



Programación dinámica

IIC2133

Selección de tareas con ganancias

Tenemos n tareas,

... cada una con una hora de inicio s_i y una hora de fin f_i

- definen el intervalo de tiempo $[s_i, f_i)$ de la tarea

Para realizar las tareas tenemos una única máquina

... que sólo puede realizar una tarea a la vez

- si los intervalos de tiempo de dos tareas se traslapan, entonces solo se puede realizar una de ellas

Además ...

Cada tarea i produce una ganancia v_i si es realizada

El problema es ...

¿Cuáles tareas realizar de manera de **maximizar la suma de las ganancias de las tareas realizadas?**

P.ej., $[s_i, f_i), v_i$

$i=1$ $[0, 5), 2$

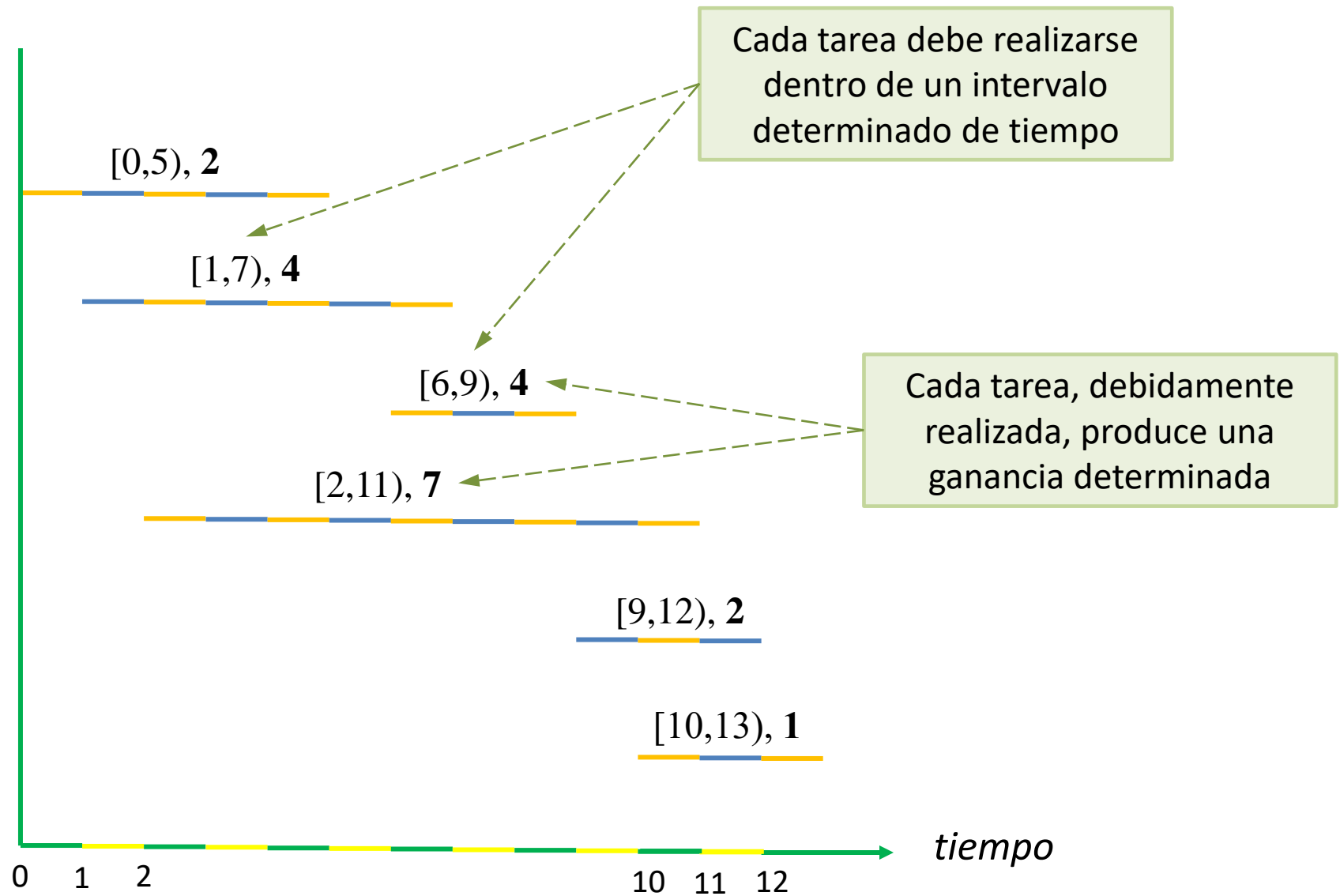
$i=2$ $[1, 7), 4$

$i=3$ $[6, 9), 4$

$i=4$ $[2, 11), 7$

$i=5$ $[9, 12), 2$

$i=6$ $[10, 13), 1$



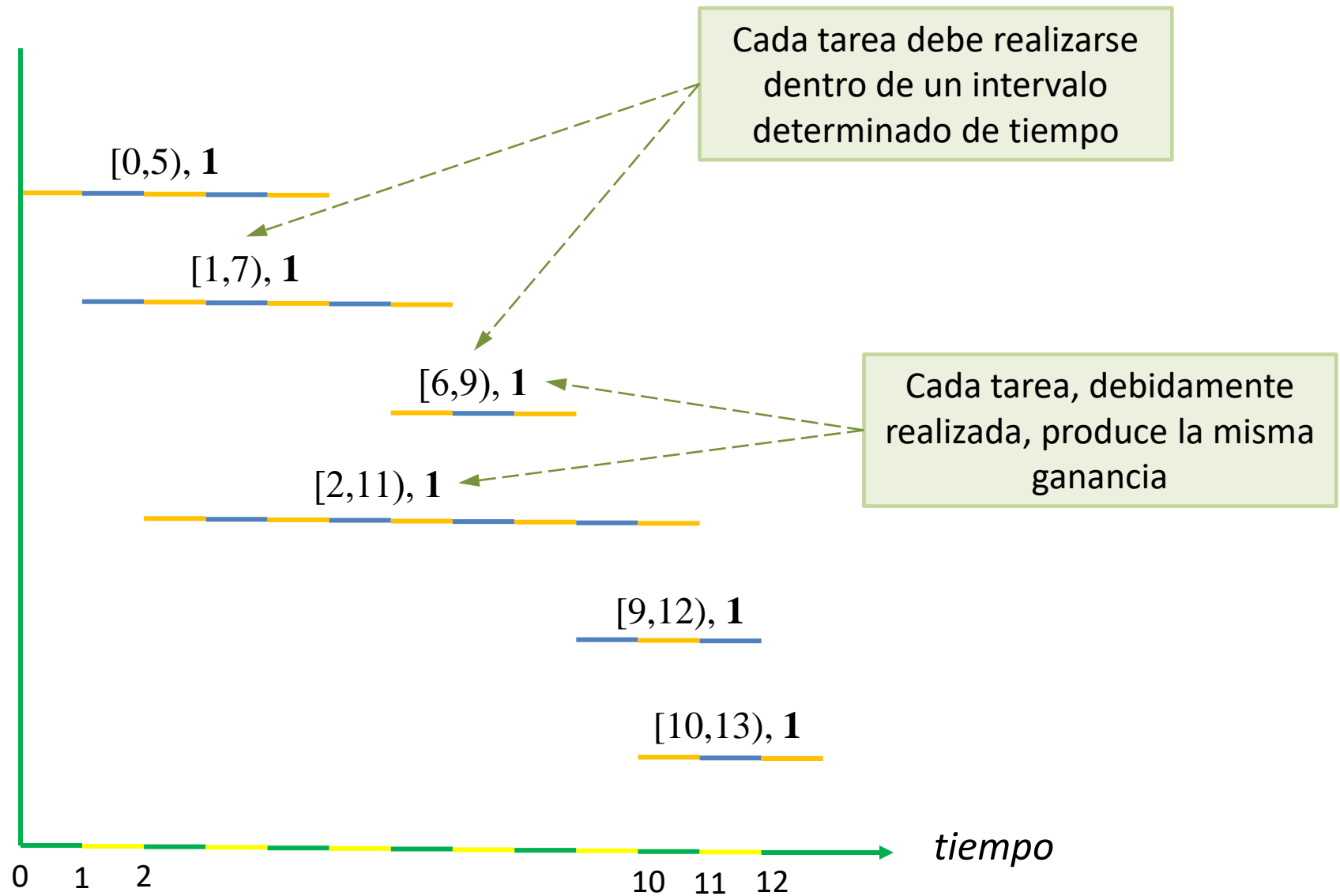
Veamos primero la versión en que cada tarea produce una ganancia de 1 si es realizada —todas valen lo mismo

¿Cuáles tareas realizar de manera de maximizar la suma de las ganancias de las tareas realizadas?

La ganancia total va a ser igual al número de tareas realizadas:

- el problema consiste entonces en seleccionar un subconjunto de tamaño máximo de tareas mutuamente compatibles

P.ej.,	$[s_i, f_i), v_i$
$i=1$	$[0, 5), 1$
$i=2$	$[1, 7), 1$
$i=3$	$[6, 9), 1$
$i=4$	$[2, 11), 1$
$i=5$	$[9, 12), 1$
$i=6$	$[10, 13), 1$



Tres estrategias codiciosas que **no** producen una solución óptima

a) elegir primero la tarea que empieza más temprano



b) elegir primero la tarea más corta



c) elegir primero la tarea que tiene menos incompatibilidades con otras tareas



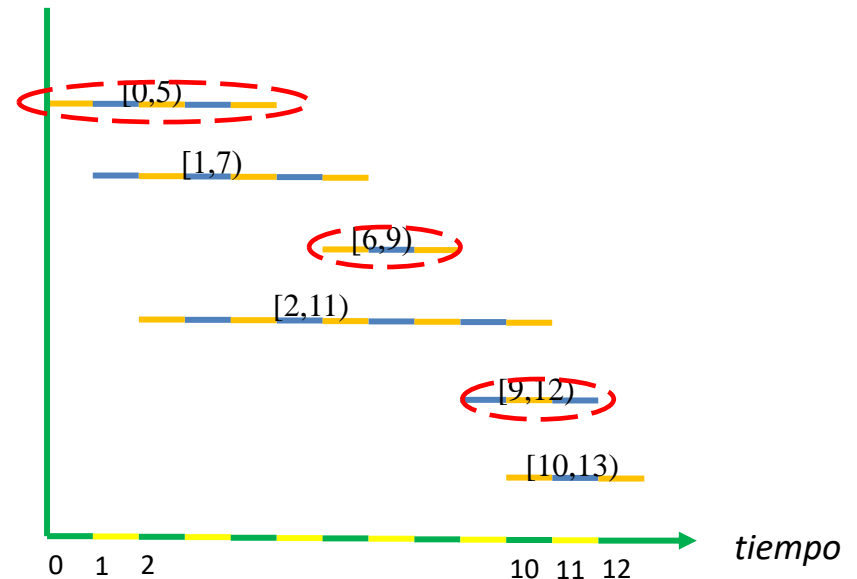
Intervalo de tiempo de una tarea; el tiempo transcurre de izquierda a derecha

Sin embargo, el problema sí puede resolverse mediante una *estrategia codiciosa* (cuando todas las tareas valen lo mismo):

elegir primero la tarea que termina más temprano

En el ej., las tareas elegidas son las tareas 1, 3 y 5, y el valor de la solución es 3:

- la tarea 1 es la que termina más temprano, en $t = 5$
- (la tarea 2 es la segunda tarea que termina más temprano, pero es incompatible con la tarea 1 \Rightarrow la descartamos)
- la tarea 3 es la tercera tarea que termina más temprano, en $t = 9$, y es compatible con la tarea 1
- (la tarea 4 es incompatible con las tareas 1 y 3)
- la tarea 5 es compatible con las tareas 1 y 3
- (la tarea 6 es incompatible con la tarea 5)



Volvamos a la versión original: cada tarea produce una ganancia v_i si es realizada

¿Cuáles tareas realizar de manera de maximizar la suma de las ganancias de las tareas realizadas?

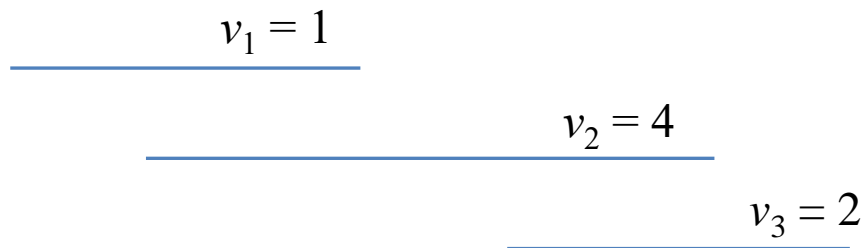
- ahora no importa el número de tareas realizadas

P.ej.,	$[s_i, f_i), v_i$
$i=1$	$[0, 5), 2$
$i=2$	$[1, 7), 4$
$i=3$	$[6, 9), 4$
$i=4$	$[2, 11), 7$
$i=5$	$[9, 12), 2$
$i=6$	$[10, 13), 1$

En este caso más general

... ni siquiera la estrategia de elegir primero la tarea que termina más temprano produce garantizadamente una solución óptima:

- el ej. de abajo muestra que la estrategia codiciosa elegiría las tareas 1 y 3, con un valor total de 3
- ... mientras que la elección de sólo la tarea 2 tiene un valor de 4



En estos casos, en que las estrategias algorítmicas más atractivas
dividir para reinar (*mergeSort*, *quickSort*), elección codiciosa (*Dijkstra*,
Kruskal), recorridos de grafos (*DFS*, *BFS*)

... no funcionan,

... recurrimos a la **programación dinámica**, técnica de aplicabilidad muy amplia

Suponemos que las tareas están ordenadas por hora de término:

- $f_1 \leq f_2 \leq \dots \leq f_n$

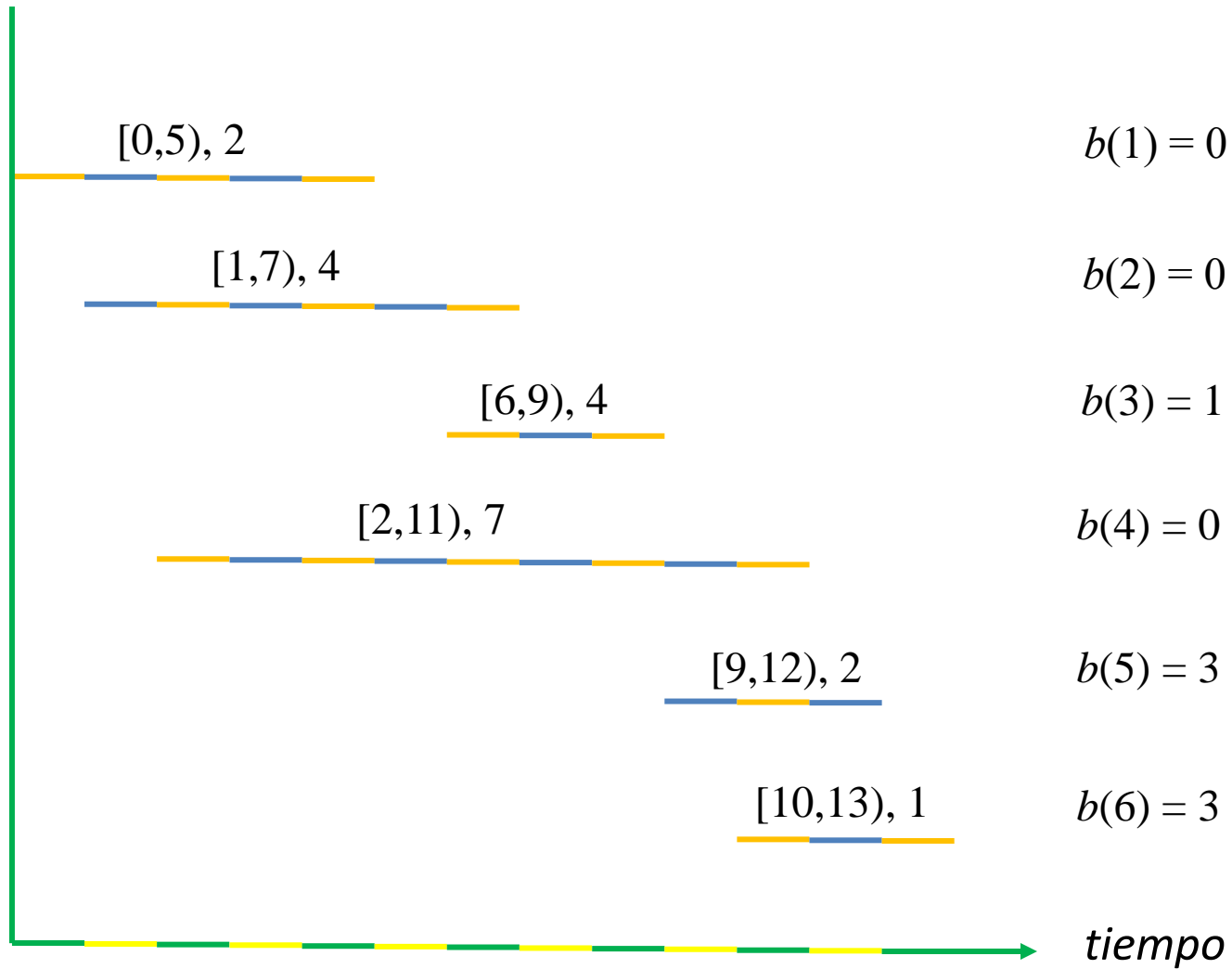
Para cada tarea k definimos ...

$b(k)$ = la tarea i que termina más tarde antes del inicio de k :

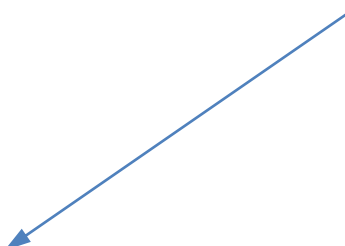
- $f_i \leq s_k$ tal que para todo $r > i$, $f_r > s_k$
- $b(k) = 0$, si ninguna tarea $i < k$ satisface la condición anterior

P.ej.,

$[0, 5), 2$	$b(1) = 0$
$[1, 7), 4$	$b(2) = 0$
$[6, 9), 4$	$b(3) = 1$
$[2, 11), 7$	$b(4) = 0$
$[9, 12), 2$	$b(5) = 3$
$[10, 13), 1$	$b(6) = 3$



Aquí —“supongamos”— empieza propiamente la aplicación de la técnica de programación dinámica:
la definición de $b(k)$ es una conveniencia para nuestros propósitos en este problema particular
es parte del un “preprocesamiento” necesario para aplicar la técnica, junto a la ordenación de las tareas por hora de término



Supongamos que tenemos una solución óptima Ω

Obviamente, con respecto a la presencia de la tarea n —la que termina más tarde entre todas las tareas— en esta solución óptima sólo hay sólo dos posibilidades:

- la tarea n **pertenece** a Ω
- ... o bien la tarea n **no pertenece** a Ω

No sabemos cuál de las dos posibilidades es la que finalmente se va a dar en la solución óptima => **hay que analizar ambas ...**

Si la tarea n **no pertenece** a Ω

... entonces Ω es igual a la solución óptima para las tareas 1, ..., $n-1$:

- un problema del mismo tipo, pero más pequeño: no incluye la tarea n

...

...

En cambio, si la tarea n **pertenece** a Ω

... entonces ninguna tarea r , $b(n) < r < n$, puede pertenecer a Ω

... y Ω debe incluir,

... además de la tarea n ,

... una solución óptima para las tareas $1, \dots, b(n)^{[*]}$:

- nuevamente, un problema del mismo tipo, pero más pequeño

[*] esta afirmación se puede demostrar por contradicción

Es decir, en ambos casos, encontrar la solución óptima para las tareas 1, ..., n involucra encontrar las soluciones óptimas a problemas más pequeños del mismo tipo:

- esta característica es clave en los problemas que se pueden resolver usando programación dinámica
- ... y, como veremos, da origen a una posible formulación recursiva de la solución del problema

Así, dado que la solución óptima al problema original involucra encontrar soluciones óptimas a problemas más pequeños del mismo tipo,

... y aplicando este mismo razonamiento a estas soluciones a los problemas más pequeños, podemos afirmar lo siguiente:

1) Sea Ω_j la solución óptima al problema de las tareas 1, ..., j

... y sea $\text{opt}(j)$ su valor

... entonces buscamos Ω_n con valor $\text{opt}(n)$

...

...

2) Además, generalizando de las diaps. anteriores, para cada Ω_j podemos decir lo siguiente con respecto a la tarea j :

- si j pertenece a Ω_j , entonces $\text{opt}(j) = v_j + \text{opt}(b(j))$
- si j no pertenece a Ω_j , entonces $\text{opt}(j) = \text{opt}(j-1)$

3) Por lo tanto,

$$\text{opt}(j) = \max\{ v_j + \text{opt}(b(j)) , \text{opt}(j-1) \} \quad [**]$$

A partir de 3), podemos escribir un algoritmo recursivo para calcular $\text{opt}(n)$:

```
opt(j):  
    if  $j = 0$ :  
        return 0  
    else:  
        return  $\max\{v_j + \text{opt}(b(j)), \text{opt}(j-1)\}$ 
```

- supone que las tareas están ordenadas por hora de término y que tenemos calculados los $b(j)$ para cada j
- $\text{opt}(0) = 0$, basado en la convención de que éste es el óptimo para un conjunto vacío de tareas

La corrección del algoritmo se puede demostrar por inducción

El problema de **opt** es su tiempo de ejecución en el peor caso:

- cada llamada a **opt** da origen a otras dos llamadas a **opt**
- esto es, *tiempo exponencial*
- p.ej., la próxima diapositiva muestra las llamadas recursivas que se producen como consecuencia de llamar inicialmente a `opt(6)`

La llamada $\text{opt}(6)$ da origen a las llamadas $\text{opt}(5)^{[1]}$ y **$\text{opt}(3)^{[2]}$** :

$^{[1]}$ $\text{opt}(5)$ da origen a las llamadas $\text{opt}(4)$ y **$\text{opt}(3)$** :

- $\text{opt}(4)$ da origen a la llamada **$\text{opt}(3)$**

- $\text{opt}(3)$ da origen a ...

$^{[2]}$ $\text{opt}(3)$ da origen a las llamadas $\text{opt}(2)$ y $\text{opt}(1)$:

- $\text{opt}(2)$ da origen a la llamada $\text{opt}(1)$

Sin embargo, en total solo se está resolviendo $n+1$ subproblemas diferentes:

- **opt(0), opt(1), ..., opt(n)**
- la razón por la que toma tiempo exponencial es el *número de veces* que resuelve cada subproblema

... p.ej., en la diapositiva anterior, se producen tres llamadas separadas a **opt(3)**, todas las cuales resuelven el mismo subproblema y, por supuesto, obtienen el mismo resultado

- esta es otra característica clave en la aplicación de programación dinámica

Podemos guardar el valor de **opt(j)** en un arreglo global la primera vez que lo calculamos

... y luego usar este valor ya calculado en lugar de todas las futuras llamadas recursivas a **opt(j)**

```
rec-opt(j):  
    if j = 0:  
        return 0  
    else:  
        if m[j] no está vacía:  
            return m[j]  
        else:  
            m[j] = max{  $v_j + \text{rec-opt}(b(j))$  ,  $\text{rec-opt}(j-1)$  }  
            return m[j]
```

rec-opt(n) es $O(n)$:

- ¿por qué?

Por supuesto, además de calcular el valor de la solución óptima,
... necesitamos saber cuál es esta solución

La clave es el arreglo \mathbf{m} :

- usamos el valor de soluciones óptimas a los subproblemas sobre las tareas $1, 2, \dots, j$ para cada j
- ... y usa $[**]$ para definir el valor de $\mathbf{m}[j]$ basado en los valores que aparecen antes (es decir, en índices menores que j) en \mathbf{m}

Una vez que llenamos \mathbf{m} , el problema está resuelto:

- $\mathbf{m}[n]$ contiene el valor de la solución óptima
- ... y podemos usar \mathbf{m} para reconstruir la solución propiamente tal

recordemos que $[**]$ es $\text{opt}(j) = \max\{ v_j + \text{opt}(b(j)) , \text{opt}(j-1) \}$

También podemos calcular los valores en m iterativamente:

- $m[0] = 0$ y vamos incrementando j
- cada vez que necesitamos calcular un valor $m[j]$, usamos $[**]$

it-opt:

$m[0] = 0$

for $j = 1, 2, \dots, n$:

$m[j] = \max\{ v_j + m[b(j)] , m[j-1] \}$

it-opt:

$m[0] = 0$

for $j = 1, 2, \dots, n$:

$m[j] = \max\{ v_j + m[b(j)] , m[j-1] \}$

Podemos demostrar por inducción que **it-opt** asigna a $m[j]$ el valor $\text{opt}(j)$

it-opt es claramente $O(n)$

El ej. de la selección de tareas con ganancias es representativo de los problemas que pueden ser resueltos eficientemente mediante programación dinámica

Generalizando, para desarrollar un algoritmo de programación dinámica

... necesitamos una colección de subproblemas, derivados del problema original, que cumplan ciertas propiedades:

- próx. diapo.

- i) el número de subproblemas es (idealmente) polinomial
 - ii) la solución al problema original puede calcularse a partir de las soluciones a los subproblemas
 - iii) hay un orden natural de los subproblemas, desde “el más pequeño” hasta “el más grande”
- ... y una recurrencia (ojalá) fácil de calcular (tal como $[**]$ en la diap. 20)
- ... que permite calcular la solución a un subproblema a partir de las soluciones a subproblemas más pequeños

La mochila con objetos 0/1^[*]

Tenemos n objetos **no fraccionables**^[*], cada uno con un valor v_k y un peso w_k ,

... y queremos ponerlos en una mochila con capacidad W :

- $W < \sum w_k$, es decir, no podemos poner todos los objetos en la mochila

... de manera de *maximizar la suma de los valores*, pero *sin exceder la capacidad de la mochila*

[*]cada objeto va completo o, simplemente, no va en la mochila: no se puede poner sólo una parte del objeto en la mochila

Si $knap(p, q, \omega)$ representa el problema de maximizar

$$\sum_{k=p}^q v_k x_k$$

... sujeto a

$$\sum_{k=p}^q w_k x_k \leq \omega \text{ y } x_k = \{0,1\}$$

... entonces nuestro problema es

$$knap(1, n, W)$$

Sea y_1, y_2, \dots, y_n una selección óptima de valores 0/1 para x_1, x_2, \dots, x_n :

- es decir, cada y_i vale 0, si el objeto j no está en la solución óptima, y vale 1 si el objeto j está en la solución óptima

En particular ...

y_1 puede ser 0 o 1 (no sabemos cuál es ... pero —del ej. anterior— sabemos qué implica cada posibilidad)

Si $y_1 = 0$ (es decir, el objeto 1 no está en la solución)

... entonces y_2, y_3, \dots, y_n debe ser una selección óptima para $\text{knap}(2, n, W)$:

- de lo contrario, no sería una selección óptima para $\text{knap}(1, n, W)$

Si $y_1 = 1$

... entonces y_2, y_3, \dots, y_n debe ser una selección óptima para $\text{knap}(2, n, W-w_1)$:

- de lo contrario, habría otra selección z_2, z_3, \dots, z_n de valores 0/1 tal que

$$\dots \sum w_k z_k \leq W-w_1 \text{ y } \sum v_k z_k > \sum v_k y_k, 2 \leq k \leq n$$

... por lo que la selección $y_1, z_2, z_3, \dots, z_n$ sería una selección para $\text{knap}(1, n, W)$ con mayor valor

Es decir, el problema se puede resolver a partir de las soluciones óptimas a subproblemas (más pequeños) del mismo tipo

Sea $g_k(\omega)$ el valor de una solución óptima a $\text{knap}(k+1, n, \omega)$:

- $g_0(W)$ es el valor de una solución óptima a $\text{knap}(1, n, W)$ —el problema original
- las decisiones posibles para x_1 son 0 y 1
- de las diapos. anteriores se deduce que

$$g_0(W) = \max\{ g_1(W) , g_1(W-w_1) + v_1 \}$$

Más aún,

... si y_1, y_2, \dots, y_n es una solución óptima a $\text{knap}(1, n, W)$,

... entonces para cada $j, 1 \leq j \leq n$

$$y_1, \dots, y_j, y_{j+1}, \dots, y_n$$

... deben ser soluciones óptimas a^[1]

$$\text{knap}(1, j, \sum w_k y_k), 1 \leq k \leq j$$

$$\text{knap}(j+1, n, W - \sum w_k y_k), 1 \leq k \leq j$$

Por lo tanto^[2],

$$g_k(\omega) = \max\{ g_{k+1}(\omega) , g_{k+1}(\omega - w_{k+1}) + v_{k+1} \}$$

... en que $g_n(\omega) = 0$ si $\omega = 0$ y $g_n(\omega) = -\infty$ si $\omega < 0$

[¹] significa que la solución a un subproblema puede calcularse a partir de las soluciones a subproblemas del mismo tipo más pequeños

[²] significa que hay una recurrencia (fácil) de calcular

P.ej., si $n = 3$, $(w_1, w_2, w_3) = (2, 3, 4)$, $(v_1, v_2, v_3) = (1, 2, 5)$, y $W = 6$

... tenemos que calcular

$$g_0(6) = \max\{ g_1(6), g_1(4)+1 \}$$

$$g_1(6) = \max\{ g_2(6), g_2(3)+2 \} = \max\{5, 2\} = 5, \text{ ya que}$$

$$g_2(6) = \max\{ g_3(6), g_3(2)+5 \} = \max\{0, 5\} = 5$$

$$g_2(3) = \max\{ g_3(3), g_3(-1)+5 \} = \max\{0, -\infty\} = 0$$

$$g_1(4) = \max\{ g_2(4), g_2(1)+2 \} = \max\{5, 2\} = 5, \text{ ya que}$$

$$g_2(4) = \max\{ g_3(4), g_3(0)+5 \} = \max\{0, 5\} = 5$$

$$g_2(1) = \max\{ g_3(1), g_3(-3)+5 \} = \max\{0, -\infty\} = 0$$

$$\text{Luego, } g_0(6) = \max\{5, 5 + 1\} = 6$$