

# **TEMA 9: PROGRAMACIÓN DINÁMICA**

COMPUTABILIDAD Y ALGORITMIA

M. Colebrook Santamaría

J. Riera Ledesma

J. Hernández Aceituno

# Objetivos

- Intro
- Ejemplo: coeficiente binomial
- Devolver cambio
- El Problema de la Mochila
- El Principio de Optimalidad y la Memoización

# Intro (1)

- En la sección anterior de Divide y Vencerás, hemos visto que es posible **dividir los problemas en subproblemas**, y **combinar sus soluciones** para resolver el problema principal.
- A veces se consideran **subproblemas solapados**.
- Si esto ocurre, se puede aprovechar para resolver cada subproblema una sola vez, guardando la solución para su uso posterior, lo que generará un **algoritmo más eficiente**.

# Intro (2)

- Se desea evitar calcular dos veces el mismo resultado, manteniendo una tabla de resultados conocidos que se llene a medida que se resuelven los subproblemas.
- Al contrario que el método de refinamiento progresivo aplicado en Divide y Vencerás, la **Programación Dinámica** (*Dynamic Programming*) es una **técnica ascendente** que permite encontrar la solución óptima a un problema: se comienza con los subproblemas más **sencillos** y, combinando sus soluciones, se obtienen las respuestas a los subproblemas de **tamaños más grandes**, hasta llegar a la **solución del problema original**.

# Ejemplo: Coeficiente binomial (1)

- Vamos a considerar el siguiente problema del cálculo del coeficiente binomial:

$$C(n, k) = \binom{n}{k} = \frac{n!}{k! \cdot (n-k)!} = \begin{cases} 1 & \text{si } k = 0 \text{ o } k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{si } 0 < k < n \\ 0 & \text{en otro caso} \end{cases}$$

- Si  $0 \leq k \leq n$ , podemos calcular  $C(n, k)$  directamente:

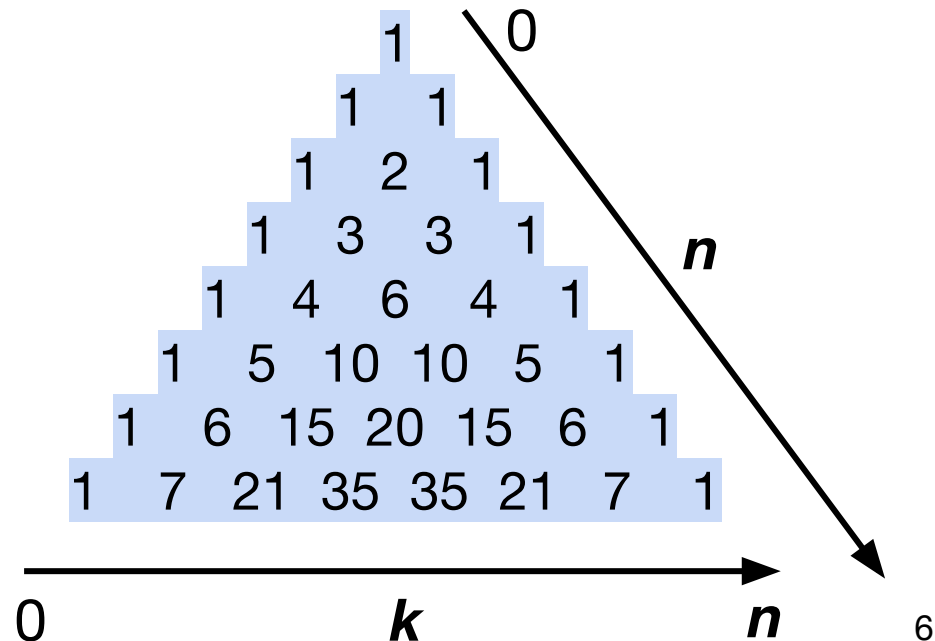
```
función C(n, k): valor {  
    si k = 0 o k = n entonces devolver 1  
    en otro caso devolver C(n-1, k-1) + C(n-1, k)  
}
```

# Ejemplo: Coeficiente binomial (2)

- Muchos de los valores de  $C(i, j)$  con  $i < n$ ,  $j < k$  se calculan una y otra vez. Por ejemplo,

$$C(5,3) = C(4,2) + C(4,3) \rightarrow \begin{cases} C(4,2) = C(3,1) + C(3,2) \\ C(4,3) = C(3,2) + C(3,3) \end{cases}$$

- Si utilizamos una tabla de resultados intermedios obtendremos un algoritmo más eficiente (Triángulo de Pascal).
- La tabla debería rellenarse **línea por línea**.



# Ejemplo: Coeficiente binomial (3)

```
función C(n, k): valor {  
  si k = 0 o k = n entonces devolver 1  
  en otro caso {  
    // Sea A una matriz de tamaño (n+1)x(n+1)  
    para i <- 0 hasta n hacer  
      A[i,0] <- A[i,i] <- 1 // Lados del triángulo  
  
    para i <- 2 hasta n hacer // Interior del triángulo  
      para j <- 1 hasta i-1 hacer  
        A[i, j] <- A[i-1, j-1] + A[i-1, j]  
  
    devolver A[n, k]  
  } }  
}
```

# Ejemplo: Coeficiente binomial (4)

■ Ejemplo de ejecución:  $C(7, 5)$

$n \setminus k$	0	1	2	3	4	5	6	7
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		
6	1	6	15	20	15	6	1	
7	1	7	21	35	35	21	7	1



# Ejemplo: Coeficiente binomial (5)


- El tiempo y el espacio necesarios para rellenar toda la tabla son de orden  $T(n) = 1 + 2 + 3 + \dots + n = n \cdot (n + 1) / 2 = \Theta(n^2)$
- Sin embargo, no es necesario rellenar toda la tabla: basta con mantener un vector de tamaño  $k$  que representa la línea actual y actualizarlo de derecha a izquierda.
- De este modo, calcular  $C(n,k)$  el algoritmo requiere un tiempo de  $\Theta(n \cdot k)$  y un espacio de  $\Theta(k)$ .

# Ejemplo: Coeficiente binomial (6)

```
función C(n, k): valor {  
  si k = 0 o k = n entonces devolver 1  
  en otro caso {  
    // Sea A un vector de tamaño k+1  
    A[0] = 1  
    para j <- 1 hasta n hacer // Primera fila  
      A[j] <- 0  
  
    para i <- 1 hasta n hacer // Cada iteración es una fila  
      para j <- min(i,k) hasta 1 hacer  
        A[j] <- A[j] + A[j-1] // De derecha a izquierda  
  
    devolver A[k]  
  }  
}
```

# Ejemplo: Coeficiente binomial (7)

- Ejemplo de ejecución:  $C(7, 5)$



iteración \ k	0	1	2	3	4	5
0	1	0	0	0	0	0
1	1	1	0	0	0	0
2	1	2	1	0	0	0
3	1	3	3	1	0	0
4	1	4	6	4	1	0
5	1	5	10	10	5	1
6	1	6	15	20	15	6
7	1	7	21	35	35	21

# Devolver cambio (1)

- El problema del cambio (introducido en una sección anterior) consiste en desarrollar un algoritmo para pagar una cierta cantidad empleando el menor número de monedas posible.
- Ya se describió un **algoritmo voraz** que funciona solamente en un número limitado de casos. Con ciertos sistemas monetarios o cuando el número o tipo de monedas es limitado, el algoritmo puede encontrar una respuesta que **no sea óptima** o **no encontrarla**.
- Por ejemplo, si tenemos  $M=\{1, 4, 6\}$  y  $n=8$ , el algoritmo voraz propondrá pagar con tres monedas  $\{6, 1, 1\}$ , siendo la solución óptima sólo dos monedas  $\{4, 4\}$ .

# Devolver cambio (2)

- Supongamos un **sistema monetario  $M$**  con  **$m$  tipos diferentes de monedas**, siendo  $v_i > 0$  el valor de la moneda,  $1 \leq i \leq m$ , y con un suministro ilimitado de monedas de cada tipo.
- El problema consiste en devolver el cambio de una **cantidad  $n$**  con el mínimo número de monedas.
- Para resolver el problema, preparamos una tabla  **$C[1..m, 0..n]$**  con una fila por cada valor de moneda y una columna por cada cantidad desde 0 a  **$n$** .
- **$C[i, j]$**  será el número mínimo de monedas necesario para cambiar **la cantidad  $j$** ,  $0 \leq j \leq n$ , empleando **sólo monedas de los tipos 1 a  $i$** , con  $1 \leq i \leq m$ .

# Devolver cambio (3)

- La **solución del problema** viene dada por  $C[m, n]$ , que indicará el número de monedas óptimo para cambiar la cantidad  $m$  usando monedas de tipos 1 a  $n$ .
- Se inicializa  $C[i, 0] = 0$  para todo  $i$  (0 monedas para la cantidad 0). Para calcular  $C[i, j]$ , hay dos opciones:

- Si **no se utilizan monedas** del tipo  $i$ , el número de monedas es el mismo que si no existiese el tipo  $i$

$$C[i, j] = C[i - 1, j]$$

- Si **se emplea una moneda** del tipo  $i$ , el número de monedas es **1 más** el número de monedas necesario para cubrir la cantidad  $j$  **menos** el valor de la moneda  $i$

$$C[i, j] = 1 + C[i, j - v_i]$$

# Devolver cambio (4)

- Dado que debemos **minimizar el número de monedas** utilizadas, escogeremos el mínimo de ambos valores:

$$C[i, j] = \min(C[i - 1, j], 1 + C[i, j - v_i])$$

- Esto implica dos casos extremos que se salen de la tabla:
  - $i = 1$ : Sólo hay un tipo de moneda disponible  
 $\nexists C[i - 1, j] \Rightarrow C[1, j] = 1 + C[1, j - v_1]$
  - $j < v_i$ : El valor de la moneda  $i$  es mayor que la cantidad a pagar:  
 $\nexists C[i, j - v_i] \Rightarrow C[i, j] = C[i - 1, j]$
  - $i = 1$  y  $j < v_1$ : Es imposible pagar una cantidad  $j$  empleando sólo monedas del tipo **1**:  $C[1, j] = \infty$
- Si  $C[m, n] = \infty$ , entonces no existe solución.

# Devolver cambio (5)

- Por ejemplo, para pagar 8 unidades con  $M=\{1, 4, 6\}$ :

Cantidad $n$ :		0	1	2	3	4	5	6	7	8	
1:	$v_1 = 1$	0	1	2	3	4	5	6	7	8	$\leftarrow i = 1$
2:	$v_2 = 4$	0	1	2	3	1	2	3	4	2	
3:	$v_3 = 6$	0	1	2	3	1	2	1	2	2	

$\nwarrow j < v_i$

- La solución para este caso es:

$$\begin{aligned}
 \mathbf{C[3,8]} &= \min(C[2, 8], 1+C[3, 8 - v_3]) = \\
 &= \min(C[2, 8], 1+C[3, 2]) = \min(2, 1+2) = \mathbf{2}
 \end{aligned}$$



# Devolver cambio (6)

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

- Si usamos la moneda 3 (de valor  $v_3 = 6$ ) para pagar la cantidad 8, usaremos **1** moneda más que para pagar la cantidad  $8 - v_3 = 2 \Rightarrow 1 + C[3, 2] = 3 \rightarrow \{6, 1, 1\}$
- Si no la usamos, usaremos las mismas monedas que pagando la cantidad 8 con las monedas 1 y 2  $\Rightarrow C[2, 8] = 2 \rightarrow \{4, 4\}$

# Devolver cambio (7)

```
función monedas(n: cantidad): matriz {  
  vector v[1..m] <- [1, 4, 6]  
  matriz C[1..m, 0..n]  
  para i <- 1 hasta m hacer {  
    C[i, 0] <- 0  
    para j <- 1 hasta n hacer  
      si i=1 y j<v[1] entonces          C[i,j] <-  $\infty$   
      en otro caso si i=1 entonces      C[i,j] <- 1+C[i,j-v[1]]  
      en otro caso si j<v[i] entonces C[i,j] <- C[i-1,j]  
      en otro caso C[i,j] <- min(C[i-1,j], 1+C[i,j-v[i]])  
  }  
  devolver C  
}
```

# Devolver cambio (8)

- El valor  $C[m, n]$  indica el número mínimo de monedas necesario para cubrir la cantidad  $n$ , pero no especifica qué monedas son las que hay que escoger.
- El conjunto de monedas a escoger para pagar la cantidad  $n$  se calcula haciendo un recorrido dentro de la matriz desde  $C[m, n]$  hasta  $C[-, 0]$ :
  - Si  $C[i, j] = C[i-1, j]$  (con  $i > 1$ ) la moneda  $i$  no se usa para pagar la cantidad  $j \rightarrow$  Pasamos a  $C[i-1, j]$
  - En otro caso, se añade **una** moneda de tipo  $i$  a la solución  $\rightarrow$  Pasamos a  $C[i, j - v_i]$
  - Cuando se llega a  $C[i, 0]$ , para cualquier valor de  $i$ , el valor restante a pagar es 0  $\rightarrow$  Fin

# Devolver cambio (9)

Cantidad $n$ :		0	1	2	3	4	5	6	7	8
1:	$v_1 = 1$	0	1	2	3	4	5	6	7	8
2:	$v_2 = 4$	<b>4 0</b>	1	2	3	<b>3 1</b>	2	3	4	<b>2 2</b>
3:	$v_3 = 6$	0	1	2	3	1	2	1	2	<b>1 2</b>

1.  $C[3, 8] = C[2, 8] \rightarrow$  no se selecciona moneda
2.  $C[2, 8] = 1 + C[2, 8 - v_2] = 1 + C[2, 4] \rightarrow$  entra  $v_2 = 4$
3.  $C[2, 4] = 1 + C[2, 4 - v_2] = 1 + C[2, 0] \rightarrow$  entra  $v_2 = 4$
4.  $C[2, 0] \rightarrow j = 0$ , no hacen falta más monedas

■ La solución por tanto es **2 monedas de valor 4**.

# Devolver cambio (10)

```
función cambio_monedas(m: entero, C: matriz): vector {  
  vector cambio[1..m]  
  para i <- 1 hasta m hacer cambio[i] <- 0  
  i <- m, j <- n  
  mientras j > 0 hacer {  
    si i > 1 y C[i, j] = C[i - 1, j] entonces  
      i <- i - 1  
    en otro caso { // C[i, j] = 1 + C[i, j - v[i]]  
      cambio[i] <- cambio[i] + 1  
      j <- j - v[i]  
    }  
  }  
  devolver cambio  
}
```

# Devolver cambio (11)

- La complejidad del algoritmo viene dada por la suma de los siguiente pasos:
  - Rellenar la matriz es de orden  $\Theta(m \cdot n)$
  - En el camino desde  $C[m, n]$  hasta  $C[-, 0]$ :
    - se dan como máximo  $m-1$  pasos hacia arriba, cuando no se escoge la moneda
    - La solución  $C[m, n]$  coincide con el número de saltos que se dan hacia la izquierda. Sabemos además que, en el peor caso,  $C[m, n] \leq n$  (si sólo se escogiesen monedas de valor 1)
- Por tanto,  $T(m, n) = m \cdot n + m - 1 + n = \Theta(m \cdot n)$

# El problema de la mochila (1)

- Sean  **$n$  objetos** y una mochila tales que cada objeto  $i=1\dots n$  tiene un **peso  $p_i$**  y un **valor  $v_i$**  y el peso máximo que soporta la mochila es  **$P$** . El objetivo es llenar la mochila maximizando el valor total de los objetos **enteros** y respetando la limitación de peso máximo  **$P$** .

$$\max \sum_{i=1}^n x_i v_i$$

$$s.a. \sum_{i=1}^n x_i p_i \leq P$$

$$v_i > 0, \quad p_i > 0, \quad x_i \in \{0, 1\}, \quad i = 1, \dots, n$$

# El problema de la mochila (2)

- El algoritmo voraz (*greedy*) **puede no proporcionar una solución óptima** cuando las  $x_i$  son 0 ó 1.
- Para resolver este problema usando PD, se prepara una tabla con  $V[1..n, 0..P]$  que tiene:
  - Una **fila** por cada **objeto** disponible
  - Una **columna** para cada **peso** desde 0 a  $P$
- Por tanto,  $V[i, j]$  será el **valor máximo** de los objetos que podríamos transportar si el **límite de peso fuera  $j$** , con  $0 \leq j \leq P$ , y si solamente incluyéramos los **objetos numerados desde el 1 hasta el  $i$** , con  $1 \leq i \leq n$ .
- La solución estaría en la celda  $V[n, P]$ .



# El problema de la mochila (3)

- Para calcular  $V[i, j]$  existen dos opciones:
  - **No añadir el objeto  $i$**  a la mochila:  $V[i, j] = V[i-1, j]$
  - **Añadir el objeto  $i$** , lo cual implica incrementar el valor en  $v_i$  y reducir la capacidad disponible en  $p_i$ :

$$V[i, j] = V[i-1, j-p_i] + v_i$$

Esta opción no es válida si el objeto no cabe:  $p_i > j$

- Obsérvese que, como un **objeto no se puede añadir más de una vez**, el primer índice es  $i - 1$  tanto si se escoge ( $x_i = 1$ ) como si no ( $x_i = 0$ ).
- Como se desea **maximizar el valor acumulado**:

$$V[i, j] = \max(V[i-1, j], V[i-1, j-p_i] + v_i)$$

# El problema de la mochila (4)

- Ejemplo:  $n = 5$  objetos, peso máximo  $P = 11$

Límite $P$ :	0	1	2	3	4	5	6	7	8	9	10	11
1: $p_1=1, v_1=1$	0	1	1	1	1	1	1	1	1	1	1	1
2: $p_2=2, v_2=6$	0	1	6	7	7	7	7	7	7	7	7	7
3: $p_3=5, v_3=18$	0	1	6	7	7	18	19	24	25	25	25	25
4: $p_4=6, v_4=22$	0	1	6	7	7	18	22	24	28	29	29	40
5: $p_5=7, v_5=28$	0	1	6	7	7	18	22	28	29	34	35	40

- $V[5,11] = \max(V[4, 11 - p_5] + v_5, V[4, 11]) =$   
 $= \max(V[4, 4] + 28, 40) = \max(35, 40) = 40$

# El problema de la mochila (5)

Límite $P$ :	0	1	2	3	4	5	6	7	8	9	10	11
1: $p_1=1, v_1=1$	0	1	1	1	1	1	1	1	1	1	1	1
2: $p_2=2, v_2=6$	0	1	6	7	7	7	7	7	7	7	7	7
3: $p_3=5, v_3=18$	0	1	6	7	7	18	19	24	25	25	25	25
4: $p_4=6, v_4=22$	0	1	6	7	7	18	22	24	28	29	29	40
5: $p_5=7, v_5=28$	0	1	6	7	7	18	22	28	29	34	35	40

- Si se añade el objeto 5, la capacidad es  $11 - p_5 = 4$  y el valor  $V[5, 11]$  es  $V[4, 4] + v_5 = 7 + 28 = 35$
- Si no se añade, la capacidad y el valor se mantienen.  $V[5, 11] = V[4, 11] = 40$

# El problema de la mochila (6)

- La tabla indica el valor máximo obtenido, pero calcular qué objetos se incluyen mediante un recorrido puede producir resultados ambiguos (aunque válidos):

Límite $P$ :		0	1	2	3	4	5
1:	$p_1=2, v_1=1$	0	0	1	1	1	1
2:	$p_2=3, v_2=2$	0	0	1	2	2	3
3:	$p_3=4, v_3=3$	0	0	1	2	3	3

$V[3, 5] = \max(V[2, 1] + v_3, V[2, 5]) = \max(3, 3) \Rightarrow$  Las soluciones  $\{ p_1, p_2 \}$  y  $\{ p_3 \}$  tienen igual valor, pero distinto peso

- Por ello, se mantiene una matriz auxiliar  $x$  que indica para cada posición si el objeto fue escogido ( $\checkmark$ ) o no ( $-$ ).

# El problema de la mochila (7)

Límite $P$ :		0	1	2	3	4	5	6	7	8	9	10	11
1:	$p_1=1, v_1=1$	- 0	✓ 1	✓ 1	✓ 1	✓ 1	✓ 1	✓ 1	✓ 1	✓ 1	✓ 1	✓ 1	✓ 1
2:	$p_2=2, v_2=6$	<b>4</b> - 0	- 1	✓ 6	✓ 7	✓ 7	✓ 7	✓ 7	✓ 7	✓ 7	✓ 7	✓ 7	✓ 7
3:	$p_3=5, v_3=18$	- 0	- 1	- 6	- 7	- 7	<b>3</b> ✓ 18	✓ 19	✓ 24	✓ 25	✓ 25	✓ 25	✓ 25
4:	$p_4=6, v_4=22$	- 0	- 1	- 6	- 7	- 7	- 18	✓ 22	- 24	✓ 28	✓ 29	✓ 29	<b>2</b> ✓ 40
5:	$p_5=7, v_5=28$	- 0	- 1	- 6	- 7	- 7	- 18	- 22	✓ 28	✓ 29	✓ 34	✓ 35	<b>1</b> - 40

1.  $x[5, 11] = 0 (-) \rightarrow$  no se selecciona nada  $\rightarrow [4, 11]$
  2.  $x[4, 11] = 1 (✓) \rightarrow$  se escoge el objeto **4**  $\rightarrow [3, 11 - p_4] = [3, 5]$
  3.  $x[3, 5] = 1 (✓) \rightarrow$  se escoge el objeto **3**  $\rightarrow [2, 5 - p_3] = [2, 0]$
  4.  $x[2, 0] \rightarrow$  La capacidad ha llegado a 0  $\rightarrow$  Fin
- La solución es introducir los objetos 3 y 4, con un **valor de**  
 $v_3 + v_4 = 18 + 22 = 40$

# El problema de la mochila (8)

```
función mochila(P, p[1..n], v[1..n]): valor {  
  matriz V[0..n, 0..P]  
  matriz x[1..n, 0..P]  
  para j <- 0 hasta P hacer V[0,j] <- 0 // Fila 0, sin objetos  
  para i <- 1 hasta n hacer  
    para j <- 0 hasta P hacer {  
      si j < p[i] o V[i-1, j-p[i]] + v[i] < V[i-1,j] entonces {  
        x[i,j] <- 0 ; V[i,j] <- V[i-1,j]  
      } en otro caso {  
        x[i,j] <- 1 ; V[i,j] <- V[i-1, j-p[i]] + v[i]  
      }  
    }  
  devolver V[n,P]  
}
```

# El problema de la mochila (9)

```
función solución_mochila(P, p[1..n], x[1..n, 0..P]):  
    j <- P                                conjunto {  
    S <-  $\emptyset$   
    para i <- n hasta 1 hacer  
        si x[i, j] == 1 entonces {  
            S <- S  $\cup$  {i}  
            j <- j - p[i]  
            si j = 0 entonces salir del bucle  
        }  
    devolver S  
}
```

# El problema de la mochila (10)

- Complejidad:
  - Construir la tabla  $V$  requiere  $n \cdot P$  pasos
  - En el peor caso, si se incluyesen todos los objetos en la mochila, obtener la solución óptima requeriría:
    - $n$  saltos entre filas
    - $n$  saltos entre columnas
- La complejidad total por tanto es:

$$T(n) = n \cdot P + 2n = \Theta(n \cdot P)$$



# El Principio de Optimalidad (1)

- Este principio afirma que en una **sucesión óptima** de decisiones u opciones, toda **subsecuencia debe ser también óptima**.
- También se puede formular de la siguiente forma: *la **solución óptima** de cualquier caso no trivial de un problema es una **combinación de soluciones óptimas** de **algunos** de sus casos.*
- Eso significa que, aunque el único valor de la tabla que nos interesa realmente es el de la última fila y última columna, **todas las demás entradas de la tabla** deben representar también selecciones óptimas.
- Sin embargo, cuando el **Principio de Optimalidad no es aplicable**, es probable que no sea posible atacar el problema en cuestión empleando PD.

# El Principio de Optimalidad (2)

- Cuando se desarrolla un algoritmo de PD, seguimos una secuencia de 4 pasos:
  1. Caracterizar la estructura de la solución óptima (máximo valor, mínimo peso, etc)
  2. Definir recursivamente el valor de una solución en función de las soluciones de subproblemas simples
  3. Computar la solución de cada subproblema, normalmente comenzando por los más simples
  4. Obtener una solución óptima con la información previamente computada

# El Principio de Optimalidad (3)

- Todo problema debe poseer dos características clave para poder aplicar la PD:
  - **Subestructura óptima:** si la solución óptima al problema contiene dentro de ella soluciones óptimas a subproblemas.
  - **Subproblemas solapados:** el algoritmo recursivo revisita los mismos subproblemas repetidamente. Normalmente, el número de subproblemas distintos es polinomial en el tamaño de la entrada.
- Una forma de desarrollar eficientemente la PD es mediante **memoización** (***memoization***), que consiste en almacenar las soluciones de los subproblemas visitados para que sean usadas posteriormente.

# Referencias

- ★ Brassard, G. and Bratley, P. (1998) “Fundamentos de Algoritmia”, *Prentice-Hall*. [**Capítulo 8**]
- ★ Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C. (2009) “Introduction to Algorithms”, 3rd ed, *MIT Press*. [**Capítulo 15**].
- ★ Ecuaciones editadas con:  
[s1.daumcdn.net/editor/fp/service\\_nc/pencil/Pencil\\_chromestore.html](https://s1.daumcdn.net/editor/fp/service_nc/pencil/Pencil_chromestore.html)