Programación dinámica

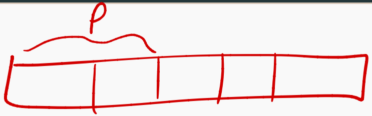
Alejandro Anzola Ávila

2024-2

Algoritmos y Estructuras de Datos – Grupo 4

Problema de *rod-cutting*

Formulación



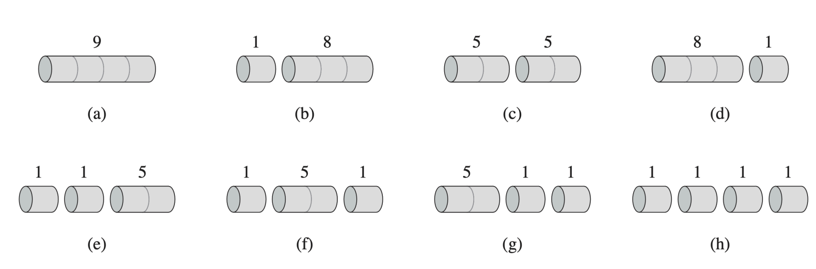
Dada una vara de longitud n y una tabla de precios p_i para $i = 1, 2, \ldots, n$, determine la máxima ganancia r_n obtenida de cortar la vara y vender las partes.

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

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

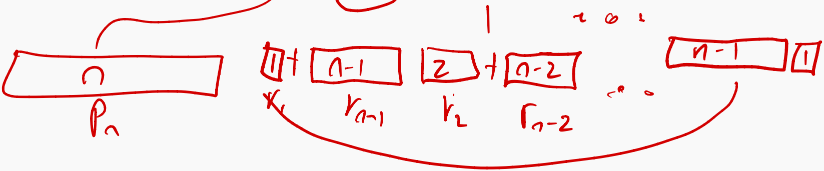
2 2
$\boxed{5}\boxed{5} = 10$
$5 + 8 = 13$

$9 + 1 = 10$

$\boxed{9}\boxed{1} = 25$
24 1

Mas generalmente tenemos que

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

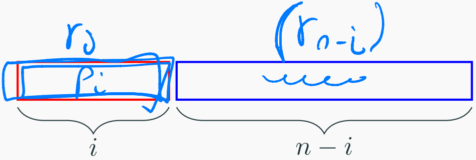


Mas generalmente tenemos que

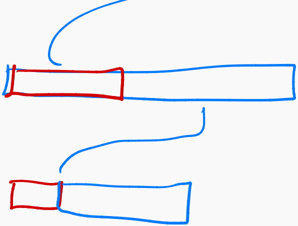
$$r_n = \max(\, p_n,\, r_1 + r_{n-1},\, r_2 + r_{n-2},\, \ldots,\, r_{n-1} + r_1 \,)$$

¿Existe una mejor forma de representar esto?

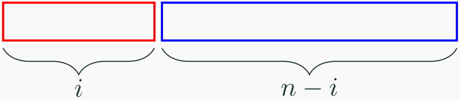
Podemos optar por simplificar el problema a solo resolver un subproblema:



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



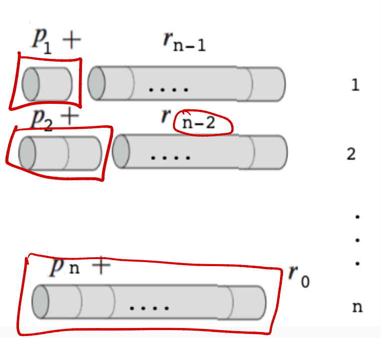
Podemos optar por simplificar el problema a solo resolver un subproblema:



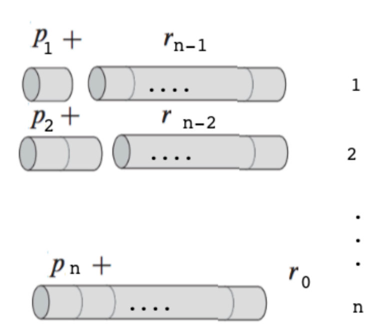
Para un precio fijo p_i, y caso recursivo de r_{n-i}.

¿Que estrategia de solución es esta?

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



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



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

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

```
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 $p[1\,..\,N]$, $n \le N$.

De formula a pseudo-código

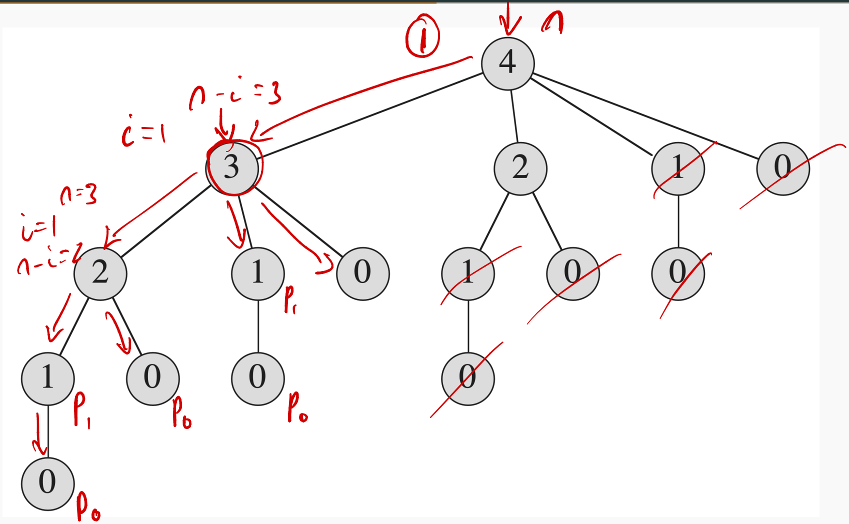
$$r_n = \begin{cases} 0 & \text{si } n = 0 \\ \max_{1 \le i \le n}(p_i + r_{n-i}) & \text{si } n > 0 \end{cases}$$

```
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 $p[1\,..\,N]$, $n \le N$.

¿Complejidad Θ?

$\Theta(2^n)$



Para $n = 4$

Programación dinámica

La programación dinámica (DP) 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*.

$$\Theta(b^n) \to \Theta(n^c) \qquad c \geq 1$$

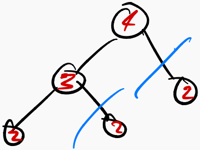
Formas de implementar DP

Existen dos maneras de implementar DP:

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

Top-down con memorización

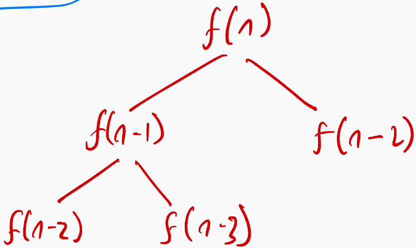
1. Se verifica si el sub-problema ya fue resuelto
2. Si es así, retornamos el valor guardado
3. Sino, se computa la solución del sub-problema normalmente, y se guarda el resultado



Podemos aplicar esta solución con Fibonacci.

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

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

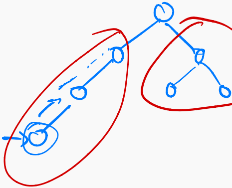


Recursivo

FIB(n)

if $n == 0 \vee n == 1$
    return 1
return FIB($n-1$) + 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

$\text{FIB}(n)$

```
if n == 0 ∨ n == 1
        return 1
return FIB(n − 1) + FIB(n − 2)
```

Recursivo con memorización

$\text{FIB}(m, n)$

```
if m[n] ≠ −∞                          1.
        return m[n]
if n == 0 ∨ n == 1                    Θ(1)
        return 1
m[n] = FIB(m, n − 1) + FIB(m, n − 2)
return m[n]
```

Bottom-up (tabulación)

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

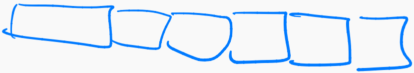
Análogo a la estrategia *incremental*.

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
M	1	1	2	3	5	8	13	?		



$\Theta(2^n)$

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

w

$= w \cdot 2^n$

m

w

PRE-CALCULATE-FIBONACCI(m, n)

```
m[0] = 1
m[1] = 1
for i = 2 to n
        m[i] = m[i − 1] + m[i − 2]
```

$\Theta(n)$

FIB(m, n)

```
return m[n]
```

$\Theta(1)$

$\Theta(n + w)$

¿Complejidad Θ de cada uno?

$\Theta(2^n) \rightarrow O(1)$

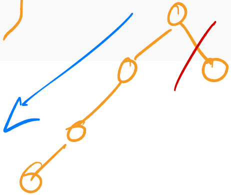
Top-down con CUT-ROD

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



p_i r_{n-i}

Ejercicios

1. Implemente Fibonacci de manera recursiva con y sin memorización
2. Compare los tiempos de ejecución entre cada uno con un valor de n lo suficientemente grande.

2. Problema de KNAPSACK (mochila)

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.

M = [None, None, None, ... None]

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

P = 12 2 1 1 4

B = 4 2 2 1 10

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. ¿Cual es el resultado de ejecutar con $M = \boxed{15\text{kg}}$, $P = \langle 12\text{kg}, 2\text{kg}, 1\text{kg}, 1\text{kg}, 4\text{kg} \rangle$, $B = \langle 4, 2, 2, 1, 10 \rangle$?
3. $\star$ Implemente la solución optimizada con tabulación.

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.

M

$= [0\ 0 \ldots]^M$ $M \times n$

m=[None, None,]

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

n

La solución optima es $f(M, n)$. (m, i) mem[m][i] = None/$-\infty$

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. ¿Cual es el resultado de ejecutar con $M = 15$kg, $P = \langle 12\text{kg}, 2\text{kg}, 1\text{kg}, 1\text{kg}, 4\text{kg} \rangle$, $B = \langle 4, 2, 2, 1, 10 \rangle$?
3. ⋆ Implemente la solución optimizada con tabulación.