


# **Programa de teoría**

## **Parte I. Estructuras de Datos.**

1. Abstracciones y especificaciones.
2. Conjuntos y diccionarios.
3. Representación de conjuntos mediante árboles.
4. Grafos.

## **Parte II. Algorítmica.**

1. Análisis de algoritmos.
2. Divide y vencerás.
3. Algoritmos voraces.
-  **4. Programación dinámica.**
5. Backtracking.
6. Ramificación y poda.

A.E.D.

Tema 4. Programación dinámica.

1

# **PARTE II: ALGORÍTMICA**

## **Tema 4. Programación dinámica.**

- 4.1. Método general.
- 4.2. Análisis de tiempos de ejecución.
- 4.3. Ejemplos de aplicación.
  - 4.3.1. Problema de la mochila 0/1.
  - 4.3.2. Problema del cambio de monedas.

A.E.D.

Tema 4. Programación dinámica.

2

## 4.1. Método general.

- La base de la **programación dinámica** es el **razonamiento inductivo**: ¿cómo resolver un problema combinando soluciones para problemas más pequeños?
- La idea es la misma que en **divide y vencerás...** pero aplicando una estrategia distinta.
- **Similitud**:
  - Descomposición recursiva del problema.
  - Se obtiene aplicando un razonamiento inductivo.
- **Diferencia**:
  - Divide y vencerás: aplicar directamente la fórmula recursiva (programa recursivo).
  - Programación dinámica: resolver primero los problemas más pequeños, guardando los resultados en una tabla (programa iterativo).

A.E.D.

3

Tema 4. Programación dinámica.

## 4.1. Método general.

- **Ejemplo. Cálculo de los números de Fibonacci.**

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

- **Con divide y vencerás.**

**operación Fibonacci (n: entero): entero**

**si  $n \leq 2$  entonces devolver 1**

**sino devolver Fibonacci(n-1) + Fibonacci(n-2)**

- **Con programación dinámica.**

**operación Fibonacci (n: entero): entero**

**$T[1] := 1; T[2] := 1$**

**para  $i := 3, \dots, n$  hacer**

**$T[i] := T[i-1] + T[i-2]$**

**devolver  $T[n]$**

A.E.D.

4

Tema 4. Programación dinámica.



## 4.1. Método general.

- **Ejemplo. Algoritmo de Floyd**, para calcular los caminos mínimos entre cualquier par de nodos de un grafo.
- **Razonamiento inductivo**: para calcular los caminos mínimos pudiendo pasar por los **k** primeros nodos usamos los caminos mínimos pasando por los **k-1** primeros.
- **$D_k(i, j)$** : camino mínimo de **i** a **j** pudiendo pasar por los nodos 1, 2, ..., **k**.

$$D_k(i, j) = \begin{cases} C[i, j] & \text{Si } k=0 \\ \min(D_{k-1}(i, j), D_{k-1}(i, k) + D_{k-1}(k, j)) & \text{Si } k>0 \end{cases}$$

$D_n(i, j)$  → caminos mínimos finales

A.E.D.

Tema 4. Programación dinámica.

7

## 4.1. Método general.

- **Ejemplo. Algoritmo de Floyd.**
- Aplicación de la fórmula:
  - Empezar por el problema pequeño: **k = 0**
  - Avanzar hacia problemas más grandes: **k = 1, 2, 3, ...**
- ¿Cómo se garantiza que un algoritmo de programación dinámica obtiene la solución correcta?
- Una descomposición es correcta si cumple el **Principio de optimalidad de Bellman**:

La solución óptima de un problema se obtiene combinando soluciones óptimas de subproblemas.

A.E.D.

Tema 4. Programación dinámica.

8

## 4.1. Método general.

- O bien: cualquier subsecuencia de una secuencia óptima debe ser, a su vez, una secuencia óptima.
- **Ejemplo.** Si el camino mínimo de **A** a **B** pasa por **C**, entonces los trozos de camino de **A** a **C**, y de **C** a **B** deben ser también mínimos.
- **Ojo:** el principio no siempre es aplicable.
- **Contraejemplo.** Si el camino simple más largo de **A** a **B** pasa por **C**, los trozos de **A** a **C** y de **C** a **B** no tienen por qué ser soluciones óptimas.

## 4.1. Método general.

### **Pasos para aplicar programación dinámica:**

- 1) Obtener una **descomposición recurrente** del problema:
    - Ecuación recurrente.
    - Casos base.
  - 2) Definir la **estrategia** de aplicación de la fórmula:
    - Tablas utilizadas por el algoritmo.
    - Orden y forma de rellenarlas.
  - 3) Especificar cómo se **recompone la solución** final a partir de los valores de las tablas.
- **Punto clave:** obtener la descomposición recurrente.
  - Requiere mucha “creatividad”...

## 4.1. Método general.

- Cuestiones a resolver en el razonamiento inductivo:
  - ¿Cómo reducir un problema a subproblemas más simples?
  - ¿Qué parámetros determinan el **tamaño del problema** (es decir, cuándo el problema es “más simple”)?
- **Idea:** ver lo que ocurre al tomar una decisión concreta  
→ interpretar el problema como un proceso de toma de decisiones.
- **Ejemplos. Floyd.** Decisiones: Pasar o no pasar por un nodo intermedio.
- **Mochila 0/1.** Decisiones: coger o no coger un objeto dado.

A.E.D.

Tema 4. Programación dinámica.

11

## 4.2. Análisis de tiempos de ejecución.

- La programación dinámica se basa en el uso de **tablas** donde se almacenan los resultados parciales.
- En general, el **tiempo** será de la forma:  
Tamaño de la tabla\*Tiempo de rellenar cada elemento de la tabla.
- Un aspecto **importante** es la memoria puede llegar a ocupar la tabla.
- Además, algunos de estos cálculos pueden ser innecesarios.

A.E.D.

Tema 4. Programación dinámica.

12

## 4.3. Ejemplos de aplicación.

### 4.3.1. Problema de la mochila 0/1.

- Como el problema de la mochila, pero los objetos no se pueden partir (se cogen enteros o nada).

- **Datos del problema:**

- **n**: número de objetos disponibles.
- **M**: capacidad de la mochila.
- **p** = ( $p_1, p_2, \dots, p_n$ ) pesos de los objetos.
- **b** = ( $b_1, b_2, \dots, b_n$ ) beneficios de los objetos.

- **Formulación matemática:**

Maximizar  $\sum_{i=1..n} x_i b_i$ ; sujeto a la restricción  $\sum_{i=1..n} x_i p_i \leq M$ , y  $x_i \in \{0, 1\}$

A.E.D.

Tema 4. Programación dinámica.

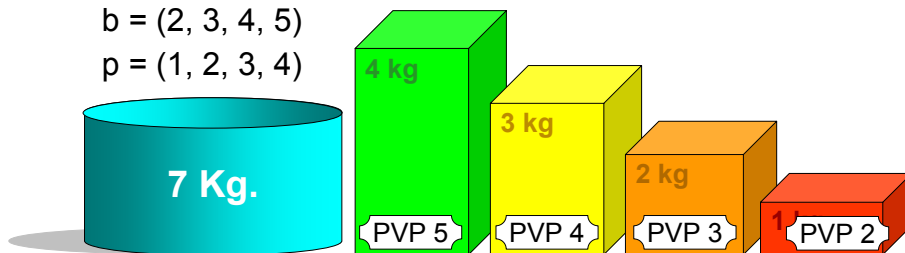
13

### 4.3.1. Problema de la mochila 0/1.

- **Ejemplo:**  $n = 4$ ;  $M = 7$

$b = (2, 3, 4, 5)$

$p = (1, 2, 3, 4)$



- ¿Qué solución devuelve el algoritmo voraz para el problema de la mochila?
- ¿Qué solución devuelve el algoritmo voraz adaptado al caso 0/1 (o se coge un objeto entero o no)?
- ¿Cuál es la solución óptima?
- **Ojo:** el problema es un NP-completo clásico.

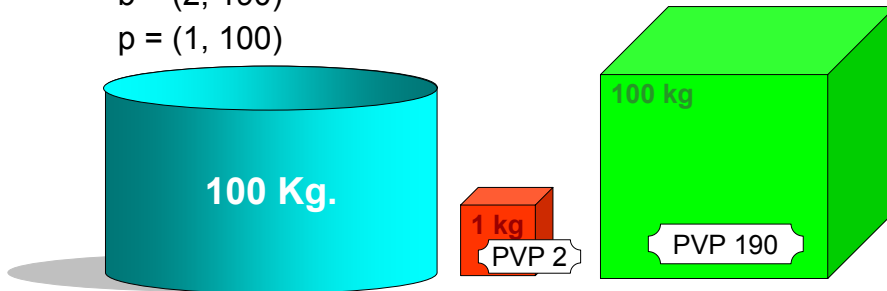
A.E.D.

Tema 4. Programación dinámica.

14

### 4.3.1. Problema de la mochila 0/1.

- **Ejemplo:**  $n = 2$ ;  $M = 100$   
 $b = (2, 190)$   
 $p = (1, 100)$



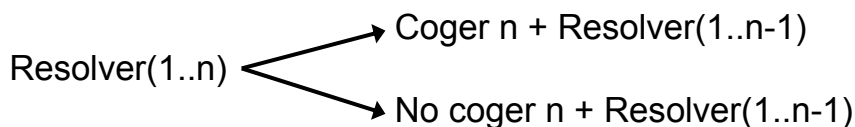
- ¿Qué solución devuelve el algoritmo voraz para el problema de la mochila?
- ¿Qué solución devuelve el algoritmo voraz adaptado al caso 0/1 (o se coge un objeto entero o no)?
- ¿Cuál es la solución óptima?

### 4.3.1. Problema de la mochila 0/1.

- Aplicamos programación dinámica al problema...

#### Paso 1)

- ¿Cómo obtener la descomposición recurrente?
- Interpretar el problema como un **proceso de toma de decisiones**: coger o no coger cada objeto.
- Después de tomar una decisión particular sobre un objeto, nos queda un problema de menor tamaño (con un objeto menos).
- ¿Coger o no coger un objeto?





### 4.3.1. Problema de la mochila 0/1.

- ¿Coger o no coger un objeto  $k$ ?
  - **Si se coge**: tenemos el beneficio  $b_k$ , pero en la mochila queda menos espacio,  $p_k$ .
  - **Si no se coge**: tenemos el mismo problema pero con un objeto menos por decidir.
- ¿Qué varía en los subproblemas?
  - Número de objetos por decidir.
  - Peso disponible en la mochila.
- **Ecuación del problema. Mochila( $k$ ,  $m$ : entero): entero**  
Problema de la mochila 0/1, considerando sólo los  $k$  primeros objetos (de los  $n$  originales) con capacidad de mochila  $m$ . Devuelve el valor de beneficio total.

A.E.D.

Tema 4. Programación dinámica.

17

### 4.3.1. Problema de la mochila 0/1.

- **Definición de Mochila( $k$ ,  $m$ : entero): entero**
  - **Si no se coge el objeto  $k$** :  
 $Mochila(k, m) = Mochila(k - 1, m)$
  - **Si se coge**:  
 $Mochila(k, m) = b_k + Mochila(k - 1, m - p_k)$
  - **Valor óptimo**: el que dé mayor beneficio:  
 $Mochila(k, m) = \max \{ Mochila(k - 1, m), b_k + Mochila(k - 1, m - p_k) \}$
- **Casos base**:
  - Si  $m=0$ , no se pueden incluir objetos:  $Mochila(k, 0) = 0$
  - Si  $k=0$ , tampoco se pueden incluir:  $Mochila(0, m) = 0$
  - ¿Y si  $m$  o  $k$  son negativos?

A.E.D.

Tema 4. Programación dinámica.

18

### 4.3.1. Problema de la mochila 0/1.

- **Casos base:**

- Si  $m$  o  $k$  son negativos, el problema es irresoluble:  
 $Mochila(k, m) = -\infty$

- **Resultado.** La siguiente ecuación obtiene la solución óptima del problema:

$$Mochila(k, m) = \begin{cases} 0 & \text{Si } k=0 \text{ ó } m=0 \\ -\infty & \text{Si } k<0 \text{ ó } m<0 \\ \max \{Mochila(k-1, m), b_k + Mochila(k-1, m-p_k)\} & \end{cases}$$

- ¿Cómo aplicarla de forma ascendente?
- Usar una tabla para guardar resultados de los subprob.
- Rellenar la tabla: empezando por los casos base, avanzar a tamaños mayores.

A.E.D.

19

Tema 4. Programación dinámica.

### 4.3.1. Problema de la mochila 0/1.

#### Paso 2) Definición de las tablas y cómo rellenarlas

##### 2.1) Dimensiones y tamaño de la tabla

- Definimos la tabla  $V$ , para guardar los resultados de los subproblemas:  $V[i, j] = Mochila(i, j)$
- La solución del problema original es  $Mochila(n, M)$ .
- Por lo tanto, la tabla debe ser:  
**V: array [0..n, 0..M] de entero**
  
- Fila 0 y columna 0: casos base de valor 0.
- Los valores que caen fuera de la tabla son casos base de valor  $-\infty$ .

A.E.D.

20

Tema 4. Programación dinámica.

### 4.3.1. Problema de la mochila 0/1.

#### 2.2) Forma de rellenar las tablas:

- Inicializar los casos base:  
 $V[i, 0] := 0; V[0, j] := 0$
- Para todo  $i$  desde 1 hasta  $n$   
    Para todo  $j$  desde 1 hasta  $M$ , aplicar la ecuación:  
         $V[i, j] := \max (V[i-1, j], b_i + V[i-1, j-p_i])$
- El beneficio óptimo es  $V[n, M]$

**Ojo:** si  $j-p_i$  es negativo, entonces es el caso  $-\infty$ , y el máximo será siempre el otro término.

A.E.D.

21

Tema 4. Programación dinámica.

### 4.3.1. Problema de la mochila 0/1.

- **Ejemplo.**  $n= 3, M= 6, p= (2, 3, 4), b= (1, 2, 5)$

	j						
V	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
i 1	0	0	1	1	1	1	1
2	0	0	1	2	2	3	3
3	0	0	1	2	5	5	6

- ¿Cuánto es el orden de complejidad del algoritmo?

A.E.D.

22

Tema 4. Programación dinámica.

### 4.3.1. Problema de la mochila 0/1.

#### Paso 3) Reconponer la solución óptima

- $V[n, M]$  almacena el beneficio óptimo, pero ¿cuál son los objetos que se cogen en esa solución?
- Obtener la tupla solución  $(x_1, x_2, \dots, x_n)$  usando  $V$ .
- **Idea:** partiendo de la posición  $V[n, M]$ , analizar las decisiones que se tomaron para cada objeto  $i$ .
  - Si  $V[i, j] = V[i-1, j]$ , entonces la solución no usa el objeto  $i$   
 $\rightarrow x_i = 0$
  - Si  $V[i, j] = V[i-1, j-p_i] + b_i$ , entonces sí se usa el objeto  $i$   
 $\rightarrow x_i = 1$
  - Si se cumplen ambas, entonces podemos usar el objeto  $i$  o no (existe más de una solución óptima).

A.E.D.

Tema 4. Programación dinámica.

23

### 4.3.1. Problema de la mochila 0/1.

#### 3) Cómo reconponer la solución óptima

```
j:= P
para i:= n, ..., 1 hacer
    si  $V[i, j] == V[i-1, j]$  entonces
         $x[i] := 0$ 
    sino //  $V[i, j] == V[i-1, j-p_i] + b_i$ 
         $x[i] := 1$ 
         $j := j - p_i$ 
    finsi
finpara
```

- Aplicar sobre el ejemplo anterior.

A.E.D.

Tema 4. Programación dinámica.

24

### 4.3.1. Problema de la mochila 0/1.

- ¿Cuánto será el tiempo de recomponer la solución?
- ¿Cómo es el tiempo en relación al algoritmo de backtracking y al de ramificación y poda?
- ¿Qué pasa si multiplicamos todos los pesos por 1000?
- ¿Se cumple el principio de optimalidad?

### 4.3.2. Problema del cambio de monedas.

- **Problema:** Dado un conjunto de  $n$  tipos de monedas, cada una con valor  $c_i$ , y dada una cantidad  $P$ , encontrar el número mínimo de monedas que tenemos que usar para obtener esa cantidad.
- El algoritmo voraz es muy eficiente, pero sólo funciona en un número limitado de casos.
- **Utilizando programación dinámica:**
  - 1) Definir el problema en función de problemas más pequeños
  - 2) Definir las tablas de subproblemas y la forma de rellenarlas
  - 3) Establecer cómo obtener el resultado a partir de las tablas

### 4.3.2. Problema del cambio de monedas.

#### 1) Descomposición recurrente del problema

- Interpretar como un problema de toma de decisiones.

- ¿Coger o no coger **una** moneda de tipo **k**?

→ **Si se coge**: usamos 1 más y tenemos que devolver cantidad  $c_k$  menos.

→ **Si no se coge**: tenemos el mismo problema pero descartando la moneda de tipo **k**.

- ¿Qué varía en los subproblemas?
  - Tipos de monedas a usar.
  - Cantidad por devolver.
- **Ecuación del problema. Cambio(k, q: entero): entero**  
Problema del cambio de monedas, considerando sólo los **k** primeros tipos, con cantidad a devolver **q**. Devuelve el número mínimo de monedas necesario.

A.E.D.

27

Tema 4. Programación dinámica.

### 4.3.2. Problema del cambio de monedas.

- **Definición de Cambio(k, q: entero): entero**

- **Si no se coge ninguna moneda de tipo k:**

$$\text{Cambio}(k, q) = \text{Cambio}(k - 1, q)$$

- **Si se coge 1 moneda de tipo k:**

$$\text{Cambio}(k, q) = 1 + \text{Cambio}(k, q - c_k)$$

- **Valor óptimo**: el que use menos monedas:

$$\text{Cambio}(k, q) = \min \{ \text{Cambio}(k - 1, q), \\ 1 + \text{Cambio}(k, q - c_k) \}$$

- **Casos base:**

- Si  $q=0$ , no usar ninguna moneda:  $\text{Cambio}(k, 0) = 0$

- En otro caso, si  $q < 0$  ó  $k \leq 0$ , no se puede resolver el problema:  $\text{Cambio}(q, k) = +\infty$

A.E.D.

28

Tema 4. Programación dinámica.

### 4.3.2. Problema del cambio de monedas.

- Ecuación recurrente:

$$\text{Cambio}(k, q) = \begin{cases} 0 & \text{Si } q=0 \\ +\infty & \text{Si } q < 0 \text{ ó } k \leq 0 \\ \min \{ \text{Cambio}(k-1, q), 1 + \text{Cambio}(k, q-c_k) \} & \end{cases}$$

#### 2) Aplicación ascendente mediante tablas

- Matriz **D**  $\rightarrow D[i, j] = \text{Cambio}(i, j)$
- **D**: array [1..n, 0..P] de entero

```

para i:= 1, ..., n hacer D[i, 0]:= 0
para i:= 1, ..., n hacer
  para j:= 1, ..., P hacer
    D[i, j]:= min(D[i-1, j], 1+D[i, j-ci])
devolver D[n, P]
    
```

Ojo si cae fuera de la tabla.

### 4.3.2. Problema del cambio de monedas.

- Ejemplo.  $n= 3, P= 8, c= (1, 4, 6)$

		j								
		0	1	2	3	4	5	6	7	8
i	1 $c_1=1$	0	1	2	3	4	5	6	7	8
	2 $c_2=4$	0	1	2	3	1	2	3	4	2
	3 $c_3=6$	0	1	2	3	1	2	1	2	2

- ¿Cuánto es el orden de complejidad del algoritmo?
- ¿Cómo es en comparación con el algoritmo voraz?

### 4.3.2. Problema del cambio de monedas.

#### 3) Cómo recomponer la solución a partir de la tabla

- ¿Cómo calcular cuántas monedas de cada tipo deben usarse, es decir, la tupla solución  $(x_1, x_2, \dots, x_n)$ ?
- Analizar las decisiones tomadas en cada celda, empezando en  $D[n, P]$ .
- ¿Cuál fue el mínimo en cada  $D[i, j]$ ?
  - $D[i - 1, j] \rightarrow$  No utilizar ninguna moneda más de tipo  $i$ .
  - $D[i, j - C[i]] + 1 \rightarrow$  Usar una moneda más de tipo  $i$ .
- Implementación:  
**x: array [1..n] de entero**  
 $\rightarrow x[i] =$  número de monedas usadas de tipo  $i$

A.E.D.

31

Tema 4. Programación dinámica.

### 4.3.2. Problema del cambio de monedas.

#### 3) Cómo recomponer la solución a partir de la tabla

$x := (0, 0, \dots, 0)$

$i := n$

$j := P$

**mientras**  $(i \neq 0)$  AND  $(j \neq 0)$  **hacer**

**si**  $D[i, j] == D[i-1, j]$  **entonces**

$i := i - 1$

**sino**

$x[i] := x[i] + 1$

$j := j - C_i$

**fin si**

**fin mientras**

- ¿Qué pasa si hay varias soluciones óptimas?
- ¿Y si no existe ninguna solución válida?

A.E.D.

32

Tema 4. Programación dinámica.



## 4. Programación dinámica.

### Conclusiones

- El **razonamiento inductivo** es una herramienta muy potente en resolución de problemas.
- Aplicable no sólo en problemas de optimización.
- ¿Cómo obtener la fórmula? Interpretar el problema como una serie de **toma de decisiones**.
- Descomposición recursiva no necesariamente implica implementación recursiva.
- **Programación dinámica**: almacenar los resultados en una tabla, empezando por los tamaños pequeños y avanzando hacia los más grandes.