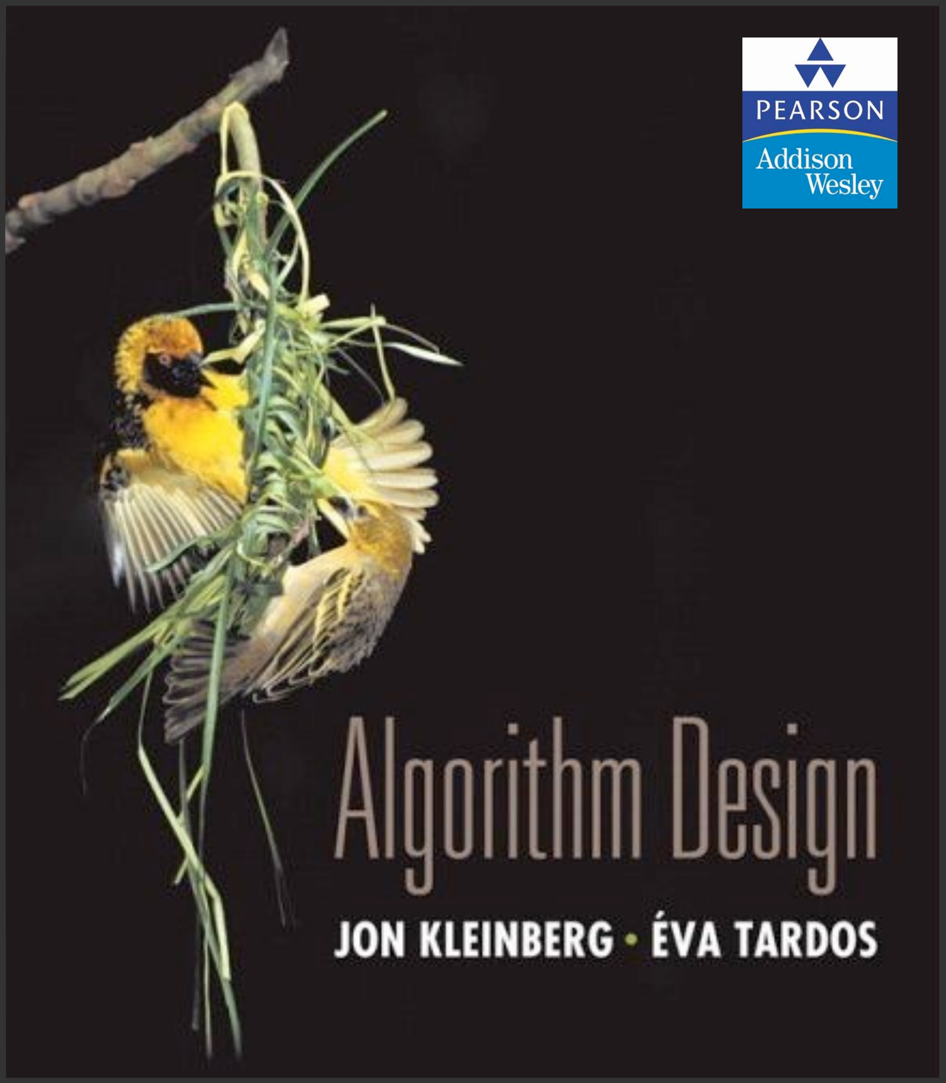


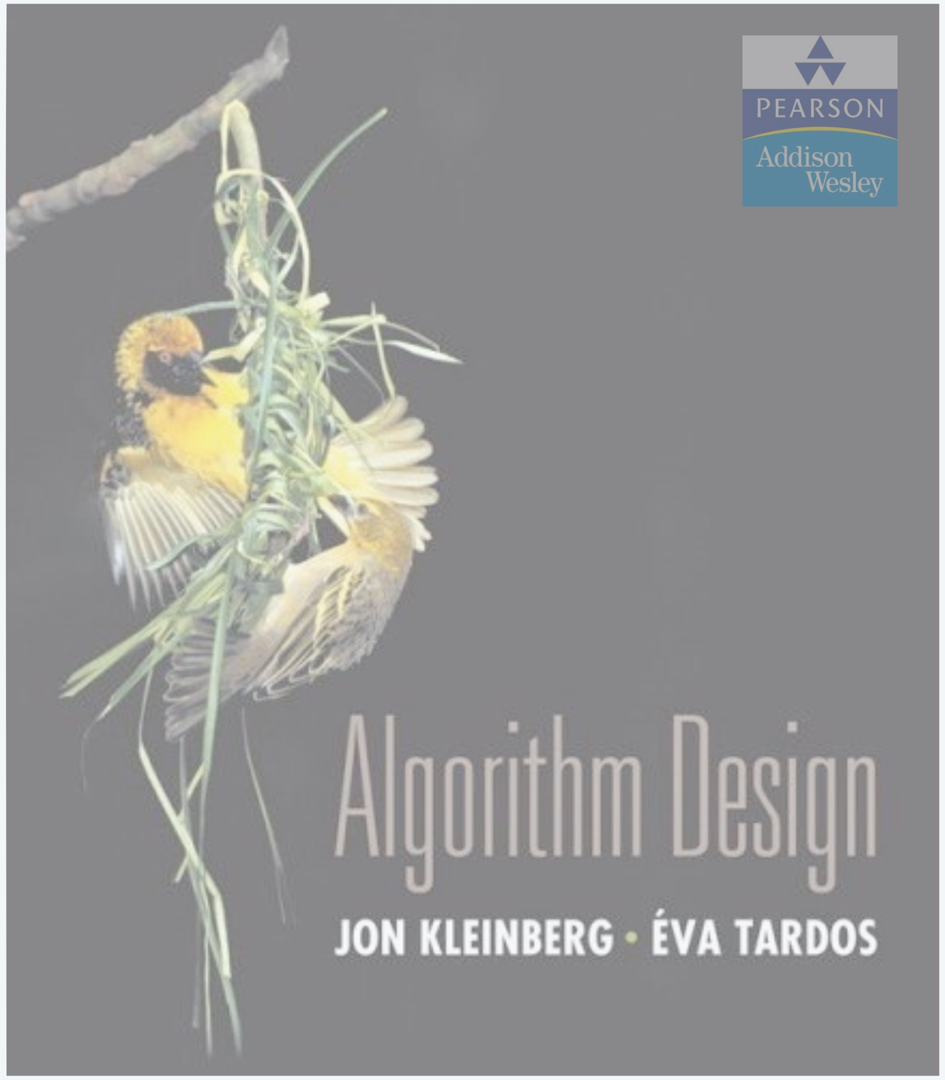
Lecture slides by Kevin Wayne

http://www.cs.princeton.edu/~wayne/kleinberg-tardos

6. Programación Dinámica I

- *Planificación de intervalos con pesos*
- *Mínimos cuadrados segmentado*
- *Problema de la mochila*

Last updated on 3/10/23 07:13



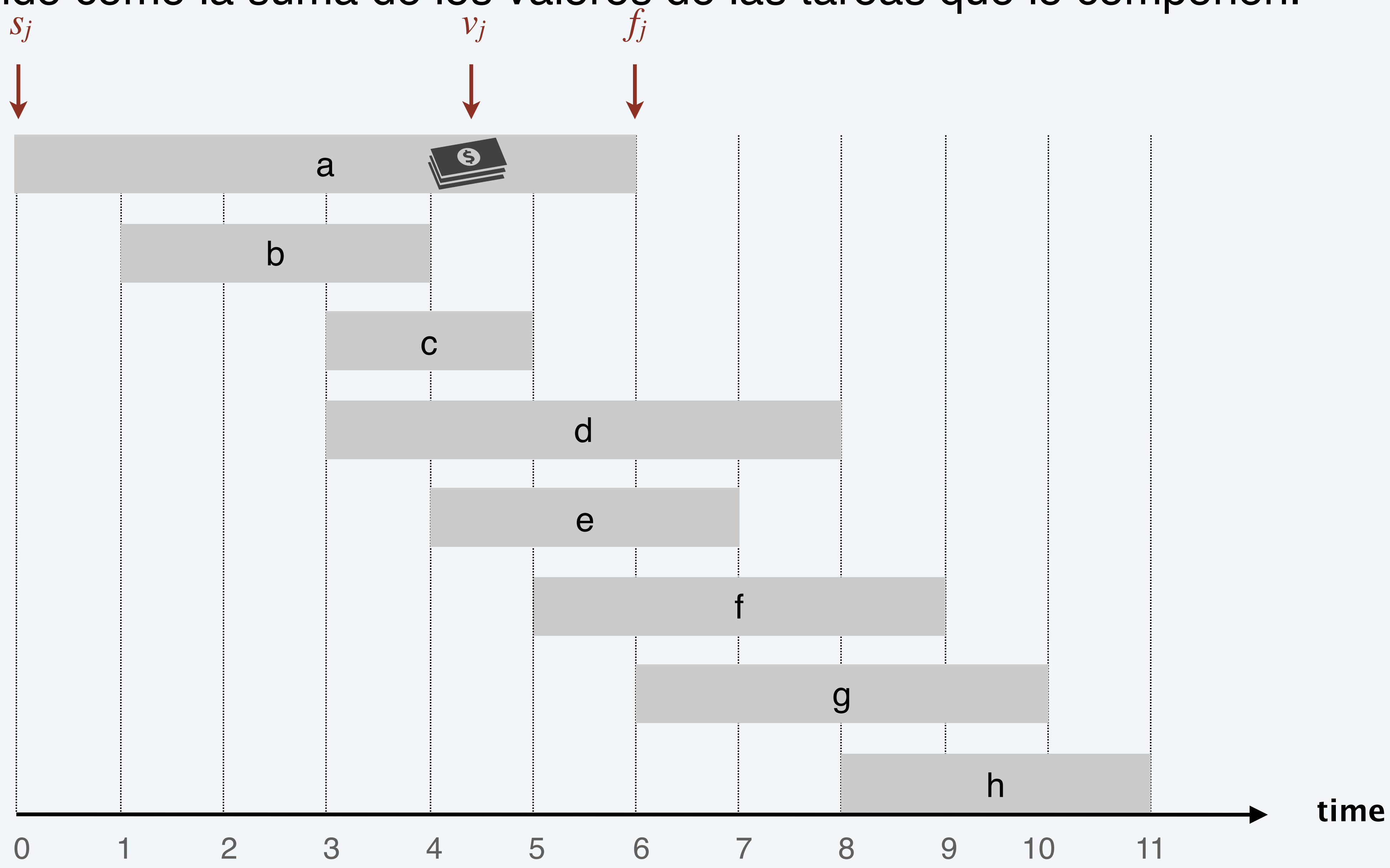
6. Programación Dinámica I

- *Planificación de intervalos con pesos*
- *Mínimos cuadrados segmentado*
- *Problema de la mochila*

Planificación de intervalos con pesos

Descripción del problema.

- Tarea j empieza en s_j, termina en f_j, y tiene *peso* o *valor* v_j.
- Dos tareas son *compatibles* si no se superponen.
- Objetivo: encontrar un **subconjunto** de tareas compatibles **de máximo *valor***, definido como la suma de los valores de las tareas que lo componen.



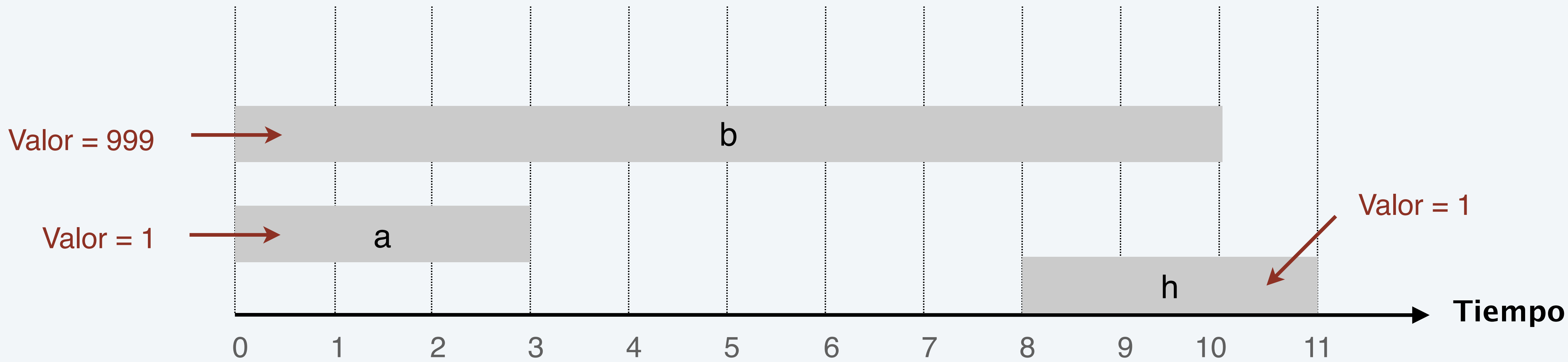
Algoritmo que ordena por tiempo de finalización

Algoritmo Greedy.

- Considerar las tareas en orden creciente de tiempo de finalización.
- Agregar una tarea a la solución si es compatible con las agregadas anteriormente.

Recordar. Es correcto si todos los intervalos tienen el mismo valor.

Observación. Falla en el caso general.

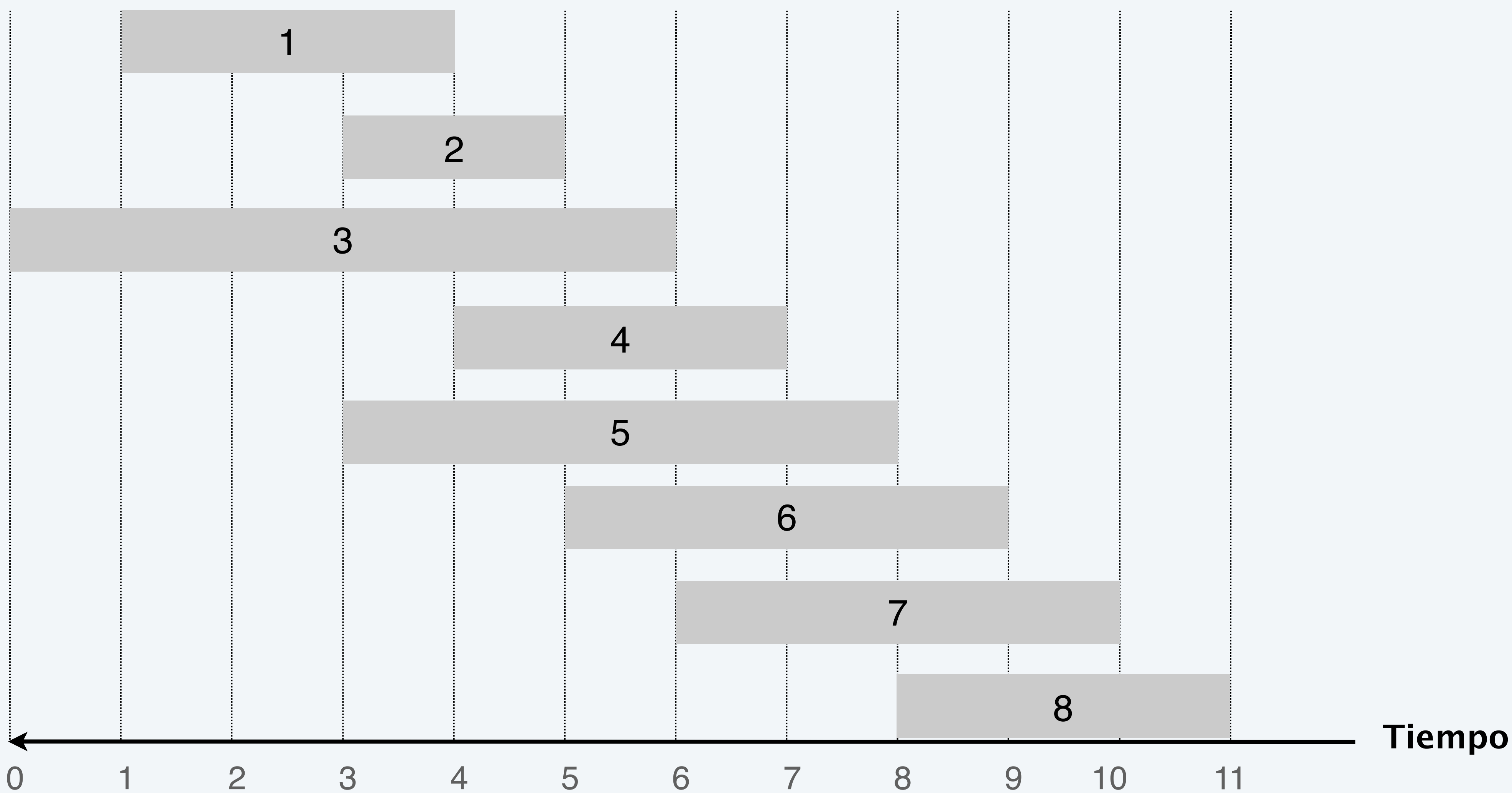


Planificación de intervalos con pesos

Notación. Etiquetar las tareas con un índice de 1 a n, ordenadas por tiempo de finalización: $f_1 \leq f_2 \leq \ldots \leq f_n$.

Def. $p(j)$ = mayor índice $i < j$ tal que la tarea i es compatible con j.

Ej. $p(8) = 5, p(7) = 3, p(2) = 0.$



Programación dinámica: elección binaria

Definición. $OPT(j)$ = **Valor** de una solución óptima para el (sub)problema que consiste en las tareas $1, 2, ..., j$.

Objetivo. $OPT(n)$ = valor de una solución óptima para el problema original.

Caso 1. $OPT(j)$ se alcanza seleccionando la tarea j. Su valor es la suma de

- la ganancia que se obtiene con la tarea j, v_j
- la máxima ganancia que puede obtener con un subconjunto de las tareas compatibles con la tarea j, $\{ 1, 2, ..., p(j) \}$.

Caso 2. $OPT(j)$ no se alcanza seleccionando la tarea j.

- Se selecciona un subconjunto óptimo para las tareas restantes $1, 2, ..., j-1$.

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max \{ v_j + OPT(p(j)),\ OPT(j-1) \} & \text{otherwise} \end{cases}$$

Algoritmo por fuerza bruta

BRUTE-FORCE $(n, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n)$

Ordenar las tareas de forma que $f_1 \leq f_2 \leq \ldots \leq f_n$.

Computar $p(1), p(2), \ldots, p(n)$.

RETURN COMPUTE-OPT(n).

COMPUTE-OPT(j)

IF $j = 0$

  RETURN 0.

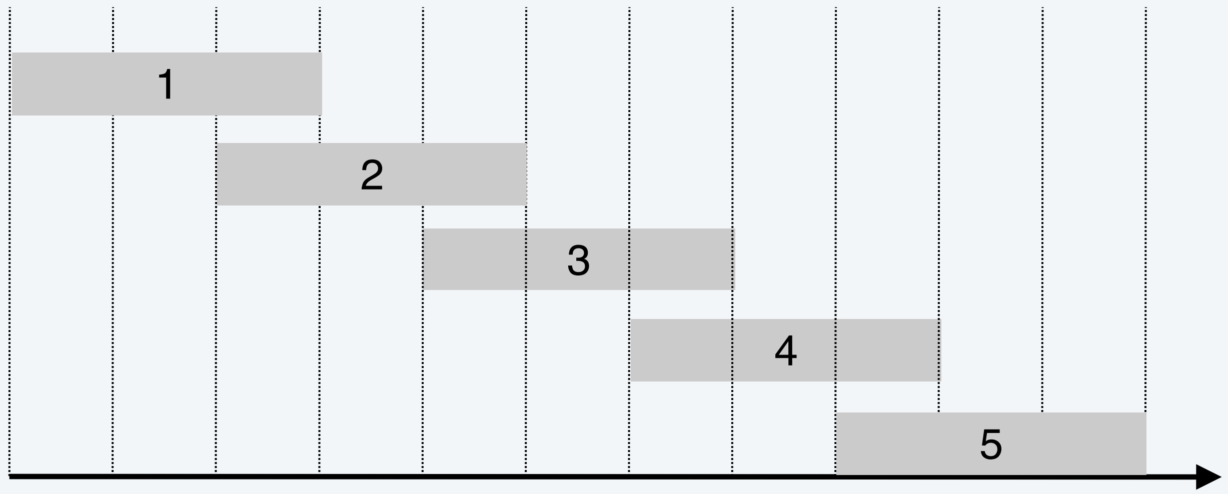
ELSE

  RETURN $\max \{ v_j +$ COMPUTE-OPT$(p(j))$, COMPUTE-OPT$(j-1) \}$.

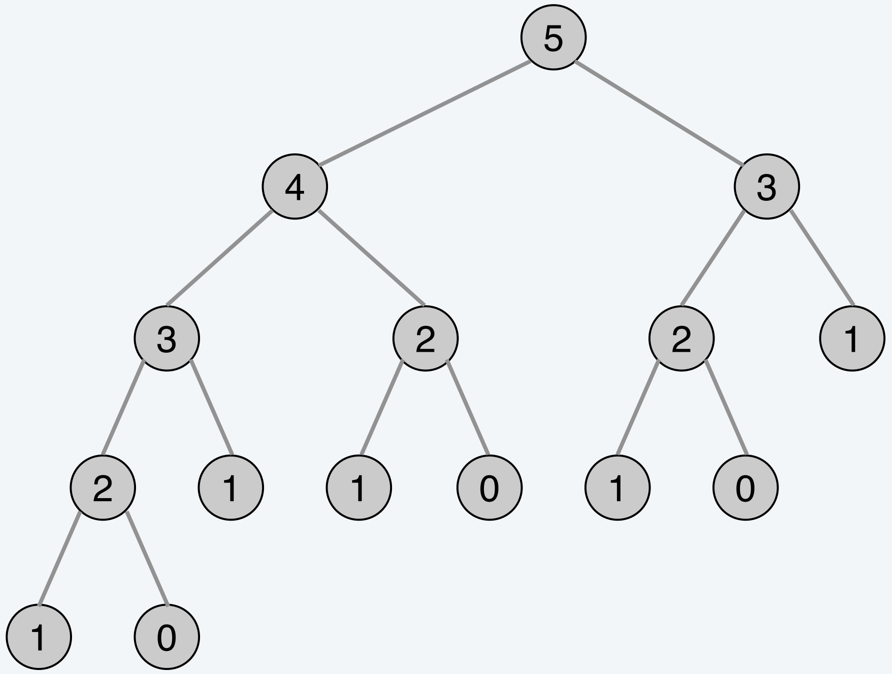
Algoritmo por fuerza bruta

Observación. Requiere tiempo exponencial.

Ej. En este ejemplo la cantidad de llamadas recursivas crece como la secuencia de Fibonacci.



p(1) = 0, p(j) = j–2



Árbol de recursión

Algoritmo recursivo con memorización (top-down)

Estrategia top-down. Almacenar resultados intermedios en una tabla la primera vez que se calculan, y tomarlos de la tabla la próxima vez que se precisen.

TOP-DOWN $(n, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n)$

Ordenar las tareas de forma que $f_1 \leq f_2 \leq \ldots \leq f_n$.

Computar $p(1), p(2), \ldots, p(n)$.

$M[0] \leftarrow 0.$ ← Arreglo global M[]

RETURN M-COMPUTE-OPT(n).

M-COMPUTE-OPT(j)

IF $M[j] = uninitialized$

$M[j] \leftarrow \max \{ v_j +$ M-COMPUTE-OPT$(p(j))$, M-COMPUTE-OPT$(j-1) \}$.

RETURN $M[j]$.

Encontrar la solución (no solo el valor)

El algoritmo anterior calcula el valor óptimo. ¿Cómo obtenemos una solución que alcance ese valor óptimo?

En una segunda pasada, usando FIND-SOLUTION(n).

```
FIND-SOLUTION ( j )
________________________________
IF j = 0
   RETURN ∅.
ELSE IF (v_j + M[p(j)] > M[j–1])
   RETURN { j } ∪ FIND-SOLUTION(p(j)).
ELSE
   RETURN FIND-SOLUTION(j–1).
```

Algoritmo iterativo (bottom-up)

Estrategia bottom-up. "Desenrrollar" la recursión desde el (los) caso base.

BOTTOM-UP $(n, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n)$

Ordenar las tareas de forma que $f_1 \leq f_2 \leq \ldots \leq f_n$.

Computar $p(1), p(2), \ldots, p(n)$.

$M[0] \leftarrow 0$.

Valores calculados previamente

FOR $j = 1$ TO n

$\quad M[j] \leftarrow \max\{ v_j + M[p(j)], M[j-1] \}$.

RETURN $M[n]$.

Encontrar la solución iterativamente

Reconstruir las decisiones que llevan a un valor óptimo para el problema completo. Esta reconstrucción es siempre top-down, es decir, en el **orden inverso** al que se construyó la tabla.

```
FIND-SOLUTION (M, n, p, v1, …, vn)
__________________________________________
j ← n.
S ← ∅.
WHILE j > 0 DO
    IF (vj + M[p(j)] > M[j–1])
        S ← S ∪ { j }.
        j ← p(j).
    ELSE
        j ← j - 1.
RETURN S.
__________________________________________
```

Programación dinámica: Algunos puntos importantes a recordar

Descomposición del problema.

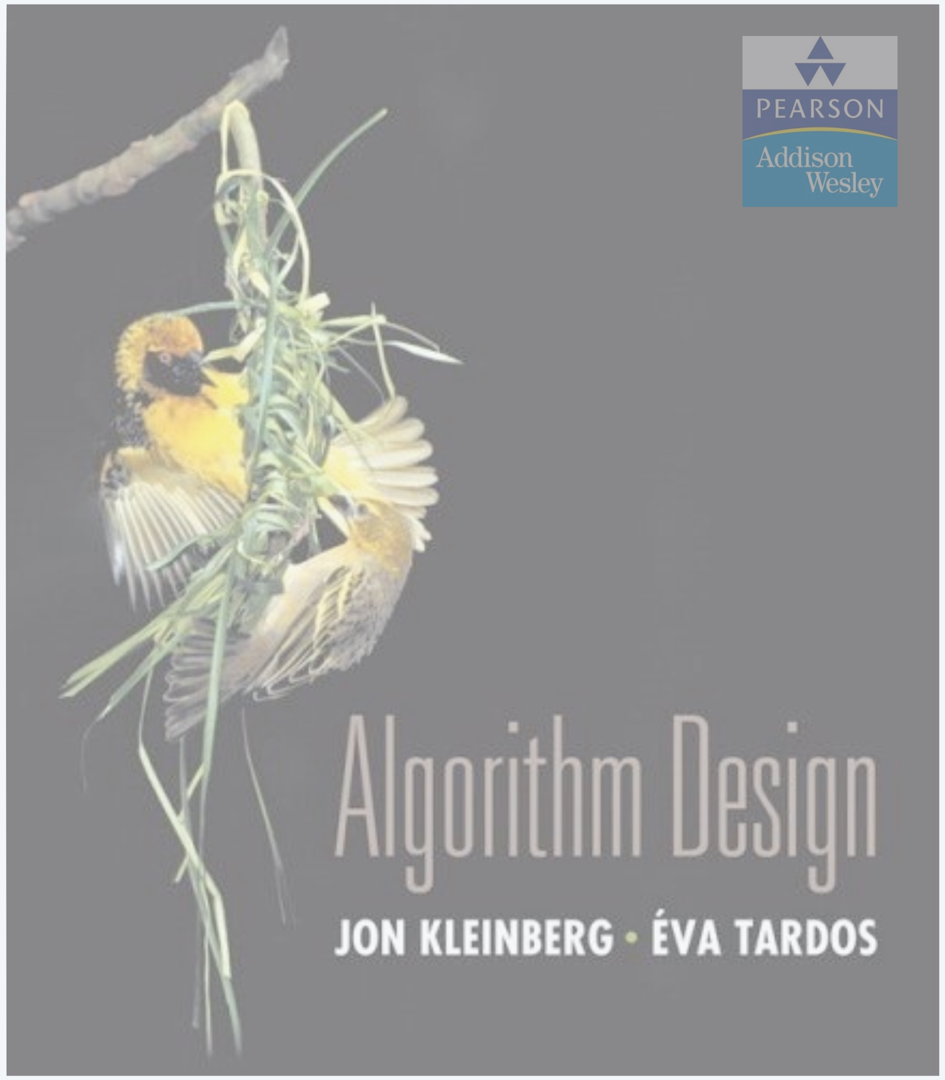
- Definir semánticamente una función para descomponer el problema. Por ejemplo, "$OPT(j) =$ Máximo valor de …."
- Definir cómo se obtiene el valor que buscamos para el problema completo. Por ejemplo, "la solución que buscamos es $OPT(n)$"
- Definir una relación de recurrencia para OPT, especificando claramente los casos en que aplica cada caso de la recurrencia.

Algoritmo para calcular el valor de una solución óptima.

- Llenar los valores de la tabla correspondientes a casos base.
- Establecer un orden de llenado de la tabla tal que, al momento de calcular una entrada de la tabla, todos los términos de la relación de recurrencia necesarios para el cálculo ya fueron calculados previamente.
- Obtener el valor para el problema completo de la tabla. Por ejemplo, $M[\ n\ ]$.

Algoritmo para obtener una solución óptima.

- Reconstruir las decisiones que llevan a un valor óptimo, partiendo desde el valor para el problema completo y avanzando en orden inverso al que se llenó la tabla.



6. PROGRAMACIÓN DINÁMICA I

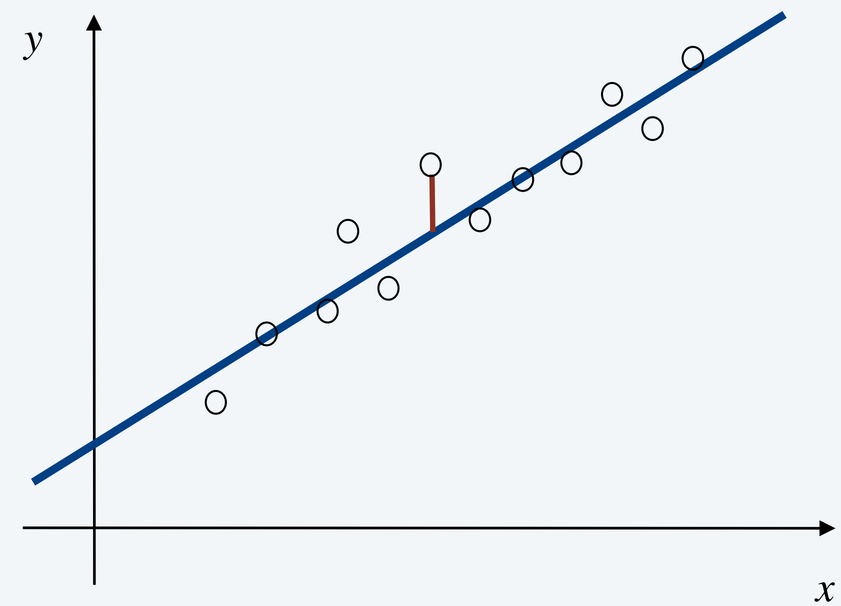
- *Planificación de intervalos con pesos*
- *Mínimos cuadrados segmentado*
- *Problema de la mochila*

Mínimos cuadrados

Mínimos cuadrados.

- Dados n puntos en el plano: $(x_1, y_1), (x_2, y_2)\ , \ldots, (x_n, y_n)$.
- Encontrar una recta $y = ax + b$ que minimiza la suma de errores cuadráticos:

$$SSE = \sum_{i=1}^{n} (y_i - ax_i - b)^2$$



Solución.

$$a = \frac{n \sum_i x_i y_i - (\sum_i x_i)(\sum_i y_i)}{n \sum_i x_i^2 - (\sum_i x_i)^2}, \quad b = \frac{\sum_i y_i - a \sum_i x_i}{n}$$

Mínimos cuadrados segmentado

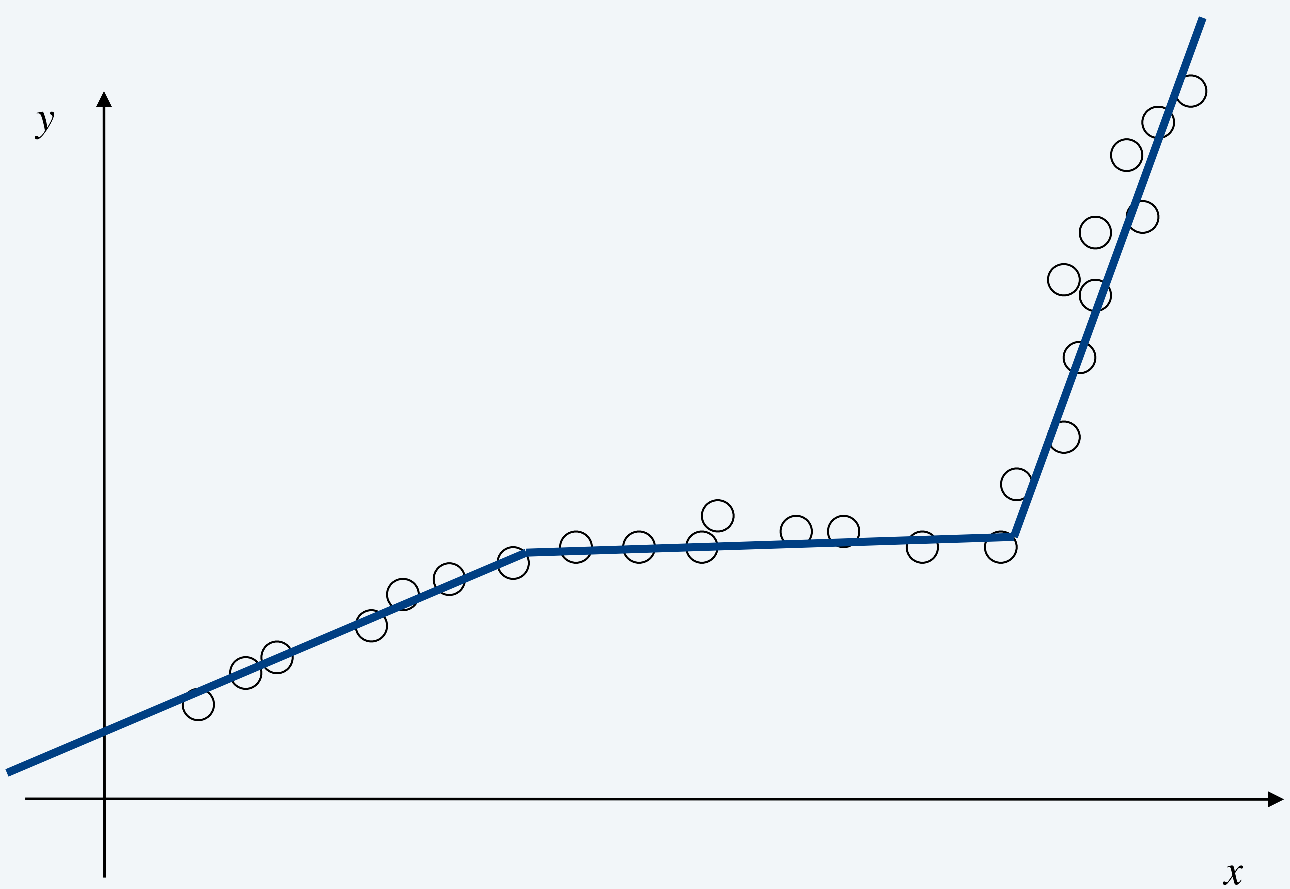
Mínimos cuadrados segmentado.

- Los puntos están aproximadamente sobre varios segmentos de recta.
- Dados n puntos en el plano: $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ con $x_1 < x_2 < ... < x_n$, encontrar una secuencia de segmentos que minimiza $f(x)$.

¿Cómo podría definirse $f(x)$ para balancear precisión y parsimonia?

bondad de ajuste

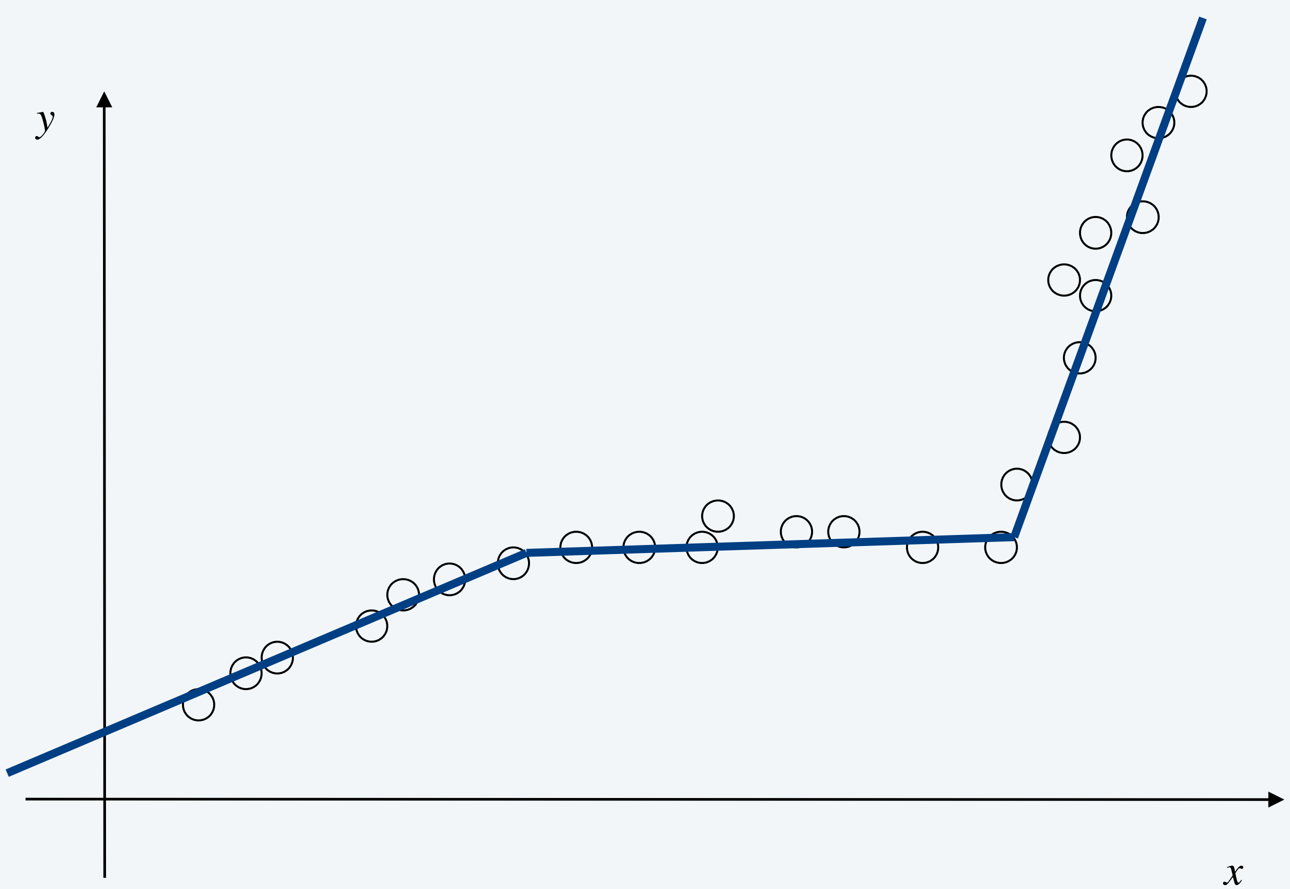
cantidad de segmentos



Mínimos cuadrados segmentado

Dados n puntos en el plano: $(x_1, y_1), (x_2, y_2)$, …, (x_n, y_n) con $x_1 < x_2 < ... < x_n$ y una constante $c > 0$, encontrar una secuencia de segmentos que minimiza $f(x) = E + c\,L$:

- E = suma de los errores cuadráticos acumulados en cada segmento.
- L = cantidad de segmentos.



Programación dinámica: elección de múltiples vías

Definición.

- $OPT(j)$ = costo mínimo para los puntos $p_1, p_2, \ldots, p_j$.
- $e(i, j)$ = error cuadrático acumulado mínimo para los puntos $p_i, p_{i+1}, \ldots, p_j$.

Cálculo de $OPT(j)$:

- El último segmento comprende los puntos $p_i, p_{i+1}, \ldots, p_j$ para algún i.
- Costo = $e(i, j) + c + OPT(i-1)$.

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \min\limits_{1 \le i \le j} \left\{ e(i,j) + c + OPT(i-1) \right\} & \text{otherwise} \end{cases}$$

Algoritmo para mínimos cuadrados segmentado

SEGMENTED-LEAST-SQUARES(n, p_1, …, p_n , c)

FOR j = 1 TO n

  FOR i = 1 TO j

    Calcular $e(i, j)$ para el segmento $p_i, p_{i+1}, \ldots, p_j$.

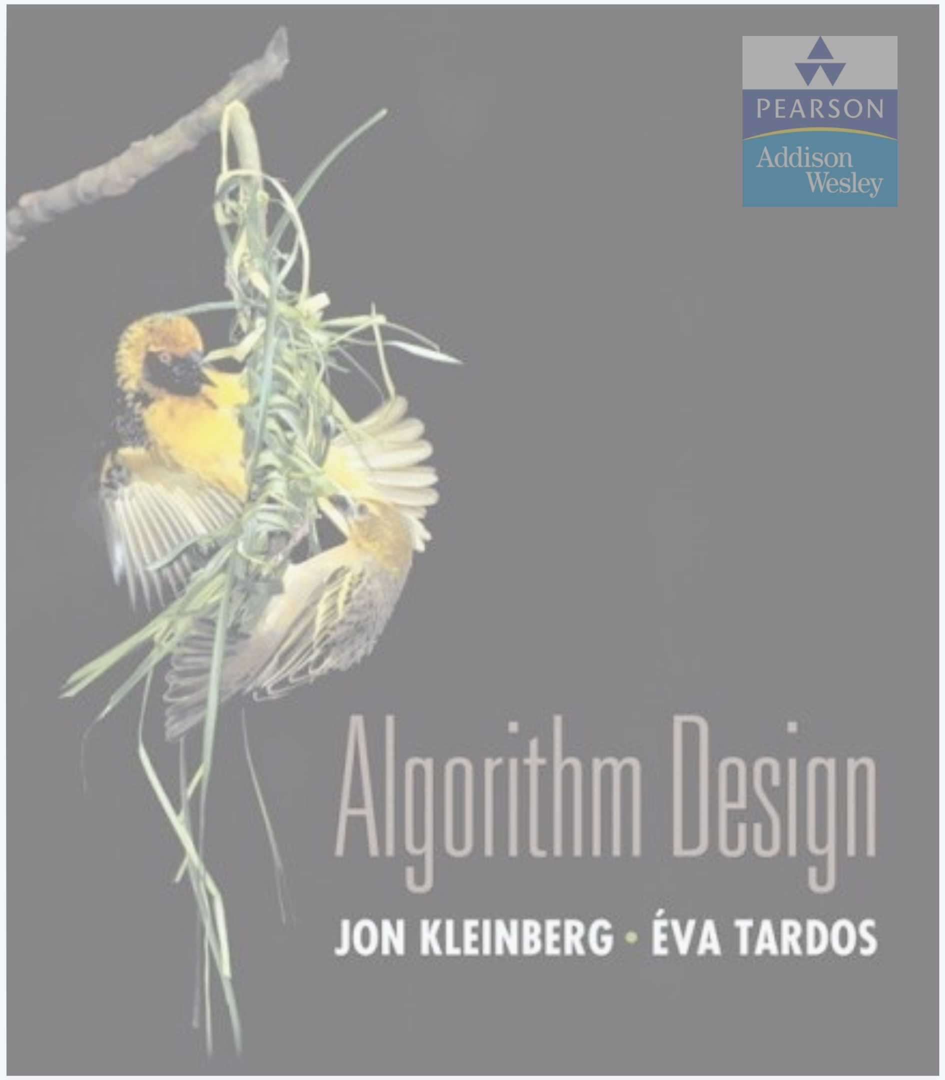
$M[0] \leftarrow 0$.

Valor calculado previamente

FOR $j = 1$ TO n

  $M[j] \leftarrow \min_{1 \leq i \leq j} \{ e(i,j) + c + M[i-1] \}$.

RETURN $M[n]$.



6. Programación Dinámica I

- *Programación de intervalos con pesos*
- *Mínimos cuadrados segmentado*
- *Problema de la mochila*

Problema de la mochila

- Tenemos n objetos y una mochila.
- Cada objeto i tiene un *peso* $w_i > 0$ y un *valor* $v_i > 0$.
- La mochila tiene una capacidad de peso W.
- Objetivo: cargar la mochila maximizando el valor cargado.

Ej. $\{ 1, 2, 5 \}$ tiene valor 35 (y peso 10).
Ej. $\{ 3, 4 \}$ tiene valor 40 (y peso 11).
Ej. $\{ 3, 5 \}$ tiene valor 46 (pero excede la capacidad).

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Instancia del problema de la mochila
(Capacidad de peso W = 11)

Greedy por valor. Agregar sucesivamente objetos con valor v_i máximo.
Greedy por peso. Agregar sucesivamente objetos con peso w_i mínimo.
Greedy por razón valor/peso. Agregar sucesivamente objetos con v_i / w_i máximo.
(en este ejemplo equivale a greedy por valor)

Observación. Ninguna de las estrategias ávidas es óptima.

Primer intento fallido

Definición. $OPT(i)$ = máximo valor para subconjunto de objetos $1, \ldots, i$.

Objetivo. $OPT(n)$.

Caso 1. $OPT(i)$ no se alcanza incluyendo al objeto i.

- Se selecciona el subconjunto de mayor valor entre los objetos $\{ 1, 2, \ldots, i-1 \}$.

Caso 2. $OPT(i)$ se alcanza incluyendo al objeto i.

- Seleccionar al objeto i no implica que debamos descartar otros objetos.
- Deberíamos seleccionar el subconjunto de mayor valor entre los objetos $\{ 1, 2, \ldots, i-1 \}$. ¿Pero con qué capacidad disponible?
- Si no sabemos qué objetos fueron seleccionados antes de i, ni siquiera sabemos si tenemos lugar para i.

Conclusión. Necesitamos más subproblemas!

Programación dinámica con una variable adicional

Def. $OPT(i, w)$ = máx. valor para subconjunto de objetos $1, \ldots, i$ con límite de peso w.

Objetivo. $OPT(n, W)$.

Posiblemente porque $w_i > w$

Caso 1. $OPT(i, w)$ no se alcanza incluyendo al objeto i.

- Una solución óptima se obtiene seleccionando un subconjunto óptimo de los objetos $\{ 1, 2, \ldots, i-1 \}$ con un límite de peso w.

Caso 2. $OPT(i, w)$ se alcanza incluyendo al objeto i. Su valor es la suma de:

- el valor del objeto i, v_i
- el máximo valor de un subconjunto de los objetos $\{ 1, 2, \ldots, i-1 \}$ cuyo peso acumulado no supere el límite disponible luego de incluir al objeto i, $w - w_i$.

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1, w) & \text{if } w_i > w \\ \max\{ OPT(i-1, w), \quad v_i + OPT(i-1, w - w_i) \} & \text{otherwise} \end{cases}$$

Problema de la mochila: algoritmo iterativo (bottom-up)

KNAPSACK $(n, W, w_1, \ldots, w_n, v_1, \ldots, v_n)$

FOR $w = 0$ TO W

$M[0, w] \leftarrow 0.$

Valores calculados previamente

FOR $i = 1$ TO n

FOR $w = 0$ TO W

IF $(w_i > w)$ $M[i, w] \leftarrow M[i-1, w].$

ELSE $M[i, w] \leftarrow \max \{ M[i-1, w],\ v_i + M[i-1, w - w_i] \}.$

RETURN $M[n, W]$.

Problema de la mochila: ejemplo

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{ OPT(i-1,w),\ \ v_i + OPT(i-1, w-w_i) \} & \text{otherwise} \end{cases}$$

Límite de peso

subconjunto de objetos 1, …, i	0	1	2	3	4	5	6	7	8	9	10	11
{ }	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	35	40

OPT(i, w) = máximo valor con subconjunto de objetos 1, …, i y límite de peso w.