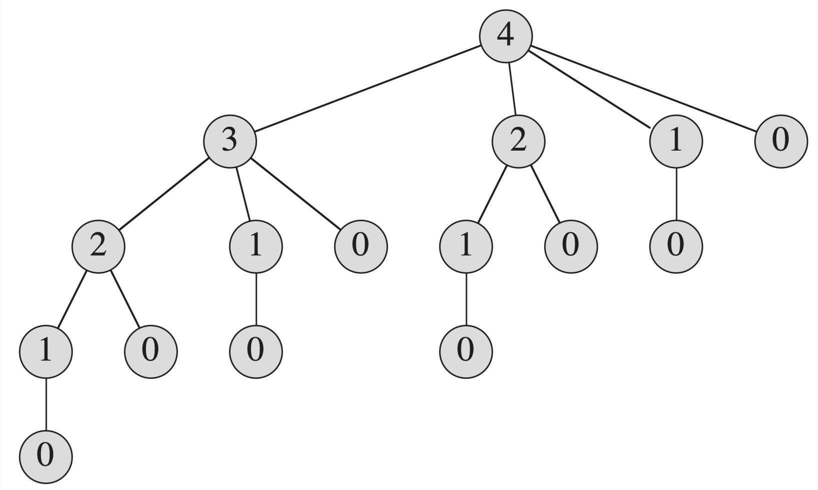
$$r_n = \begin{cases} 0 & \text{si } n = 0 \\ \max_{1 \le i \le n}(p_i + r_{n-i}) & \text{si } n > 0 \end{cases}$$

```
CUT-ROD(p, n)
if n == 0
    return 0
q = −∞
for i = 1 to n
    q = máx(q, p[i] + CUT-ROD(p, n − i))
return q
```

CUT-ROD computa la máxima ganancia para una varilla de longitud n, con los precios $p[1 \mathinner{..} N]$, $n \le N$.

¿Complejidad Θ?

Árbol de recursion para Cut-Rod



Para $n = 4$

Programación dinámica

Programación dinámica

La programación dinámica (*programación dinámica*) consiste en guardar respuestas de las soluciones ya encontradas, de manera que el computo de cada sub-problema se resuelva una sola vez.

Sacrificamos *espacio* para ahorrar en *tiempo* de ejecución.

Con esto podemos convertir soluciones de tiempo *exponencial* a soluciones de tiempo *polinomial*.

$$\Theta(b^n) \to \Theta(n^c)$$

Formas de implementar *programación dinámica*

Existen dos maneras de implementar *programación dinámica*:

1. Top-down con memorización
2. Bottom-up (tabulación)

Top-down con memorización

1. Se verifica si el sub-problema ya fue resuelto
2. Si es así, retornamos el valor guardado
3. Sino, se computa la solución del sub-problema normalmente, y se guarda el resultado

Podemos aplicar esta solución con Fibonacci.

$$f(n) = \begin{cases} 1 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ f(n-1) + f(n-2) & \text{si } n > 1 \end{cases}$$

Recursivo

$\textsc{Fib}(n)$

if $n == 0 \lor n == 1$
    return 1
return $\textsc{Fib}(n-1) + \textsc{Fib}(n-2)$

Podemos aplicar esta solución con Fibonacci.

$$f(n) = \begin{cases} 1 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ f(n-1) + f(n-2) & \text{si } n > 1 \end{cases}$$

Recursivo

$\textsc{Fib}(n)$

if $n == 0 \lor n == 1$
    return 1
return $\textsc{Fib}(n-1) + \textsc{Fib}(n-2)$

Recursivo con memorización

$\textsc{Fib}(m, n)$

if $m[n] \neq -\infty$
    return $m[n]$
if $n == 0 \lor n == 1$
    return 1
$m[n] = \textsc{Fib}(m, n-1) + \textsc{Fib}(m, n-2)$
return $m[n]$

Bottom-up (tabulación)

Consiste construir la solución desde el problema más pequeño, hasta resolver el problema completo.

Análogo a la estrategia *incremental*.

Hacemos en esencia una conversion de una solución recursiva a una solución iterativa.

Volvamos a Fibonacci

$$f(n) = \begin{cases} 1 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ f(n-1) + f(n-2) & \text{si } n > 1 \end{cases}$$

0	1	2	3	4	5	6	7	8	9
1	1								

$$f(n) = \begin{cases} 1 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ f(n-1) + f(n-2) & \text{si } n > 1 \end{cases}$$

```
Pre-Calculate-Fibonacci(m, n)
m[0] = 1
m[1] = 1
for i = 2 to n
    m[i] = m[i − 1] + m[i − 2]
```

```
Fib(m, n)
return m[n]
```

¿Complejidad Θ de cada uno?

Top-down con CUT-ROD

MEMOIZED-CUT-ROD(p, n)

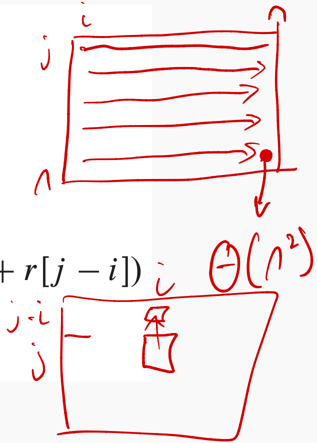
let $r[0\,..\,n]$ be a new array
for $i = 0$ **to** n
    $r[i] = -\infty$
return MEMOIZED-CUT-ROD-AUX(p, n, r)

Top-down con CUT-ROD

```
MEMOIZED-CUT-ROD(p, n)
let r[0..n] be a new array
for i = 0 to n
    r[i] = −∞
return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
if r[n] ≥ 0
    return r[n]
if n == 0
    q = 0
else q = −∞
    for i = 1 to n
        q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
r[n] = q
return q
```

```
BOTTOM-UP-CUT-ROD(p, n)
let r[0..n] be a new array
r[0] = 0
for j = 1 to n
    q = -∞
    for i = 1 to j
        q = max(q, p[i] + r[j - i])
    r[j] = q
return r[n]
```



Usa el orden natural de los subproblemas:

> *"Un problema de tamaño i es más pequeño que un subproblema de tamaño j si $i < j$"*

Entonces los problemas se resuelven en tamaños de $j = 0, 1, \ldots, n$ en ese orden.

Reconstruyendo una solución

Usualmente las soluciones de *programación dinámica* nos dan el valor de la mejor solución. Pero no nos da cual fue la solución.

Reconstruyendo una solución

Usualmente las soluciones de *programación dinámica* nos dan el valor de la mejor solución. Pero no nos da cual fue la solución.

- Para top-down podemos plantear que la recursion de como respuesta la solución como parte de la respuesta.

Reconstruyendo una solución

Usualmente las soluciones de *programación dinámica* nos dan el valor de la mejor solución. Pero no nos da cual fue la solución.

- Para top-down podemos plantear que la recursion de como respuesta la solución como parte de la respuesta.
- En bottom-up (tabulación) debemos reconstruirla.

Elementos de la programación dinámica

¿Como podemos reconocer que un problema se puede resolver con *programación dinámica*?

Elementos de la programación dinámica

¿Como podemos reconocer que un problema se puede resolver con *programación dinámica*? Existen dos ingredientes que nos puede indicar que es *programación dinámica*:

1. Subestructura óptima
2. Subproblemas superpuestos

Subestructura óptima

Un problema exhibe una *subestructura óptima* si una solución optima al problema contiene dentro soluciones optimas a subproblemas.

Subestructura óptima

Un problema exhibe una *subestructura óptima* si una solución optima al problema contiene dentro soluciones optimas a subproblemas.

1. La solución al problema consiste en tomar una *decisión*

Subestructura óptima

Un problema exhibe una *subestructura óptima* si una solución optima al problema contiene dentro soluciones optimas a subproblemas.

1. La solución al problema consiste en tomar una *decisión*
2. Para un problema dado, *existe* la decisión que nos llevará a una solución óptima.

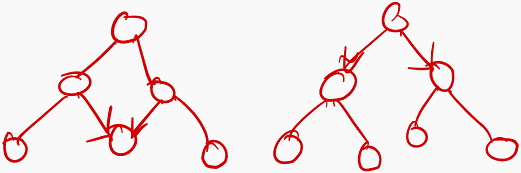
Subestructura óptima

Un problema exhibe una *subestructura óptima* si una solución optima al problema contiene dentro soluciones optimas a subproblemas.

1. La solución al problema consiste en tomar una *decisión*
2. Para un problema dado, *existe* la decisión que nos llevará a una solución óptima.
3. Entre otras consideraciones ...

En el problema de CUT-ROD una solución optima para cortar una varilla de tamaño n, usamos un solo subproblema de tamaño $n - i$, pero debemos considerar n decisiones sobre i para determinar la solución optima.

Subproblemas superpuestos



"Para un algoritmo recursivo, el problema resuelve los mismos subproblemas una y otra vez, en vez de siempre generar nuevos subproblemas"

Sub-secuencia común más larga

Formulación de secuencia

Definición de *subsecuencia*

Una subsecuencia de una secuencia es solo la secuencia original con cero o más elementos eliminados.

$$\langle A, B, C, D \rangle$$

$$\uparrow$$

$$\langle B, D \rangle$$

Formulación de secuencia

Definición formal de *subsecuencia*

Dada una secuencia $X = \langle x_1, \ldots, x_m \rangle$, otra secuencia $Z = \langle z_1, \ldots, z_k \rangle$ es subsecuencia de X si existe una secuencia estrictamente creciente $\langle i_1, i_2, \ldots, i_k \rangle$ de indices de X tales que para todo $j = 1, 2, \ldots k$, tenemos $x_{i_j} = z_j$.

Ejemplo:

1 2 3 4 5 6 7

$Z = \langle B, C, D, B \rangle$ es subsecuencia de $X = \langle A, B, C, B, D, A, B \rangle$ con indices $\langle 2, 3, 5, 7 \rangle$.

Formulación de subsecuencia común

Definición formal de subsecuencia común

Dadas dos secuencias X y Y, decimos que Z es una subsecuencia común de X y Y, si Z es una subsecuencia de X y Y.

Ejemplo:
Si $X = \langle A, B, C, B, D, A, B \rangle$ y $Y = \langle B, D, C, A, B, A \rangle$, la secuencia $\langle B, C, A \rangle$ es una subsecuencia común de X y Y.

Nótese que $\langle B, C, A \rangle$ no es la subsecuencia común más larga de X y Y, esa seria $\langle B, C, B, A \rangle$ con longitud 4.

El problema de LONGEST-COMMON-SUBSEQUENCE consiste en que, dadas dos secuencias $X = \langle x_1, \ldots, x_m \rangle$, y $Y = \langle y_1, \ldots, y_n \rangle$, queremos encontrar la longitud de la subsecuencia común más larga de X y Y.

Z sc X y Y

Diseño: fuerza bruta

Podemos enumerar todas las subsecuencias de X y verificar que cada subsecuencia que veamos también sea subsecuencia de Y.

Ya que existen 2^m subsecuencias de X posibles, esta solución seria $\Theta(2^m)$.

Diseño: formulación de prefijo

LCS exhibe una subestructura óptima.

Los subproblemas corresponden a pares de ‘prefijos’ de dos secuencias de entrada.

Diseño: formulación de prefijo

LCS exhibe una subestructura óptima.

Los subproblemas corresponden a pares de 'prefijos' de dos secuencias de entrada.

Dada una secuencia $X = \langle x_1, \ldots, x_m \rangle$, definimos el i-ésimo prefijo de X, para $i = 0, 1, \ldots, m$ como

$$X_i = \langle x_1, \ldots, x_i \rangle$$

donde X_0 es la secuencia vacía.

Diseño: formulación de prefijo

LCS exhibe una subestructura óptima.

Los subproblemas corresponden a pares de 'prefijos' de dos secuencias de entrada.

Dada una secuencia $X = \langle x_1, \ldots, x_m \rangle$, definimos el i-ésimo prefijo de X, para $i = 0, 1, \ldots, m$ como

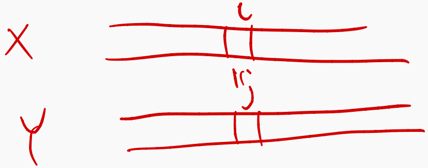
$$X_i = \langle x_1, \ldots, x_i \rangle$$

donde X_0 es la secuencia vacía.

Ejemplo:
Si $X = \langle A, B, C, B, D, A, B \rangle$, entonces $X_4 = \langle A, B, C, B \rangle$.

Podemos subdividir el problema en diferentes casos: $f(X_{i-1}, Y_{j-1}) + 1$

- Si $x_i = y_j$, entonces debemos encontrar un LCS en X_{i-1} y Y_{j-1}

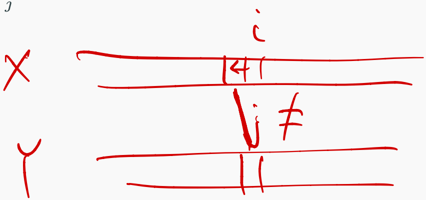


$+1$

Diseño: recursion

Podemos subdividir el problema en diferentes casos:

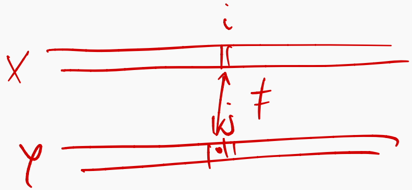
- Si $x_i = y_j$, entonces debemos encontrar un LCS en X_{i-1} y Y_{j-1}
- Si $x_i \neq y_j$, entonces debemos encontrar un LCS para
  - X_{i-1} y Y_j



Diseño: recursion

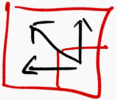
Podemos subdividir el problema en diferentes casos:

- Si $x_i = y_j$, entonces debemos encontrar un LCS en X_{i-1} y Y_{j-1}
- Si $x_i \neq y_j$, entonces debemos encontrar un LCS para
  - X_{i-1} y Y_j
  - X_i y Y_{j-1}



Podemos subdividir el problema en diferentes casos:

- Si $x_i = y_j$, entonces debemos encontrar un LCS en X_{i-1} y Y_{j-1}
- Si $x_i \neq y_j$, entonces debemos encontrar un LCS para
  - X_{i-1} y Y_j
  - X_i y Y_{j-1}



$$c[i,j] = \begin{cases} 0 & \text{si } i = 0 \vee j = 0 \\ c[i-1, j-1] + 1 & \text{si } i, j > 0 \wedge x_i = y_j \qquad = \\ \text{máx}(c[i, j-1], c[i-1, j]) & \text{si } i, j > 0 \wedge x_i \neq y_j \qquad \neq \end{cases}$$

BDCABA
ABCBDAB

BCBA

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0	0	0	1	0	1
2	B	0	1	0	0	1	1	1
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	3	3
5	D	0	1	2	2	2	3	3
6	A	0	1	2	2	3	3	4
7	B	0	1	2	2	3	4	4

Ejemplo de Cormen

i	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Ejercicios

Considere una mochila capaz de albergar un peso máximo M, y n elementos con pesos $P = \langle p_1, \ldots, p_n \rangle$, y beneficios $B = \langle b_1, \ldots, b_n \rangle$.

Queremos encontrar el *máximo* beneficio con pesos de los elementos que no superen el peso máximo de la mochila.

$$f(m, i) = \begin{cases} 0 & \text{si } m \leq 0 \lor i < 0 \\ f(m, i-1) & \text{si } m < p[i] \land i \geq 0 \\ \max\left[f(m-p[i], i-1) + b[i],\ f(m, i-1)\right] & \text{si } m \geq p[i] \land i \geq 0 \end{cases}$$

1. Problema de KNAPSACK (mochila)

¿Cual es el resultado de ejecutar con $M = 15$kg, $P = \langle 12\text{kg},\ 2\text{kg},\ 1\text{kg},\ 1\text{kg},\ 4\text{kg}\rangle$, $B = \langle 4, 2, 2, 1, 10\rangle$?

010

	P	12kg	2kg	1kg	1kg	4kg
m	B	4	2	2	1	10
15		0 ✗	0 ✗	0 ✗	0 ✗	0 ✗
14				2 ✓	1 ✓	
13			2 ✓			
12						
11			10 ←	10 ←	10 ←	10 ✓
10			12 ←	12	11	
9		14 ←	14 ✓			
8						
7						
6						
5						
4						
3		4 ✓				
2						
1						
0						

$$f(m,i) = \begin{cases} \text{máx}(f(m-p_i,\ i-1) + b_i,\ f(m,i-1)) & \text{si } m \geq p_i \wedge i > 0 \\ f(m,i-1) & \text{si } m < p_i \wedge i > 0 \\ 0 & \text{si } m \leq 0 \vee i \leq 0 \end{cases}$$

La solución optima es $f(M,n)$.

1. Implemente el algoritmo en base a la formula recursiva
   a. Sin memorización
   b. Con memorización
   c. Compare los tiempos de ejecución de cada uno para una entrada suficientemente grande.
2. $\star$ Implemente la solución optimizada con tabulación.

1. Determine el LCS de $\langle 1, 0, 0, 1, 0, 1, 0, 1\rangle$ y $\langle 1, 0, 0, 1, 0, 1, 0, 1\rangle$
2. Implemente la version optimizada con *programación dinámica* de LONGEST-COMMON-SUBSEQUENCE.
3. $\star$ Implemente la version optimizada con tabulación de LONGEST-COMMON-SUBSEQUENCE.

1. Modifique Memoized-Cut-Rod para que no solo retorne el valor, sino también la solución (las longitudes de la varilla).