

# Analisis y diseño de algoritmos. Programación Dinámica

Juan Gutiérrez

June 16, 2022

## 1 Números de Fibonacci

Considere la recurrencia:

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

Los números  $F(0), F(1), F(2), \dots, F(n)$  son denominados *Números de Fibonacci*.  $(0, 1, 1, 2, 3, 5, 8, 13, 21, 42, \dots)$

Problema: dado un entero  $n$ , calcular  $F(n)$ . Considere el siguiente algoritmo recursivo:

*Recibe:* Un número  $n$

*Devuelve:*  $F(n)$

FIBONACCI-RECURSIVO( $n$ )

1: **if**  $n == 0$

2:   **return** 0

3: **if**  $n == 1$

4:   **return** 1

5: **return** FIBONACCI-RECURSIVO( $n-1$ ) + FIBONACCI-RECURSIVO( $n-2$ )

Es claro que el tiempo de ejecución del algoritmo (suponiendo que todas las constantes valen 1) viene dado por

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

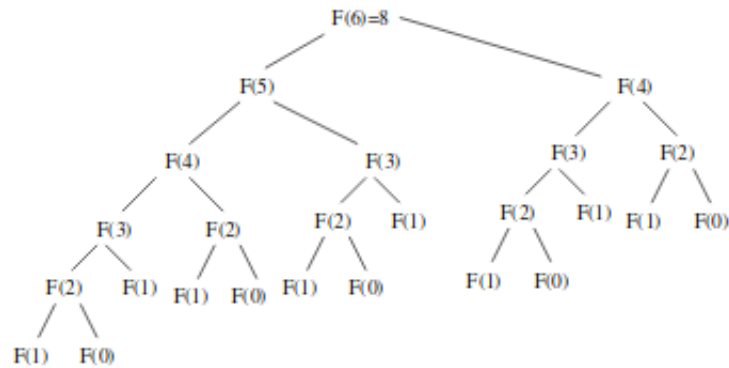
Probaremos por inducción que  $T(n) \geq \frac{2}{3}(3/2)^n$  para  $n \geq 1$ . Cuando  $n = 0$  tenemos que  $T(0) = 1 \geq \frac{2}{3}(3/2)^0 = 2/3$ . Cuando  $n = 1$  tenemos que  $T(1) =$

$1 \geq \frac{2}{3}(3/2)^1 = 1$ . Suponga entonces que  $n > 1$ . Luego.

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + 1 \\ &\geq \frac{2}{3}(3/2)^{n-1} + \frac{2}{3}(3/2)^{n-2} + 2 \\ &= \frac{2}{3}(3/2)^n \left( \frac{2}{3} + \frac{4}{9} \right) + 2 \\ &\geq \frac{2}{3}(3/2)^n \end{aligned}$$

Portanto, el tiempo de ejecución del algoritmo FIBONACCI-RECURSIVO es  $\Omega((\frac{3}{2})^n)$ .

---



Considere la siguiente mejora.

*Recibe:* Un número  $n$

*Devuelve:*  $F(n)$

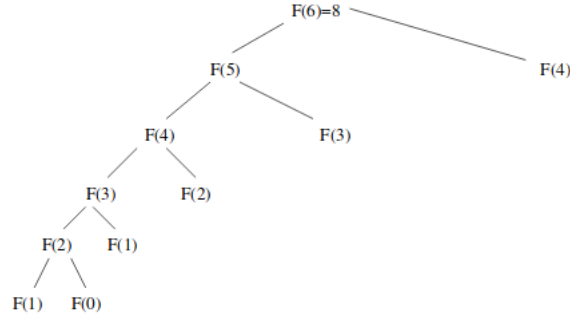
FIBONACCI-MEMOIZADO( $n$ )

- 1: **if**  $M[n] \neq -1$
- 2:     **return**  $M[n]$
- 3: **else**
- 4:      $M[n] = \text{FIBONACCI-MEMOIZADO}(n-1) + \text{FIBONACCI-MEMOIZADO}(n-2)$
- 5:     **return**  $M[n]$

MAIN( $n$ )

- 1:  $M[0] = 0$
- 2:  $M[1] = 1$
- 3: **for**  $i = 2$  **to**  $n$
- 4:      $M[i] = -1$
- 5: **return** FIBONACCI-MEMOIZADO( $n$ )

El algoritmo anterior es una versión *memoizada* de la versión recursiva. Lo que estamos haciendo es guardar resultados que ya han sido calculados en memoria.



Sea  $T(n)$  el tiempo de ejecución de la rutina FIBONACCI-MEMOIZADO. Note que el tiempo de ejecución de la llamada recursiva con parámetro  $n - 2$  siempre será constante, ya que  $M[n - 2]$  será previamente calculado en la llamada con parámetro  $n - 1$ . Portanto

$$T(n) = \begin{cases} 1 & \text{si } n \leq 1 \\ T(n - 1) + 1 & \text{si } n > 1 \end{cases}$$

Lo que implica que  $T(n) = \Theta(n)$  (otra manera de llegar a este resulta es observar que la línea 4 del algoritmo es ejecutada máximo  $n$  veces). Es claro que se ha gastado un poco más de memoria que en el caso anterior, sin embargo, la memoria gastada también es lineal en  $n$ . Concluimos que la eficiencia de FIBONACCI-MEMOIZADO es mucho mayor que la eficiencia de FIBONACCI-RECURSIVO.

Podemos mejorar aún más FIBONACCI-MEMOIZADO eliminando las llamadas recursivas y manteniendo el tiempo de ejecución.

*Recibe:* Un número  $n$

*Devuelve:*  $F(n)$

FIBONACCI-PROG-DIN( $n$ )

- 1:  $M[0] = 0$
- 2:  $M[1] = 1$
- 3: **for**  $i = 2$  **to**  $n$
- 4:      $M[i] = M[i - 1] + M[i - 2]$
- 5: **return**  $M[n]$

La técnica anterior es llamada *programación dinámica*.

## 2 Coeficientes binomiales

Para cada par de números naturales  $n, k$  con  $n \geq k$ . Considere la siguiente fórmula:  $C(n, k) = \binom{n}{k} = \frac{n!}{(n-k)!k!}$ . (Dicha fórmula cuenta la cantidad de maneras de escoger  $k$  elementos de  $n$  posibles).

Si queremos computar la función  $C(n, k)$  podemos aprovechar una propiedad dada por el triángulo de pascal:

			1			
		1		1		
		1	2	1		
	1	3	3	1		
	1	4	6	4	1	
1	5	10	10	5	1	

Note que  $C(n, k) = C(n - 1, k - 1) + C(n - 1, k)$ . Por qué se cumple? Queremos elegir  $k$  elementos de  $n$  posibles. Si el primer elemento de estos  $n$  está en el grupo escogido entonces quedan  $n - 1$  elementos para escoger  $k - 1$  elementos. Si el primer elemento de estos  $n$  no está en el grupo escogido entonces quedan  $n - 1$  elementos para escoger  $k$  posibles.

Para completar la recurrencia, debemos calcular los casos base.

Un caso base es cuando  $k = 0$ . En este caso, queremos escoger 0 elementos y la única manera es escogiendo el conjunto vacío.

El otro caso base ocurre cuando  $k = n$ . En este caso también podemos escoger un solo conjunto.

Luego

$$C(n, k) = \begin{cases} 1 & \text{si } k = 0 \text{ o } k = n \\ C(n - 1, k - 1) + C(n - 1, k) & \text{caso contrario} \end{cases}$$

Entonces tenemos directamente un algoritmo de programación dinámica.

*Recibe:* Números naturales  $n, k$  con  $n \geq k$ .

*Devuelve:*  $C(n, k)$

COEF-BIN-PROG-DIN( $n, k$ )

```

1: for  $i = 0$  to  $n$ 
2:    $C[i, i] = C[i, 0] = 1$ 
3: for  $i = 1$  to  $n$ 
4:   for  $j = 1$  to  $i - 1$ 
5:      $C[i, j] = C[i - 1, j - 1] + C[i - 1, j]$ 
6: return  $C[n, k]$ 
```

Es claro que el tiempo de ejecución del algoritmo es  $\Theta(n^2)$ .

### 3 Intervalos disjuntos

Sean  $[s_1, f_1], [s_2, f_2], \dots, [s_n, f_n]$  una secuencia de intervalos cerrados en la recta. A cada intervalo  $[s_i, t_i]$  le asignamos un *valor* o *peso*  $v_i$ . Dos intervalos son *compatibles* si no se traslapan.

**Problema Max-Intervalos-Disjuntos.** Dada una secuencia de intervalos cerrados en la recta, encontrar un subconjunto de intervalos compatibles dos a dos con el mayor peso.

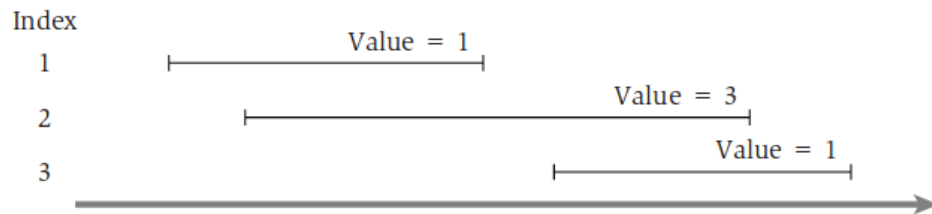


Figure 1: Tomada del libro Kleinberg, Algorithm Design

Podemos suponer, sin pérdida de generalidad, que los intervalos están ordenados de acuerdo a su punta final. Es decir, que  $f_1 \leq f_2 \leq \dots \leq f_n$ .

Según este orden, tendremos el primer intervalo, segundo intervalo, etc.

Decimos que un intervalo  $i$  está *antes* de un intervalo  $j$  si  $f_i \leq f_j$ . Para cada intervalo  $j$ , definimos  $p(j)$  como el máximo intervalo  $i < j$  tal que  $i$  y  $j$  no se intersectan.

Si no existe dicho intervalo, entonces definimos  $p(j) = 0$ .

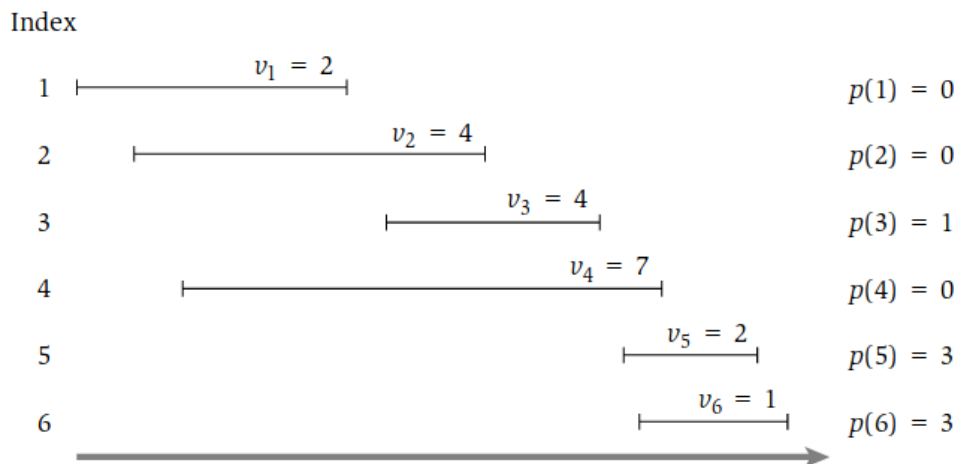


Figure 2: Tomada del libro Kleinberg, Algorithm Design

Sea  $X$  una solución óptima para el problema. Existen dos casos dependiendo del último intervalo.

- Caso 1:  $n \in X$ . En ese caso,  $X$  no contiene ningún intervalo mayor a  $p(n)$ . Considere  $X' = X \setminus \{n\}$ . Note que  $X'$  es una solución óptima para el subproblema que considera a los intervalos  $1, 2, \dots, p(j)$ .

Por qué, suponga por contradicción que  $X'$  no es una solución óptima, entonces existe una solución  $Y'$  para este subproblema con mayor peso

que  $X'$ . Luego  $Y = Y' \cup \{n\}$  sería una solución mejor que  $X$  para el problema original, contradicción.

- Caso 2:  $n \notin X$ .

En ese caso,  $X$  es una solución óptima para el subproblema que considera a los intervalos  $1, 2, \dots, n-1$ .

Por qué ? Suponga por contradicción que no lo es. Entonces existe una solución para este subproblema con mayor peso que  $X$ . Esta solución también sería una solución para el problema original, con peso mayor que  $X$ , contradicción.

A partir del análisis anterior, tenemos el siguiente lema.

**Lema 3.1.** Sea  $OPT(j)$  el valor de una solución óptima para el subproblema que considera a los intervalos  $1, 2, \dots, j$ . Entonces,

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

*Proof.* Sea  $X = \{i_1, i_2, \dots, i_k\}$  una solución óptima para el subproblema con intervalos  $1, 2, \dots, j$ , con  $i_1 < i_2 < \dots < i_k$ . Suponga primero que  $i_k = j$ . Sea  $X' = \{i_1, i_2, \dots, i_{k-1}\}$ . Note que  $X'$  es una solución óptima para el subproblema con intervalos  $1, 2, \dots, p(j)$  (vea el análisis anterior). Por lo tanto

$$OPT(j) = \sum_{i_\ell \in X} v_{i_\ell} = \sum_{i_\ell \in X'} v_{i_\ell} + v_j = OPT(p(j)) + v_j \quad (1)$$

Suponga ahora que  $i_k \neq j$ . Note que  $X$  es una solución óptima para el subproblema con intervalos  $1, 2, \dots, j-1$  (vea el análisis anterior). Por lo tanto

$$OPT(j) = OPT(j-1) \quad (2)$$

De (1) y (2), tenemos que

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

Sea  $X_1$  una solución óptima para el subproblema con intervalos  $1, 2, \dots, p(j)$ . Sea  $X_2$  una solución óptima para el subproblema con intervalos  $1, 2, \dots, j-1$ . Note que, tanto  $X_1 \cup \{j\}$  e  $X_2$  son soluciones viables para el problema original, cuyos costos son  $OPT(p(j)) + v_j$  y  $OPT(j-1)$ . Luego

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

□

El análisis anterior nos permite diseñar un algoritmo recursivo para el problema.



En un peor caso tendríamos la siguiente recurrencia  $T(n) = T(n - 1) + T(n - 2)$ :

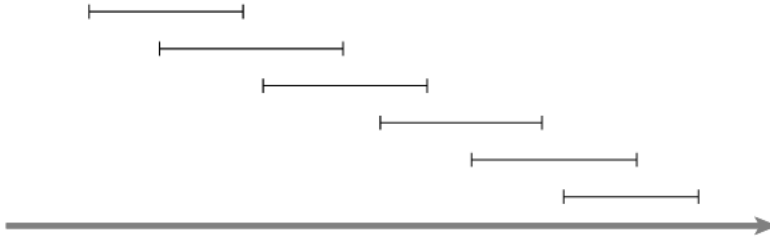


Figure 4: Tomada del libro Kleinberg, Algorithm Design

Es claro que en este caso  $T(n) = \Omega(2^n)$ .

### Versión memoizada

Note que solo existen  $n$  posibles subproblemas. Eso nos da una idea de poder desarrollar un algoritmo polinomial en  $n$ .

El detalle del algoritmo recursivo es que estamos resolviendo el mismo problema más de una vez. Para almacenar soluciones anteriores usamos la técnica de memoización.

*Recibe:* Una secuencia de intervalos  $\mathcal{I} = [s_i, f_i] : i = 1 \dots n$  con pesos  $v_i$  ordenados por la punta final. Un entero  $j$ .

*Devuelve:* El peso máximo de un subconjunto de intervalos con peso máximo de entre los  $j$  primeros intervalos.

INTERVALOS-MEMOIZADO( $\mathcal{I}, j$ )

```

1: if  $j = 0$ 
2:   return 0
3: if  $M[j] \neq -1$ 
4:   return  $M[j]$ 
5:  $\text{OPT}_1 = \text{INTERVALOS-MEMOIZADO}(\mathcal{I}, p(j))$ 
6:  $\text{OPT}_2 = \text{INTERVALOS-MEMOIZADO}(\mathcal{I}, j - 1)$ 
7:  $M[j] = \max\{v_j + \text{OPT}_1, \text{OPT}_2\}$ 
8: return  $M[j]$ 

```

Note que el tiempo de ejecución de INTERVALOS-MEMOIZADO viene determinado por el número de llamadas a la línea 8 del algoritmo y la línea 4 del algoritmo. Observe que el número máximo de llamadas a la línea 8 es  $n$ , y el número máximo de llamadas a la línea 4 es también  $n$ , por lo tanto el tiempo de ejecución es  $O(n)$ .

Observe que el algoritmo anterior otorga *el valor óptimo de la solución*, sin embargo no entrega *una solución óptima*. Sabiendo el resultado del algoritmo anterior, podemos modificar este algoritmo para encontrar dicha solución.



*Recibe:* Una secuencia de intervalos  $\mathcal{I} = [s_i, f_i] : i = 1 \dots n$  con pesos  $v_i$  ordenados por la punta final. Un entero  $j$ . Un arreglo  $M$  resultante del algoritmo INTERVALOS-MEMOIZADO.

*Devuelve:* Un subconjunto de intervalos con peso máximo de entre los  $j$  primeros intervalos.

INTERVALOS-MEMOIZADO-SOLUCION( $\mathcal{I}, j$ )

```

1: if  $j = 0$ 
2:   return  $\emptyset$ 
3: if  $v_j + M[p(j)] \geq M[j - 1]$ 
4:   return  $\{j\} \cup \text{INTERVALOS-MEMOIZADO}(\mathcal{I}, p(j))$ 
5: else
6:   return  $\text{INTERVALOS-MEMOIZADO}(\mathcal{I}, j - 1)$ 

```

El tiempo de ejecución de INTERVALOS-MEMOIZADO-SOLUCION es  $O(n)$ , ya que solo se efectúa una llamada recursiva cada vez.

## Programación dinámica (versión iterativa)

Podemos crear una versión iterativa para el problema anterior, ya que solo nos interesa lo almacenado en  $M$ .

*Recibe:* Una secuencia de intervalos  $\mathcal{I} = \{[s_i, f_i] : i = 1 \dots n\}$  con pesos  $v_i$  ordenados por la punta final de manera no decreciente.

*Devuelve:* El valor de un subconjunto de intervalos compatibles con peso máximo.

INTERVALOS-PROG-DINAMICA( $\mathcal{I}$ )

```

1:  $M[0] = 0$ 
2: for  $j = 1$  to  $n$ 
3:    $M[j] = \max\{v_j + M[p(j)], M[j - 1]\}$ 
4: return  $M[n]$ 

```

Un análisis similar a los casos anteriores nos permiten demostrar que el algoritmo hace lo pedido. Además es claro que el tiempo de ejecución del algoritmo es  $\Theta(n)$ .

**Observación 3.1.** En el análisis de los algoritmos anteriores no está considerado el tiempo de ejecución para preprocesar. Por ejemplo, para ordenar los intervalos por tiempo final debemos consumir  $O(n \lg n)$  usando algún algoritmo de ordenación conocido. Además, para calcular los valores de  $p$ , también debemos hacer un preprocesamiento  $O(n^2)$  el cual puede ser mejorado a  $O(n \lg n)$  usando búsqueda binaria.

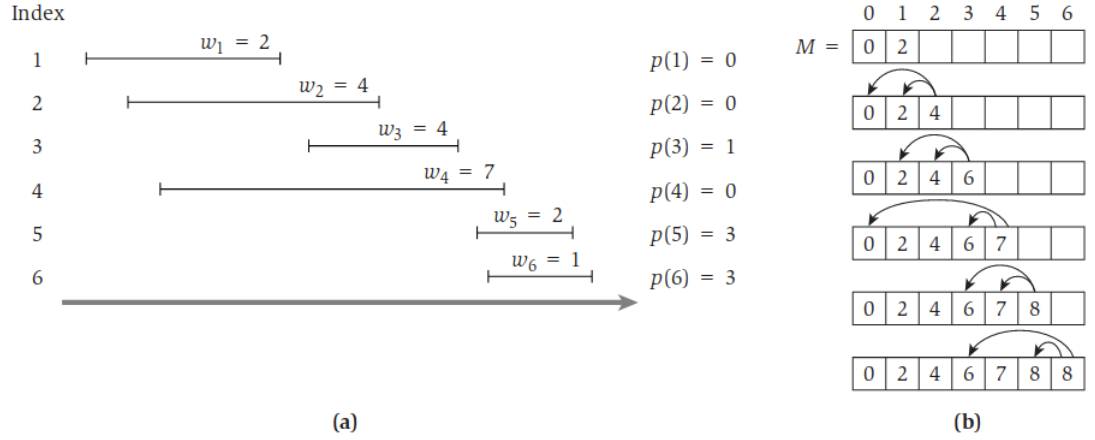


Figure 5: Tomada del libro Kleinberg, Algorithm Design

## 4 Subsecuencia creciente máxima

Dado un vector  $A[1..n]$  de números, una subsecuencia creciente es una secuencia de índices  $(i_1, i_2, \dots, i_k)$  tales que  $i_1 < i_2 < \dots < i_k$  y  $A[i_1] < A[i_2] < \dots < A[i_k]$ .

**Problema Max-Subsecuencia-Creciente.** Dado un vector  $A[1..n]$  encontrar una subsecuencia creciente de tamaño máximo en  $A$ .

Por ejemplo,  $(1, 2, 4, 6, 8)$  es una subsecuencia creciente máxima, con tamaño 5, en el arreglo  $[2, 4, 3, 5, 1, 7, 6, 9, 8, 5]$ .

Para cada  $i$ , sea  $X_i$  una subsecuencia máxima que *termina en*  $i$ . Por ejemplo, para el arreglo anterior, tenemos

$$X_1 = (1)$$

$$X_2 = (1, 2)$$

$$X_3 = (1, 3)$$

$$X_4 = (1, 2, 4)$$

$$X_5 = (5)$$

$$X_6 = (1, 2, 4, 6)$$

$$X_7 = (1, 2, 4, 7)$$

$$X_8 = (1, 2, 4, 6, 8)$$

$$X_9 = (1, 2, 4, 6, 9)$$

$$X_{10} = (1, 2, 10)$$

Reformulamos el problema MAX-SUBSECUENCIA-CRECIENTE como

**Problema Max-Subsecuencia-Creciente- $i$ .** Dado un vector  $A[1..n]$  encontrar una subsecuencia creciente de tamaño máximo en  $A$  que termina en  $i$ .

Note entonces que si  $(i_1, i_2, \dots, i_{k-1}, i_k)$  es una subsecuencia creciente que termina en  $i_k$ , entonces  $(i_1, i_2, \dots, i_{k-1})$  es una subsecuencia creciente que termina en  $i_{k-1}$ . ¿Por qué? si no fuese el caso, existiría otra secuencia que termina en  $i_{k-1}$  de mayor tamaño, al adicionarle  $i_k$  a esta secuencia, obtenemos una secuencia más grande para el problema original, una contradicción.

Por tanto,

**Lema 4.1.** *Si  $OPT(i)$  es el tamaño máximo de una subsecuencia creciente que termina en  $i$ , entonces*

$$OPT(i) = \max\{OPT(j) + 1 : j < i, A[j] < A[i]\}.$$

*Proof.* Sea  $X = (j_1, j_2, \dots, j_k = i)$  una subsecuencia creciente que máxima que termina en  $i$ . Sea  $X' = (j_1, j_2, \dots, j_{k-1})$ . Note que  $j_{k-1} < i$  y  $A[j_{k-1}] < A[i]$ , luego, como  $X'$  es una solución óptima para su correspondiente subproblema por propiedad de subestructura óptima (vea párrafo anterior)

$$|X'| \in \{|X_j| : j < i, A[j] < A[i]\},$$

donde  $X_j$  es una solución óptima que termina en  $j$ . Suponga por contradicción que existe  $j^*$  en dicho rango tal que  $|X_{j^*}| > |X'|$ . En ese caso,  $X_{j^*} \cup \{i\}$  sería una solución mejor que  $X$  para el problema original, una contradicción. Ello demuestra que

$$|X'| = \max\{|X_j| : j < i, A[j] < A[i]\},$$

como queríamos mostrar.  $\square$

Con este lema en mano, diseñaremos un algoritmo de Programación Dinámica.

*Recibe:* Un arreglo  $A[1..n]$ .

*Devuelve:* El tamaño de una subsecuencia creciente máxima .

MAX-SUB-CREC-PROG-DINAMICA( $A$ )

- 1:  $M[0] = 0$
- 2: **for**  $i = 1$  **to**  $n$
- 3:      $M[i] = \max\{M[j] : 0 \leq j < i, A[j] < A[i]\} + 1$
- 4: **return**  $\max\{M[i] : 1 \leq i \leq n\}$

Es claro que el algoritmo tiene tiempo de ejecución  $O(n^2)$ .

Sequence $s_i$	2	4	3	5	1	7	6	9	8
Length $l_i$	1	2	3	3	1	4	4	5	5
Predecessor $p_i$	—	1	1	2	—	4	4	6	6

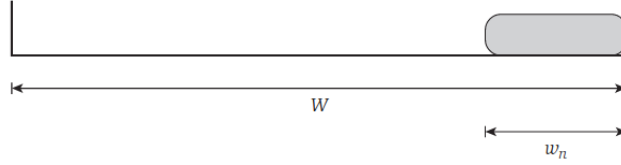
## 5 Subset Sum y Mochila

**Problema Max-Subset-Sum.** Dado un conjunto  $\{1, 2, \dots, n\}$  de items cada uno con un peso natural  $w_i$ , y un número natural  $W$ , encontrar un subconjunto de items cuya suma de pesos es la mayor posible, pero menor o igual a  $W$ .

Note que un algoritmo voraz no funciona:  $A = [12, 10, 9]$ ,  $W = 20$ . Diseñaremos un algoritmo de programación dinámica.

Debemos encontrar un subproblema. Si  $X$  es una solución óptima al problema y el último elemento no está en  $X$  entonces  $X$  es también una solución óptima para los primeros  $n - 1$  elementos (¿Por qué?).

Si el último elemento está en  $X$  entonces podemos asegurar que  $X \setminus \{n\}$  es una solución óptima para los primeros  $n - 1$  elementos *de entre las que tienen peso como máximo  $W - w_n$* .



Por tanto, nuestros subproblemas deben considerar una variable asociada más: el peso máximo. Sea  $OPT(i, W)$  el valor de una solución óptima que usa los items  $\{1, 2, \dots, i\}$  y que tiene peso menor o igual a  $W$ . Sea  $X$  una solución óptima asociada a  $OPT(n, W)$ . Tenemos que

- Si  $n \in X$  entonces  $OPT(n, W) = OPT(n - 1, W - w_n) + w_n$
- Si  $n \notin X$  entonces  $OPT(n, W) = OPT(n - 1, W)$

**Lema 5.1.** Si  $OPT(i, j)$  el valor de una solución óptima que usa los items  $\{1, 2, \dots, i\}$  y que tiene peso menor o igual a  $j$ . Entonces

$$OPT(i, j) = \begin{cases} 0 & \text{si } i = 0 \\ OPT(i - 1, W) & \text{si } i > 0 \text{ y } w_i > W \\ \max\{OPT(i - 1, W - w_i) + w_i, OPT(i - 1, W)\} & \text{si } i > 0 \text{ y } w_i \leq W \end{cases}$$

*Proof.* Ejercicio. □

A partir de la recurrencia anterior, podemos diseñar el siguiente algoritmo de programación dinámica.

*Recibe:* Un arreglo  $w[1..n]$  de números naturales (pesos) y un número natural  $W$ .

*Devuelve:* Una solución óptima para el problema Subset-Sum.

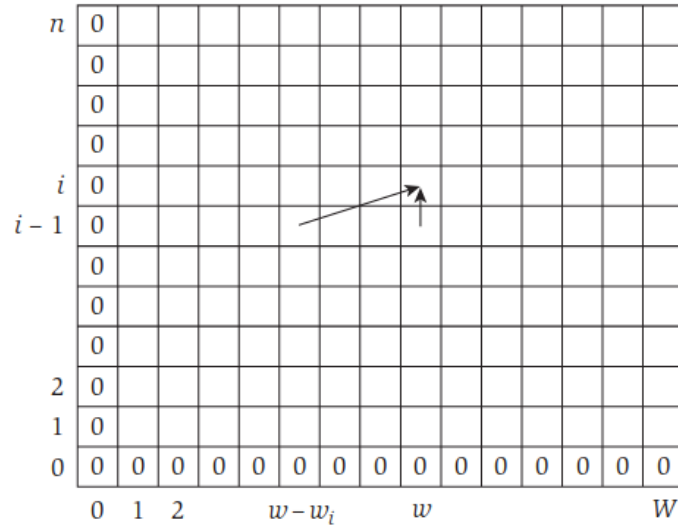
SUBSETSUM-PROGDINAMICA( $w, W$ )

```

1: for  $j = 0$  to  $W$ 
2:    $M[0, j] = 0$ 
3: for  $i = 1$  to  $n$ 
4:   for  $j = 0$  to  $W$ 
5:     if  $w[i] > j$ 
6:        $M[i, j] = M[i - 1, j]$ 
7:     else
8:        $M[i, j] = \max\{M[i - 1, j], M[i - 1, j - w[i]] + w[i]\}$ 
9: return  $M[n][W]$ 

```

Es claro que el algoritmo tiene tiempo de ejecución  $O(nW)$  (pseudopolinomial).



Knapsack size  $W = 6$ , items  $w_1 = 2, w_2 = 2, w_3 = 3$

3							
2							
1							
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Initial values

3							
2							
①	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Filling in values for  $i = 1$

3							
②	0	0	2	2	4	4	4
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Filling in values for  $i = 2$

③	0	0	2	3	4	5	5
2	0	0	2	2	4	4	4
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Filling in values for  $i = 3$

**Ejercicio 5.1.** Diseñe un algoritmo para obtener una solución óptima (no solamente el valor de dicha solución)

## Problema de la mochila

Suponga que además del vector de pesos, también recibimos un vector  $v$  de valores para cada ítem. Siendo el objetivo maximizar el valor de una solución restringiendo el peso.

Un análisis similar al anterior nos permite deducir que si  $OPT(i, W)$  es el valor de una solución óptima que usa los ítems  $\{1, 2, \dots, i\}$  y que tiene peso menor o igual a  $W$ . Sea  $X$  una solución óptima para el problema  $OPT(n, W)$ . Tenemos que

- Si  $n \in X$  entonces  $OPT(n, W) = OPT(n - 1, W - w_n) + v_n$
- Si  $n \notin X$  entonces  $OPT(n, W) = OPT(n - 1, W)$

**Ejercicio 5.2.** Diseñe un algoritmo para el problema de la mochila.

## 6 Partición lineal justa

Dado un arreglo  $A[1..n]$  de números no negativos, una *partición lineal* de  $A$  es una secuencia de índices  $P = (i_1, i_2, \dots, i_{k+1})$  donde  $1 = i_1 < i_2 < \dots < i_{k+1} = n$ .

Decimos que dicha partición tiene tamaño  $k$ . El peso de dicha partición, viene dado por

$$w(P) = \max\left\{\sum_{j=i_1}^{i_2} A[j], \sum_{j=i_2+1}^{i_3} A[j], \dots, \sum_{j=i_k+1}^{i_{k+1}} A[j]\right\}$$

Por ejemplo, si  $A = [10, 20, 30, 40, 50, 60, 70, 80, 90]$ , una partición podría ser  $[1, 3, 6, 9]$  y el peso de dicha partición sería  $\max\{10 + 20 + 30, 40 + 50 + 60, 70 + 80 + 90\} = 240$ .

**Problema Min-Partition.** Dado un arreglo  $A$  de números enteros no negativos y un número  $k$ , encontrar una partición con tamaño  $k$  de peso mínimo.

Sea  $OPT(n, k)$  el valor de una partición óptima.

Note que

$$OPT(n, k) = \begin{cases} \sum_{i=1}^n A[i] & \text{si } k = 1 \\ \min_{\ell=k}^n \max\{OPT(\ell-1, k-1), \sum_{j=\ell}^n A[j]\} & \text{en otro caso} \end{cases}$$

Podemos diseñar un algoritmo de programación dinámica.

```

MINPARTITION-PROGDINAMICA( $A, k$ )
1: for  $i = 1$  to  $n$ 
2:    $M[i, 1] = \sum_{p=1}^i A[p]$ 
3: for  $i = 2$  to  $n$ 
4:   for  $j = 2$  to  $k$ 
5:      $M[i, j] = \infty$ 
6:     for  $\ell = j$  to  $i$ 
7:        $cost = \max\{M[\ell - 1][j - 1], \sum_{p=\ell}^i A[p]\}$ 
8:       if  $cost < M[i, j]$ 
9:          $M[i, j] = cost$ 
10: return  $M[n][k]$ 

```

Es claro que el tiempo de ejecución es  $O(n^2k)$ .