


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.

PARTE II: ALGORÍTMICA

Tema 4. Programación dinámica.

4.1. Método general.

4.2. Ejemplos de aplicación.

4.2.1. Problema de la mochila 0/1.

4.2.2. Problema del cambio de monedas.

4.2.3. Problemas para profundizar.

4.3. Análisis de tiempos de ejecución.

RE- WHAT?

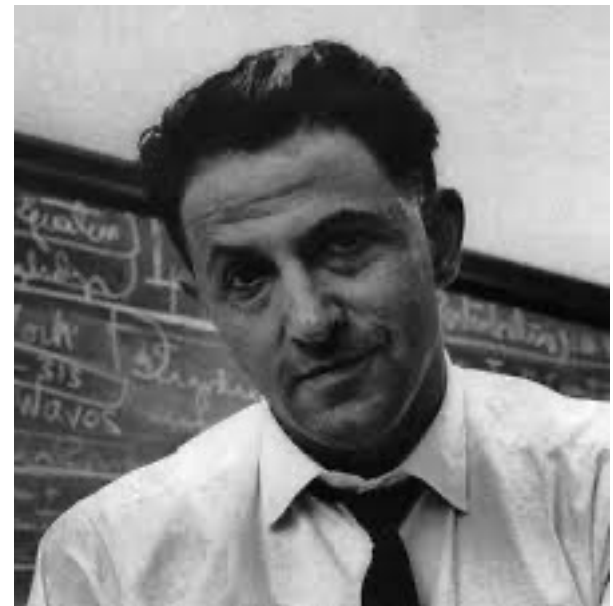
jefe



I'm doing re-...ehm
Dynamic
Programming



Richard
Bellman



4.0. La idea...

- Cuando solución a un problema viene dada por **ecuación recurrente**, esta se puede aplicar de forma:
 - Descendente (como DyV)
 - Ascendente (PD...)
- Ejemplo: serie de Fibonacci...

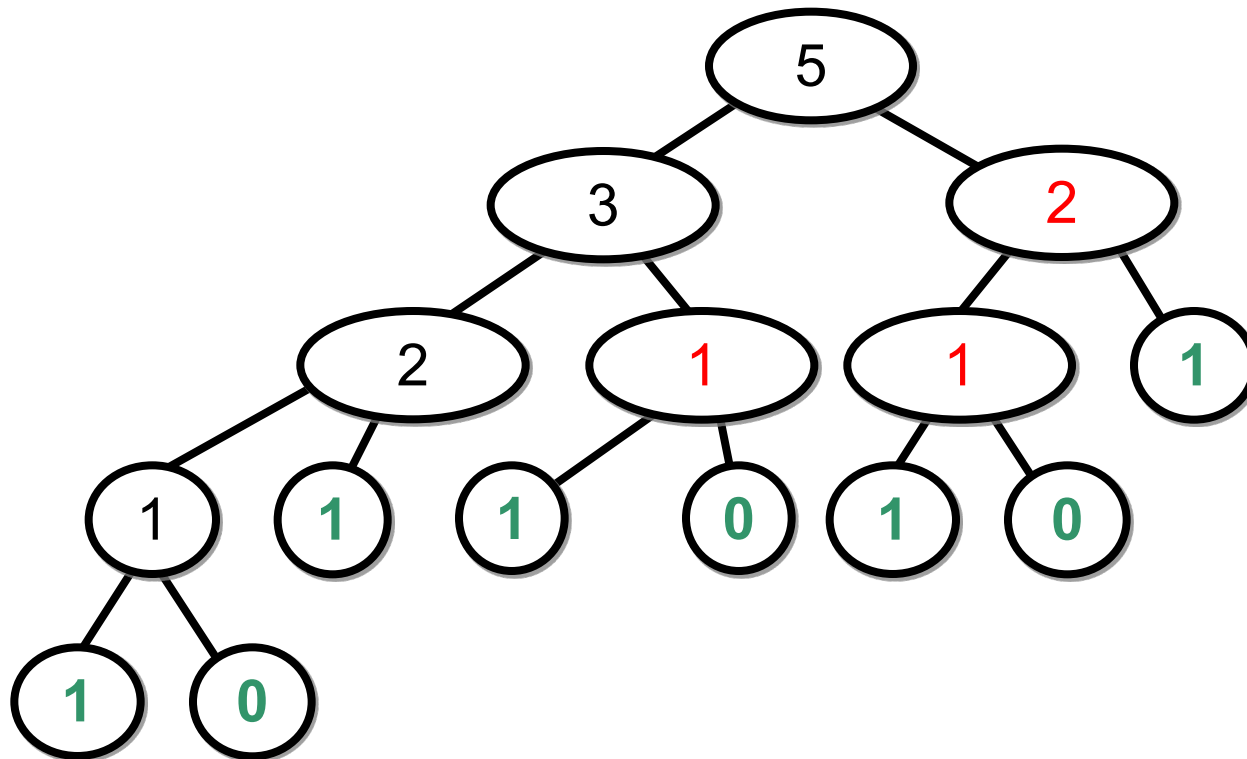
$$F(n) = \begin{cases} 0 & n = 1 \\ 1 & n = 2 \\ F(n-1) + F(n-2) & \text{Si } n > 2 \end{cases}$$

¿F(5)?

4.0. La idea...

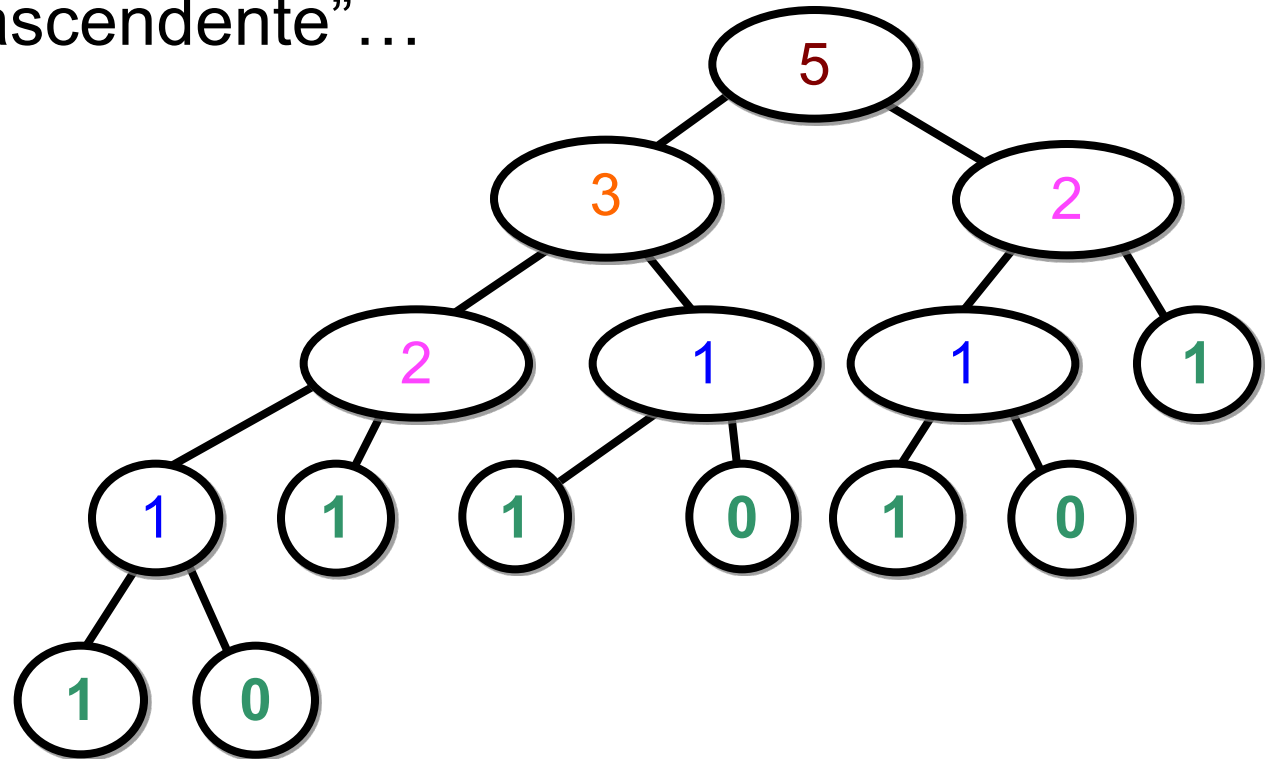
- $F(5)$, cálculo descendente...

0	1	1	2	3	5
---	---	---	---	---	---



4.0. La idea...

F(5), cálculo “ascendente”...



4.0. La idea...

$F(5)$, cálculo “ascendente”...

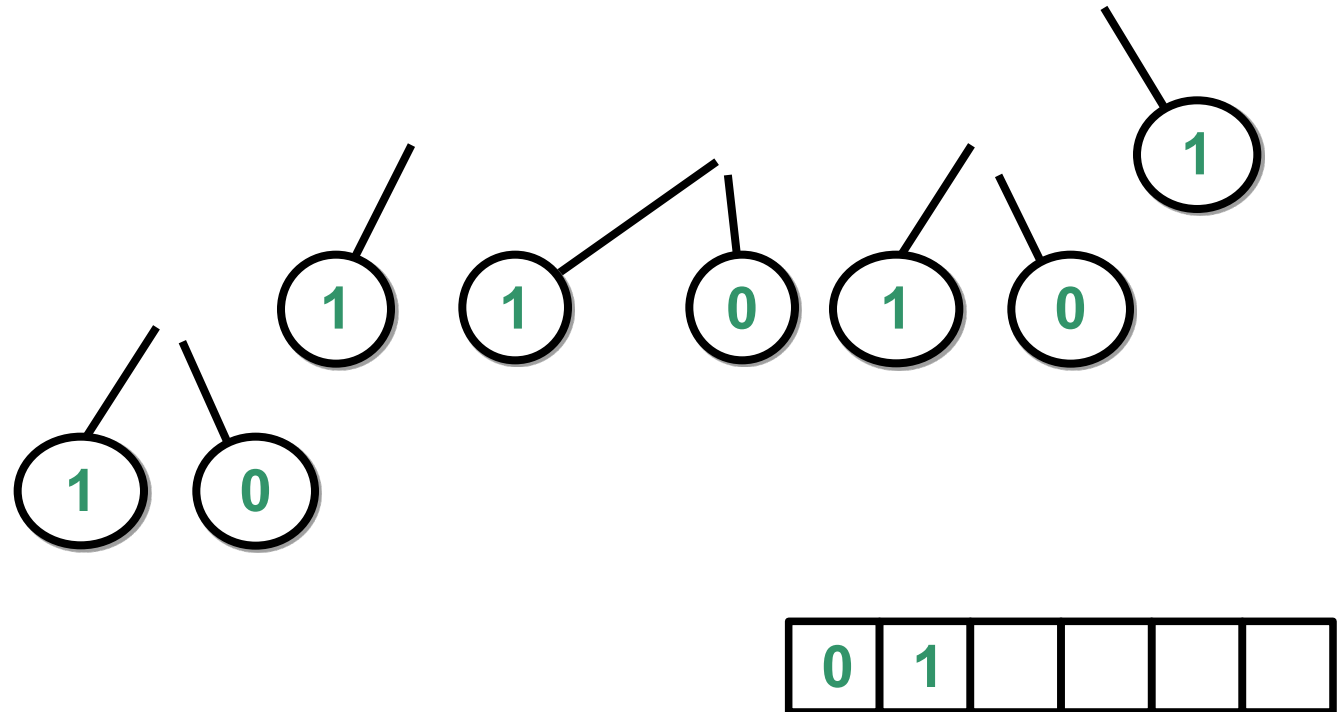
>> Podemos guardar cada $F(x)$ en posición de tabla T , de forma que $F(x) = T(x)$

>> Eso permitiría calcular cada $F(x)$ una sola vez, guardar resultado, y usarlo cuando sea necesario (varias veces)

Cada $F(x)$ va a ÚNICA posición $T(x)$, aunque aparezca varias veces en el árbol... calcular 1 vez, usar n veces...

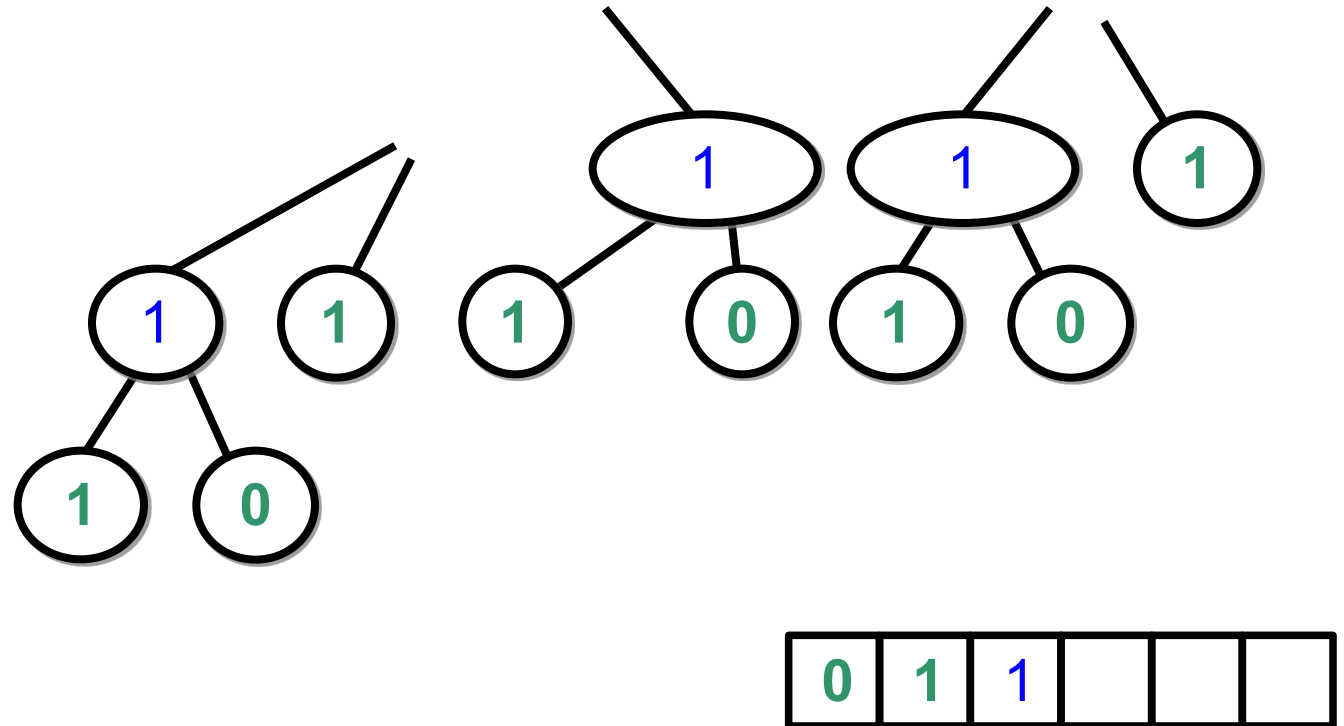
4.0. La idea...

$F(5)$, cálculo “ascendente”...



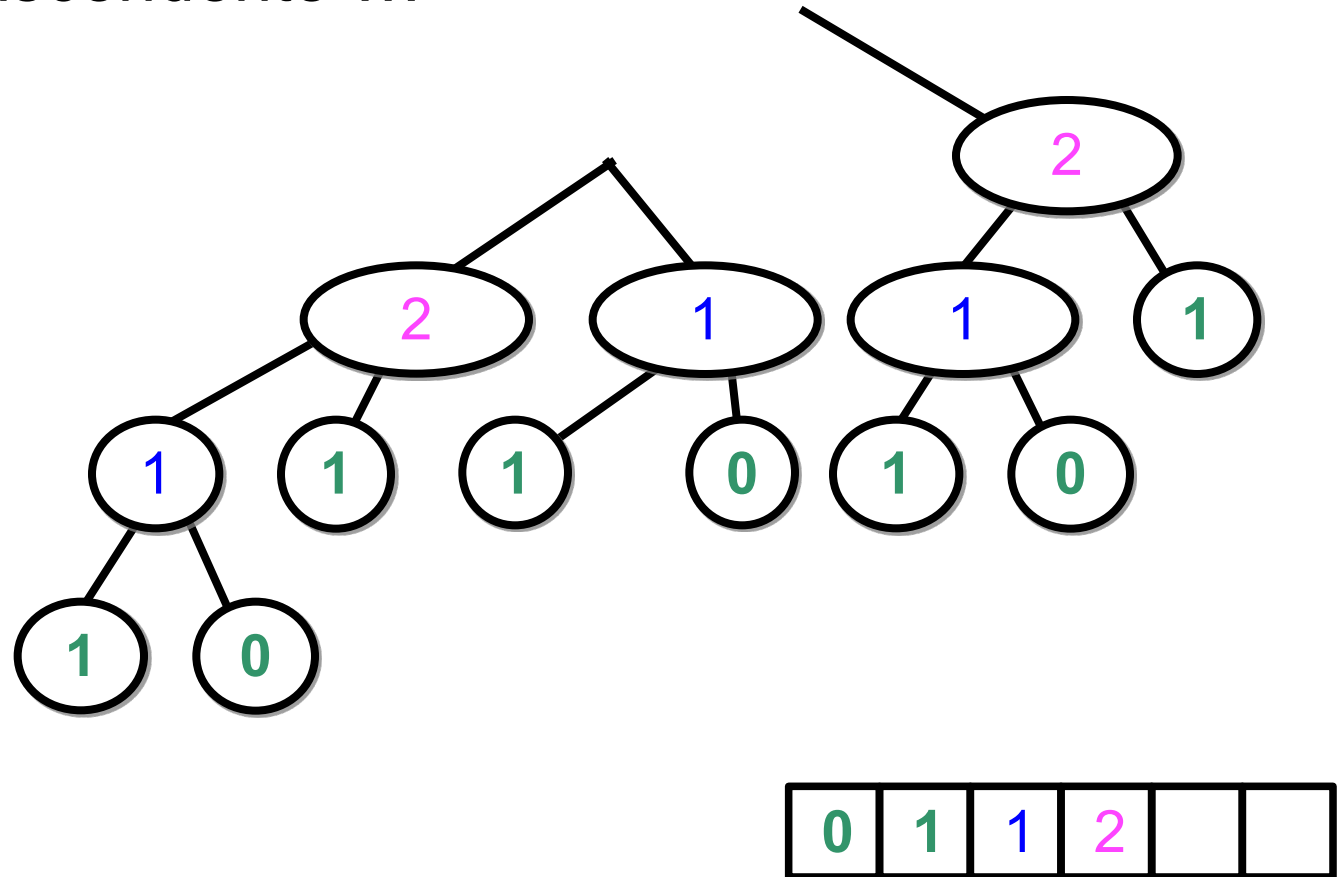
4.0. La idea...

F(5), cálculo “ascendente”...



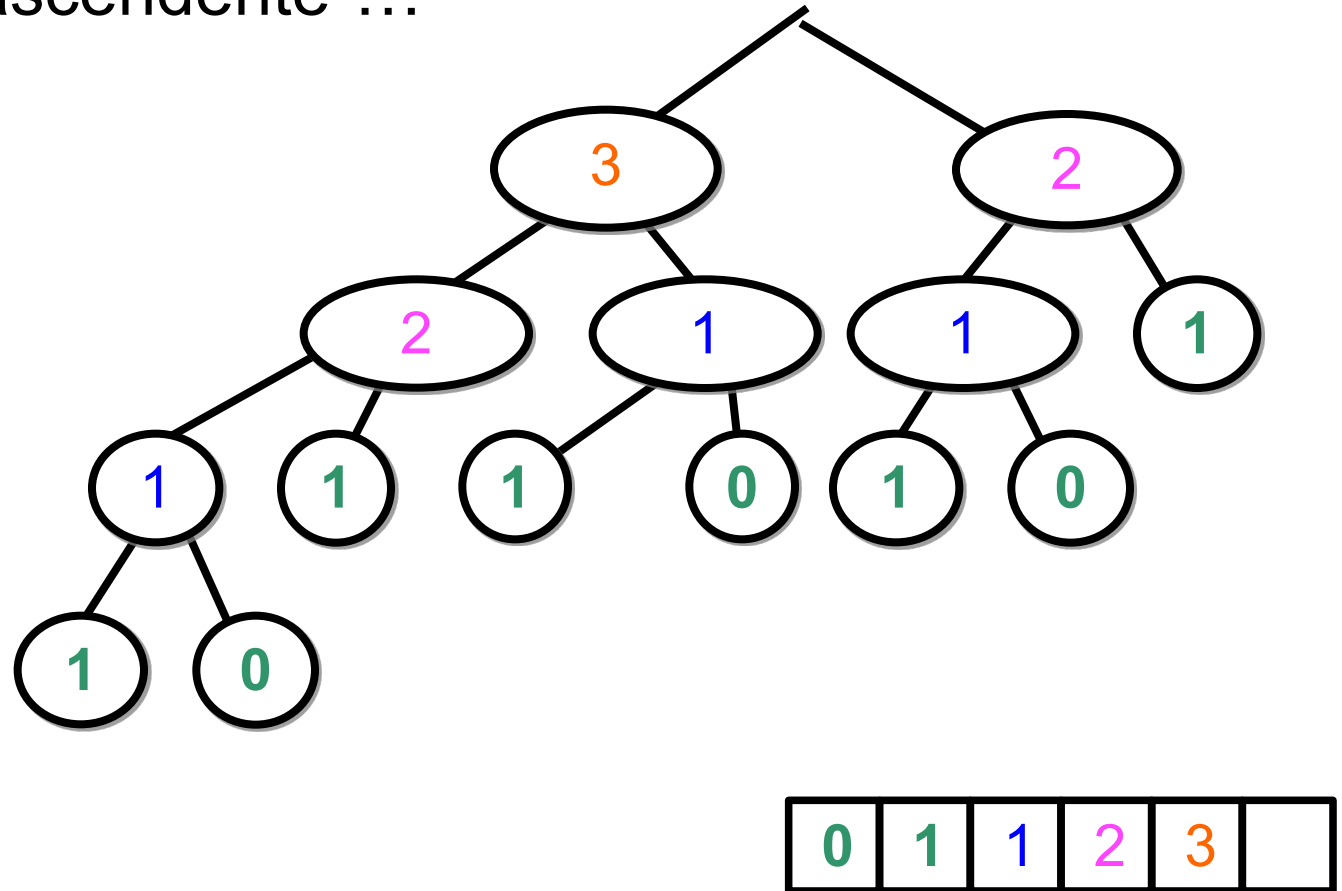
4.0. La idea...

$F(5)$, cálculo “ascendente”...



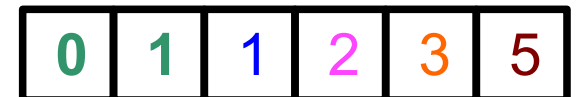
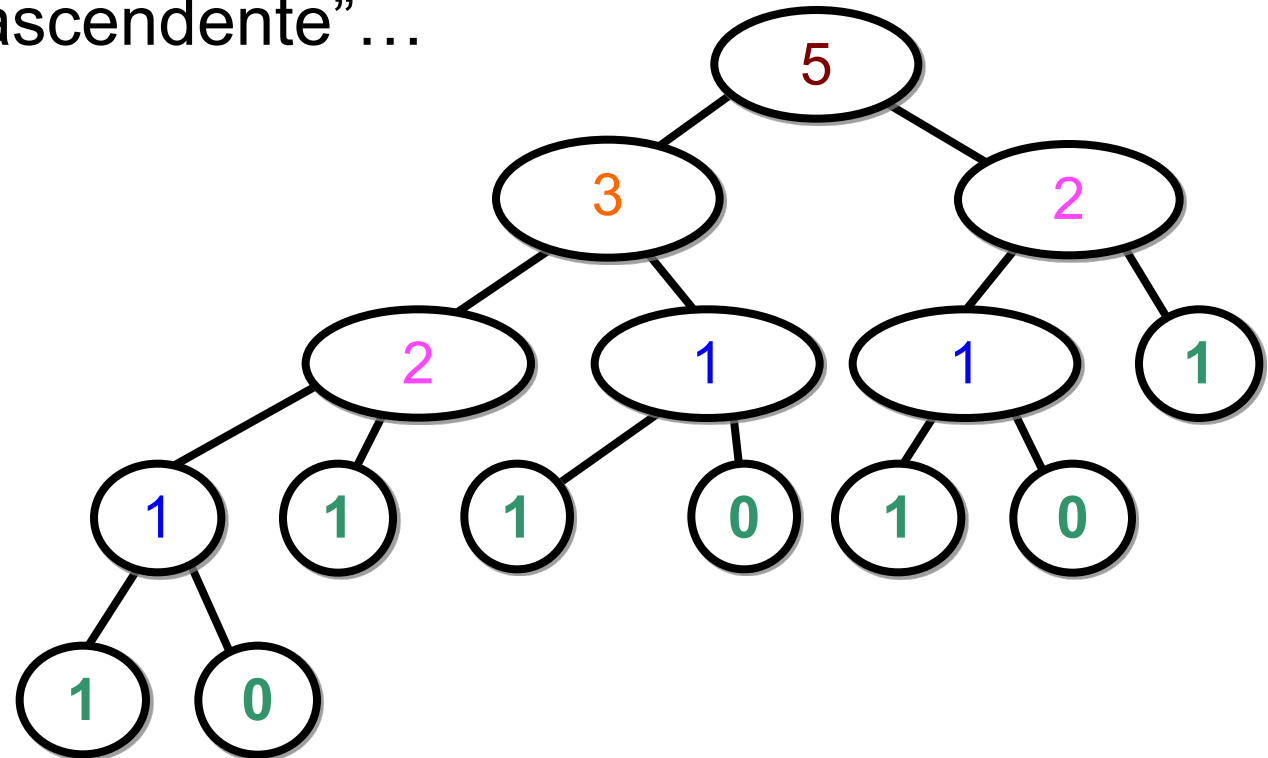
4.0. La idea...

$F(5)$, cálculo “ascendente”...



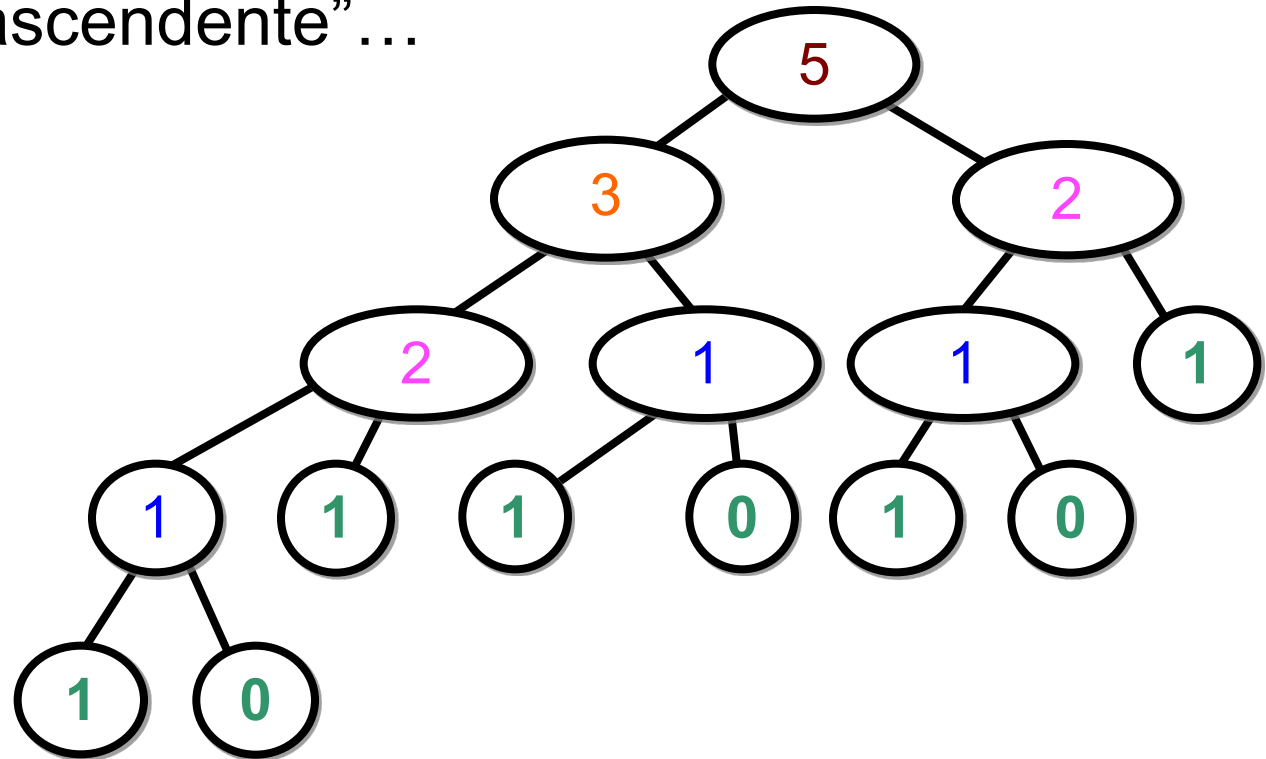
4.0. La idea...

F(5), cálculo “ascendente”...



4.0. La idea...

F(5), cálculo “ascendente”...



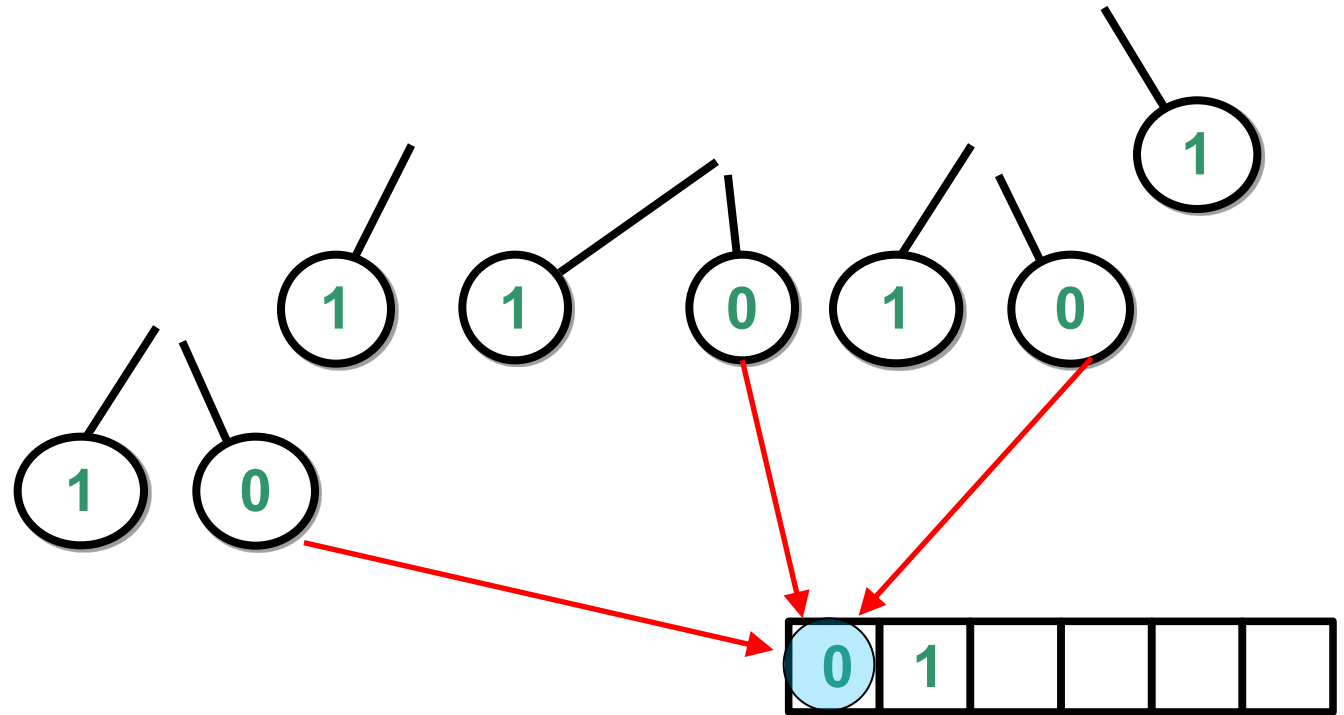
¡¡En realidad el árbol no hace falta!!

0	1	1	2	3	5
---	---	---	---	---	---

Hacer cálculos directamente en tabla, **más eficiente**

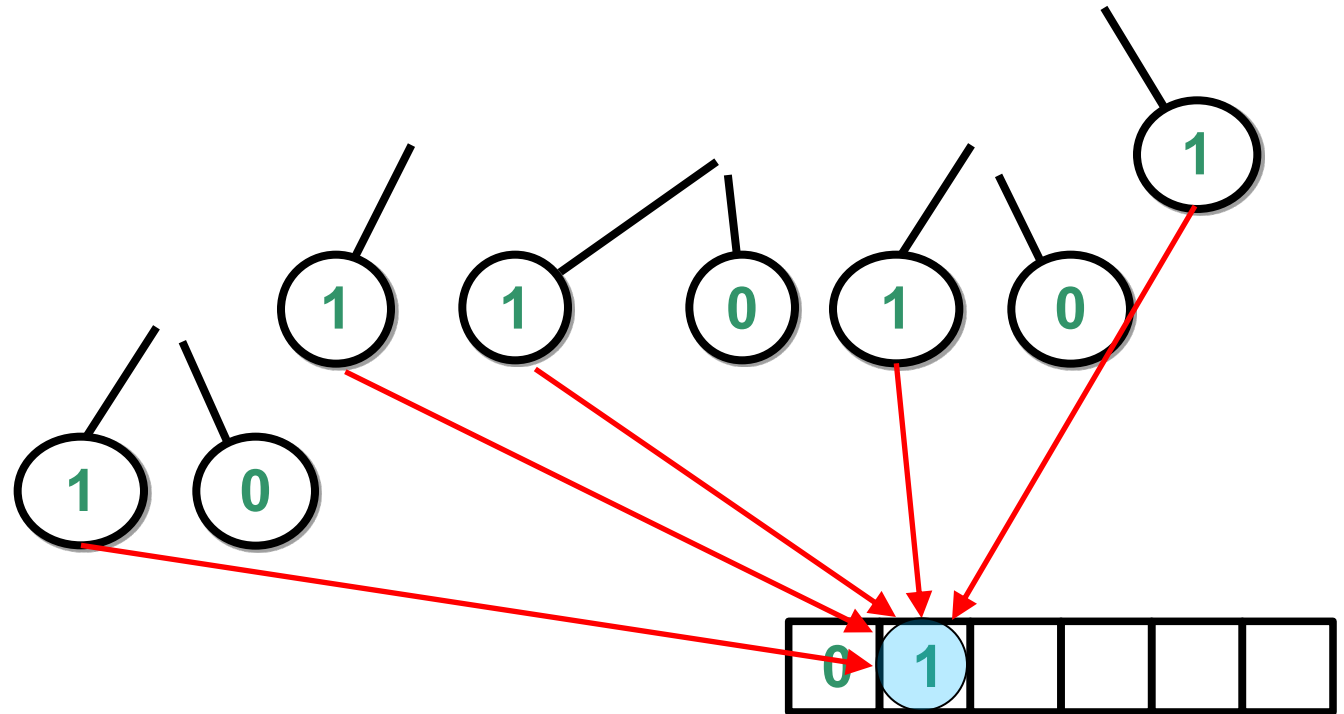
4.0. La idea...

Insistimos... date cuenta de cómo cada $F(x)$, aunque aparezca varias veces en el árbol, va a único $T(x)$...



4.0. La idea...

Insistimos... date cuenta de cómo cada $F(x)$, aunque aparezca varias veces en el árbol, va a único $T(x)$...

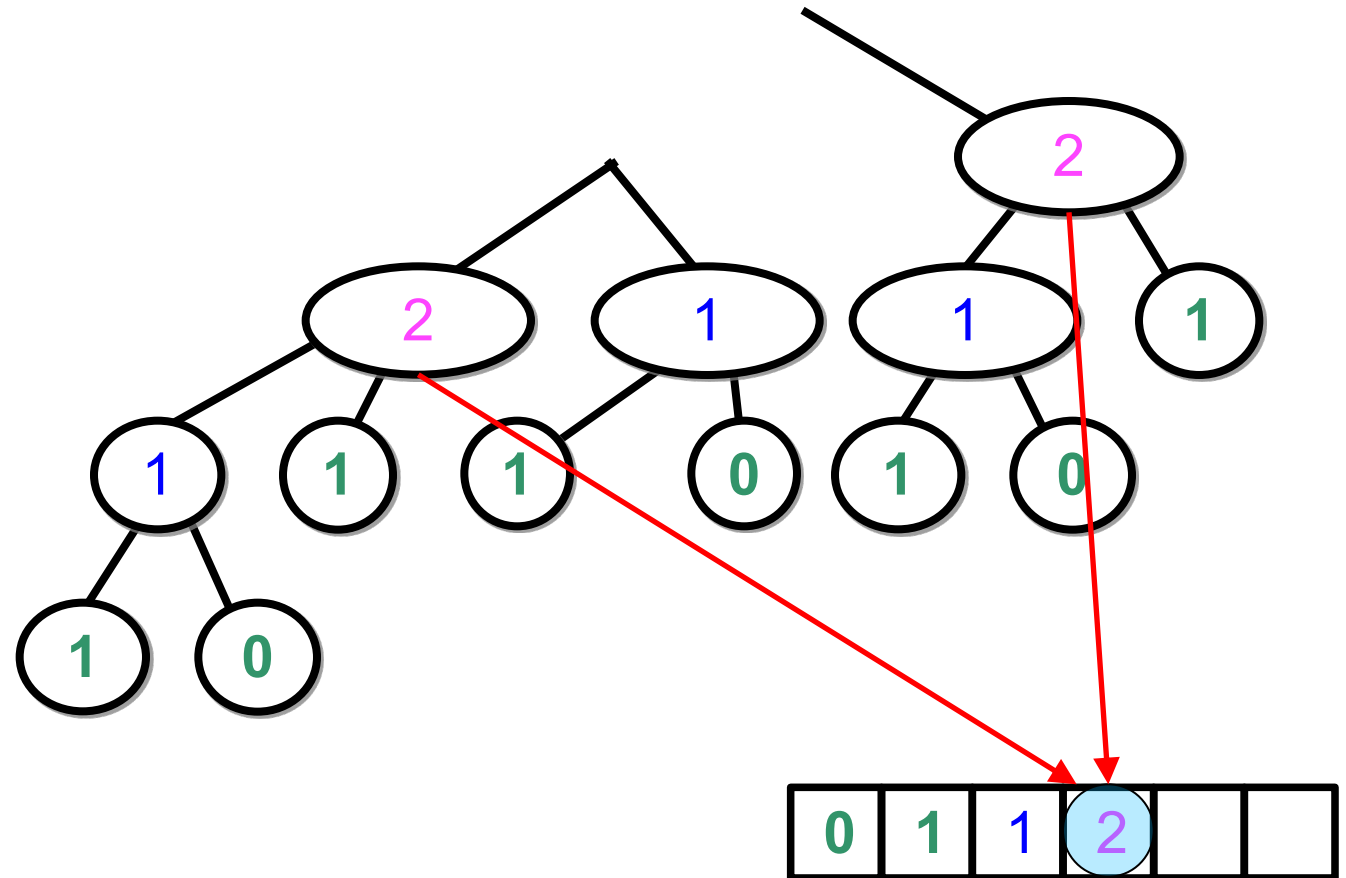


■ ■ ■



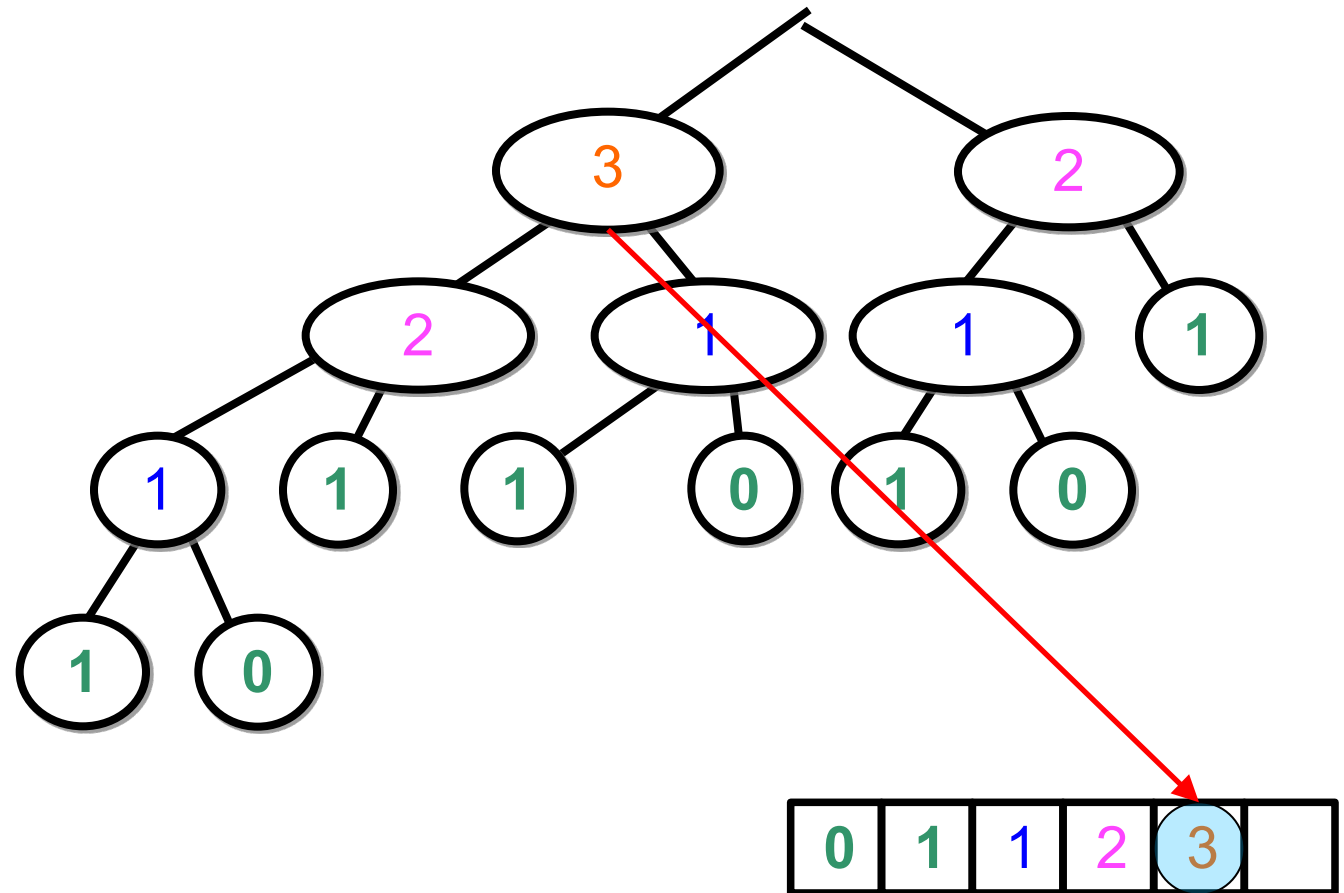
4.0. La idea...

...



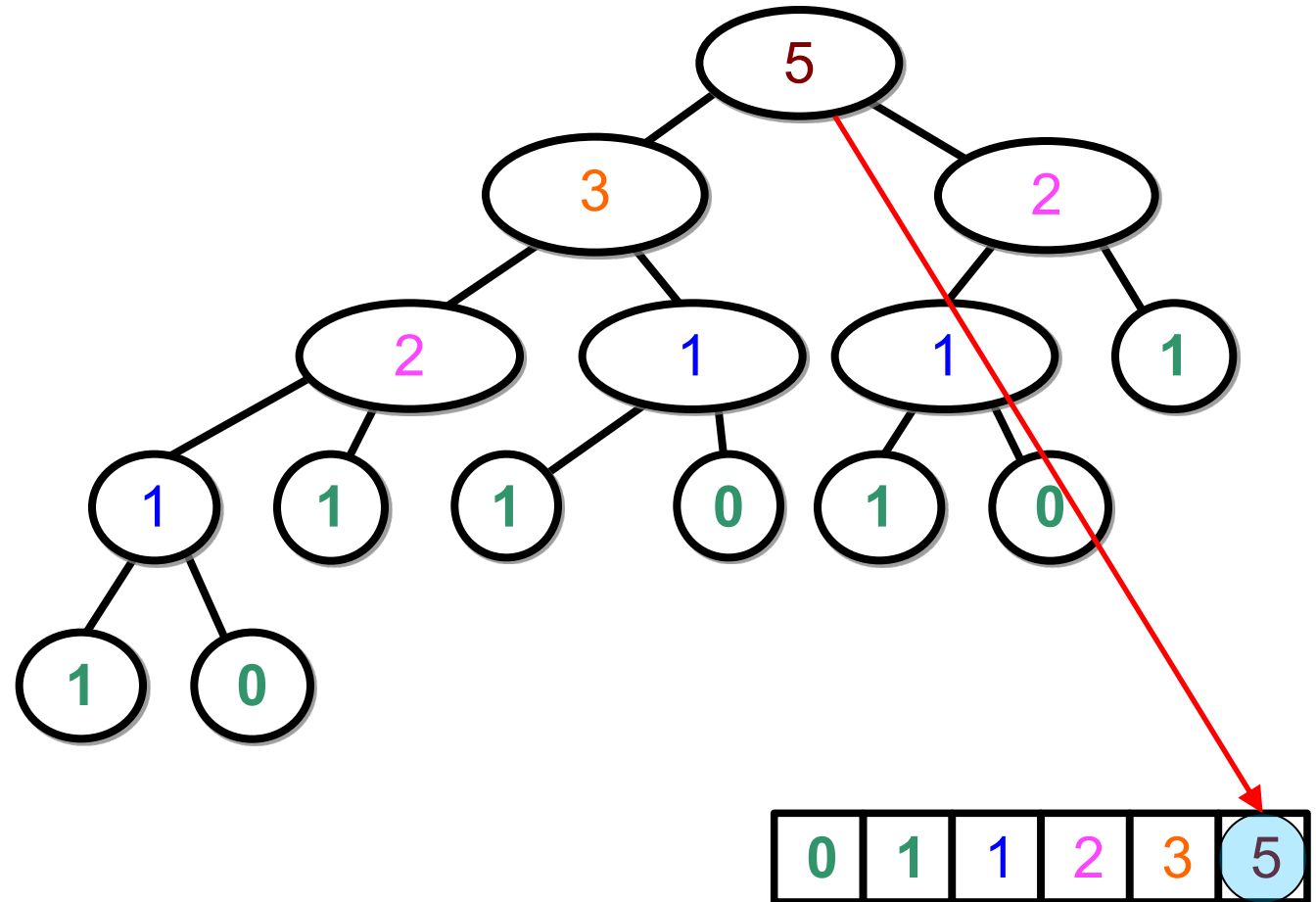
4.0. La idea...

...



4.0. La idea...

...



4.1. Método general.

- Base de **PD**: **razonamiento inductivo**: resolver problema combinando soluciones para problemas más pequeños.
- Misma idea que en **DyV** recursivo... pero otra estrategia.
- **Similitud**: descomposición recursiva del problema.
- **Diferencia**:
 - **DyV: estrategia descendente**, aplicar la fórmula recursiva.
>> Programa recursivo.
 - **PD: estrategia ascendente**, resolver primero problemas más pequeños, guardando los resultados en una tabla.
>> Programa iterativo.

4.1. Método general.

- Ejemplo: Fibonacci.

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

- DyV:

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

si $n \leq 2$ devolver 1

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

- PD:

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

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

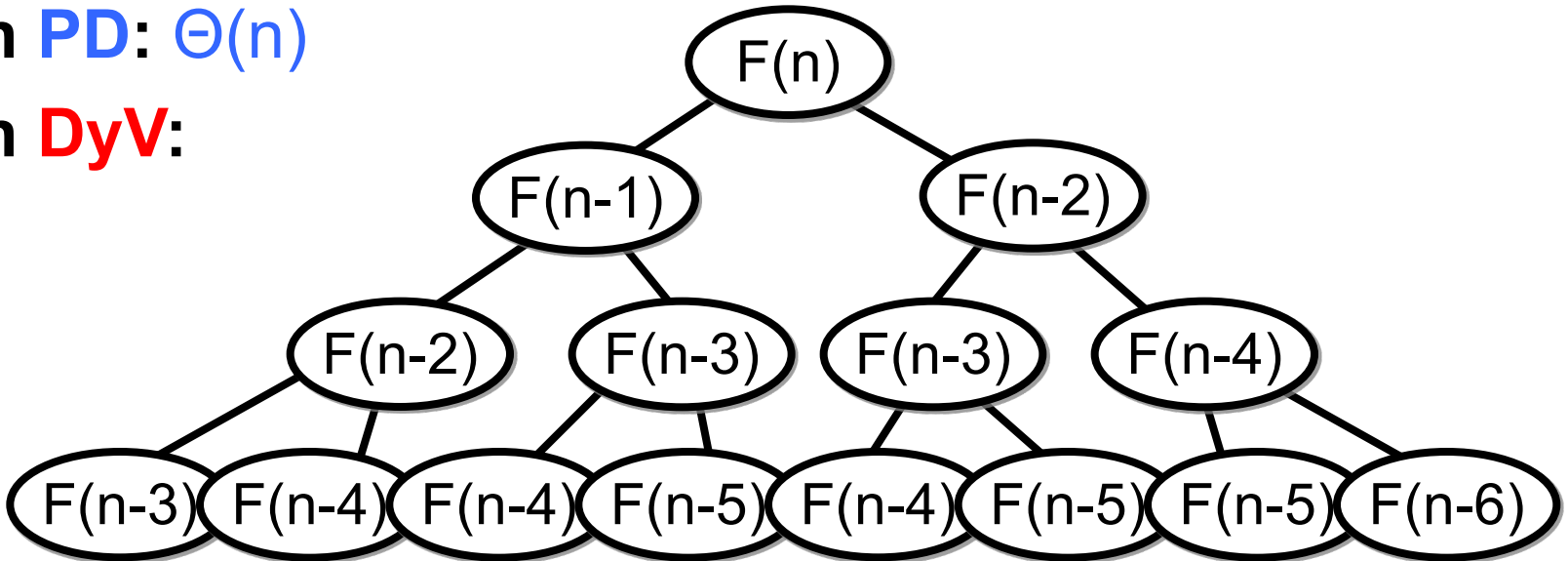
para i:= 3, ..., n hacer

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

devolver T[n]

4.1. Método general.

- Ambos usan misma fórmula recursiva, de forma distinta.
- ¿Cuál es más eficiente?
- Con **PD**: $\Theta(n)$
- Con **DyV**:



- **Problema:** Muchos cálculos están repetidos.
- El tiempo de ejecución es exponencial: $\Theta(1,62^n)$

4.1. Método general.

Métodos **ascendentes** vs. **descendentes**

- Métodos **descendentes** (**DyV**)
 - Empezar con el problema original y descomponer recursivamente en problemas de menor tamaño.
 - Problema grande >> descendemos hacia más sencillos.
- Métodos **ascendentes** (**PD**)
 - Resolver primero problemas pequeños (guardando en una tabla). Después combinar para resolver los más grandes.
 - Problemas pequeños >> ascendemos hacia más grandes.

4.1. Método general.

Pasos para aplicar PD:

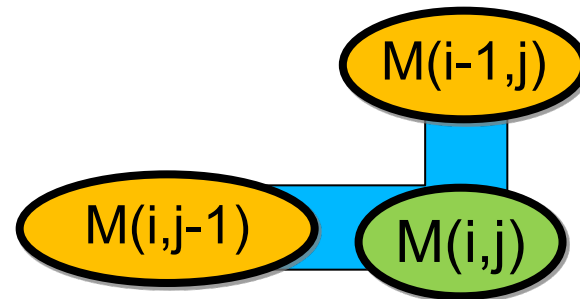
- 1) Obtener **descomposición recurrente** del problema:
 - Ecuación recurrente.
 - Casos base.
 - 2) Definir **estrategia** de aplicación de recurrencia:
 - Tablas utilizadas por el algoritmo.
 - Orden y forma de rellenarlas.
 - 3) Especificar cómo se **recompone la solución** final a partir de los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

4.1. Método general.

Pasos para aplicar PD:

1) Obtener **descomposición recurrente** del problema:

>> La “pieza de puzle”



4.1. Método general.

Pasos para aplicar PD:

2) Definir **estrategia** de aplicación de recurrencia:

- Tablas utilizadas por el algoritmo.
- Orden y forma de rellenarlas.



4.1. Método general.

Pasos para aplicar PD:

2) Definir **estrategia** de aplicación de recurrencia:

- Tablas utilizadas por el algoritmo.
- Orden y forma de rellenarlas.



4.1. Método general.

Pasos para aplicar PD:

2) Definir **estrategia** de aplicación de recurrencia:

>> Por ejemplo en este orden **no se podría**:

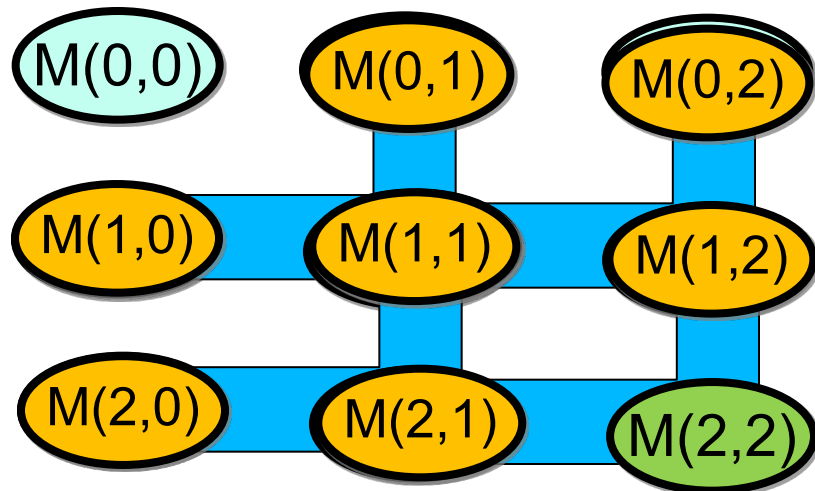


4.1. Método general.

Pasos para aplicar PD:

2) Definir **estrategia** de aplicación de recurrencia:

- Tablas utilizadas por el algoritmo.
- Orden y forma de rellenarlas.

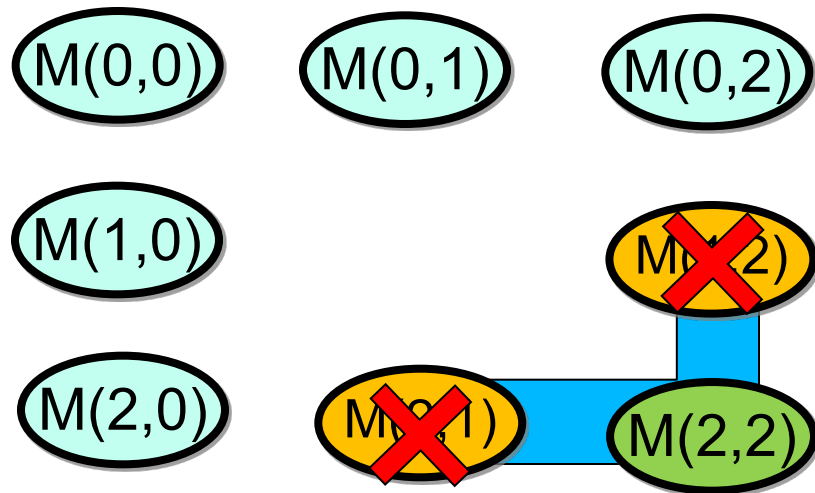


4.1. Método general.

Pasos para aplicar PD:

2) Definir **estrategia** de aplicación de recurrencia:

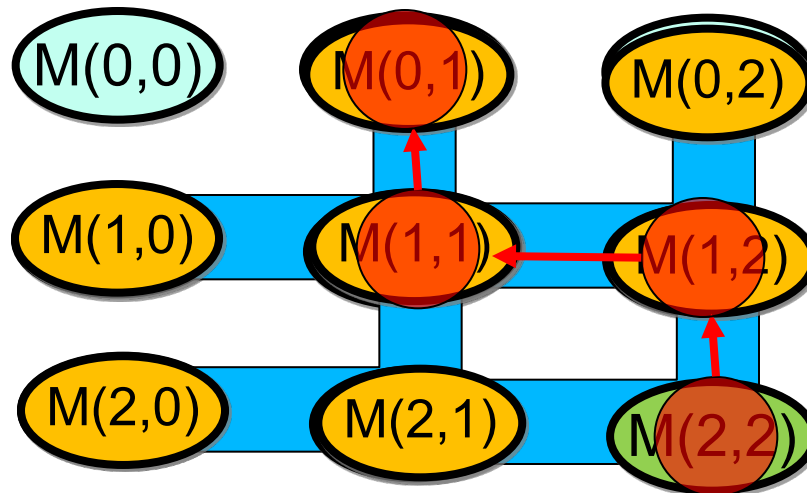
>> Por ejemplo en este orden **no se podría**:



4.1. Método general.

Pasos para aplicar PD:

- 3) Especificar cómo se **recompone** la **solución final** a partir de los valores de las tablas.



4.1. Método general.

Pasos para aplicar PD:

1) ...

2) ...

3) ...

- **Punto clave:** obtener la descomposición recurrente.
>> A veces requiere mucha “creatividad”.

>> IDEAS...

4.1. Método general.

Ideas para encontrar descomposición recurrente:

- ¿Cómo reducir problema a subproblemas más simples?
- ¿Qué parámetros determinan el **tamaño del problema**?
(y por tanto cuándo es “más simple”)
- Interpretar problema como proceso de **toma de decisiones**:
detectar cuándo tomo decisiones y qué ocurre al hacerlo
Ejemplo: **Mochila 0/1**. Decisiones: coger/no objeto
>> Tras una decisión, quedan menos objetos por decidir

Ejemplo de aplicación de estas ideas:

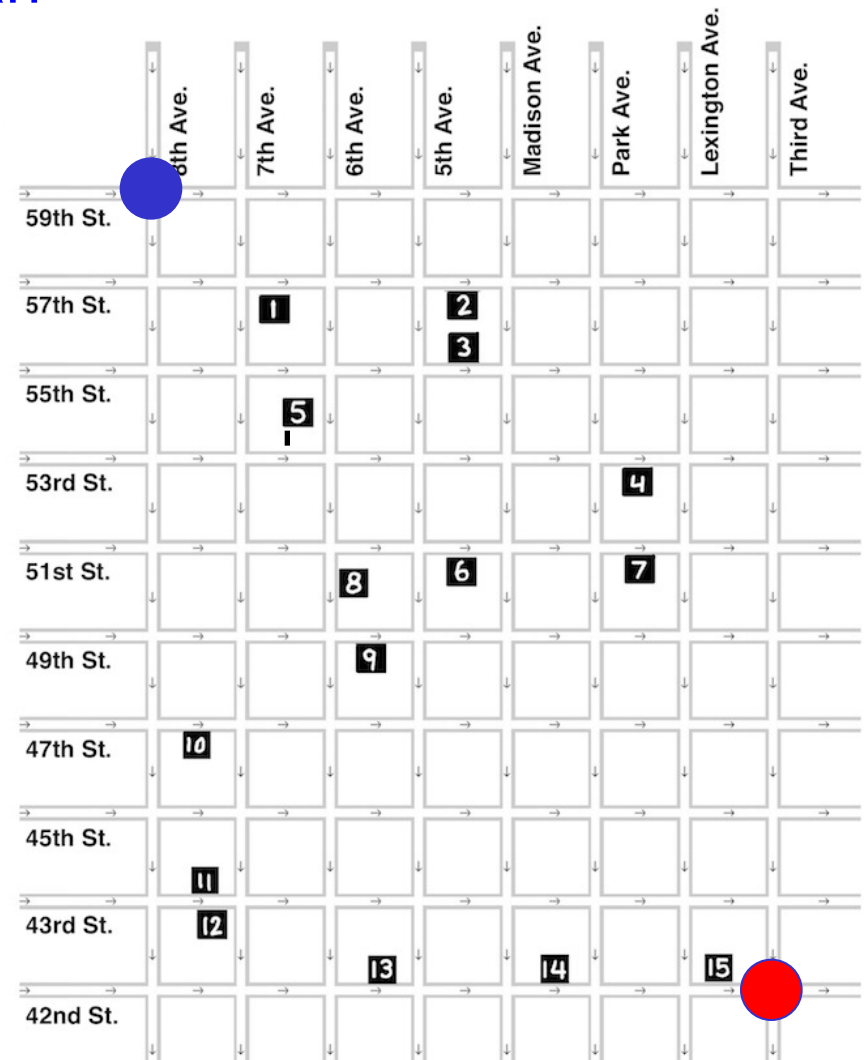
>> recorrido turístico por Manhattan

4.1. Método general.

Recorrido turístico por Manhattan

Problema:

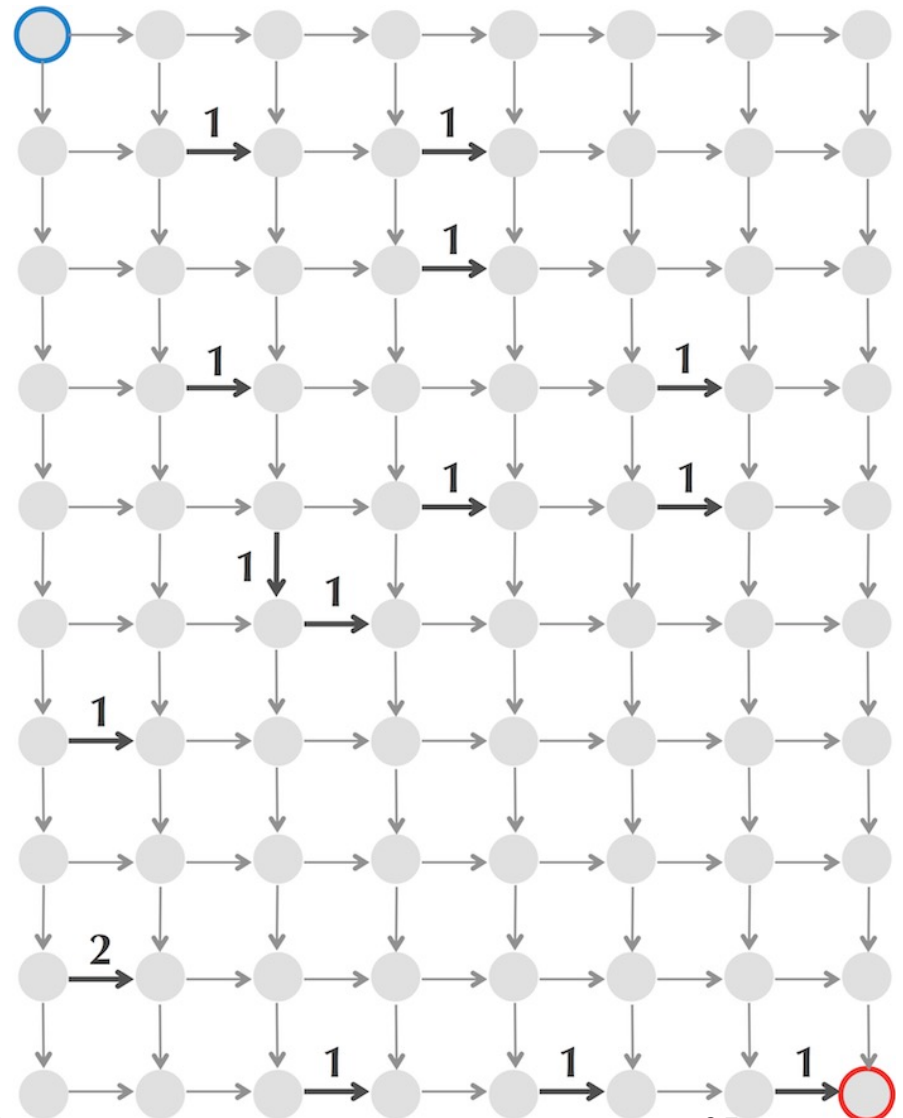
- Recorrer la isla desde
● esquina superior izquierda a
● esquina inferior derecha.
- Sólo posible moverse hacia abajo (sur) y a la derecha (este).
- Objetivo **visitar mayor número de lugares de interés.**



4.1. Método general.

(Manhattan)

- Representación del problema como un **grafo dirigido**.
- Origen (nodo azul) = $(0, 0)$
- Destino (nodo rojo) = (n, m)
- Peso aristas = nº lugares de interés en el tramo de calle.
- $s_{i,j}$ = nº máximo lugares en camino de origen a nodo (i, j)
- Calcular $s_{n,m}$ y su camino.



A.E.D.

35

4.1. Método general.

(Manhattan)

Otro ejemplo de plano...
(izquierda y arriba, coordenadas)

¿Cómo resolver por **BT**?

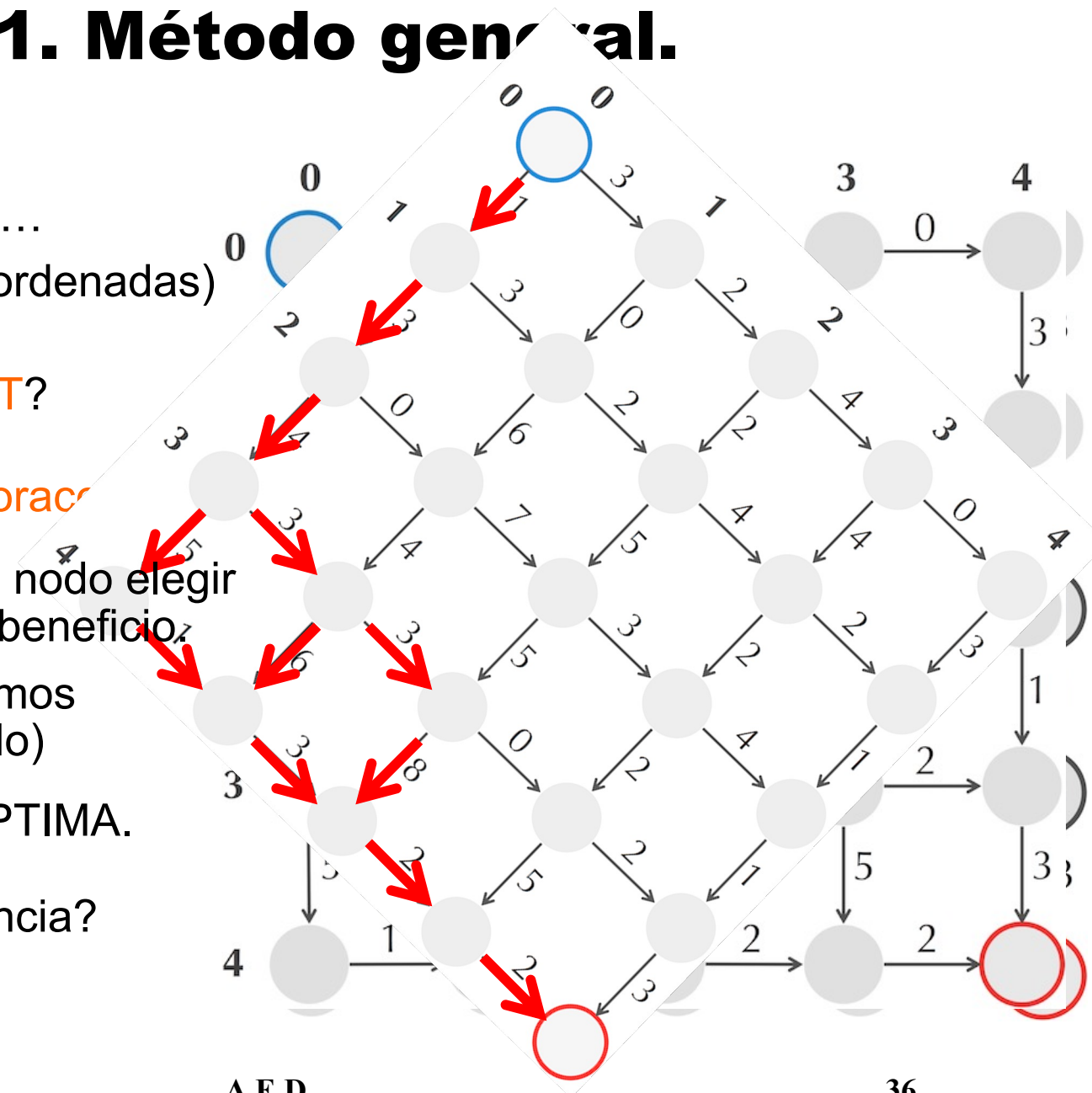
¿Cómo resolver por **vorace**?

>> Selección: en cada nodo elegir
camino con mayor beneficio.

(En cada nodo escribimos
beneficio acumulado)

>> SOLUCIÓN NO ÓPTIMA.

¿Y por **PD**? ¿Recurrencia?



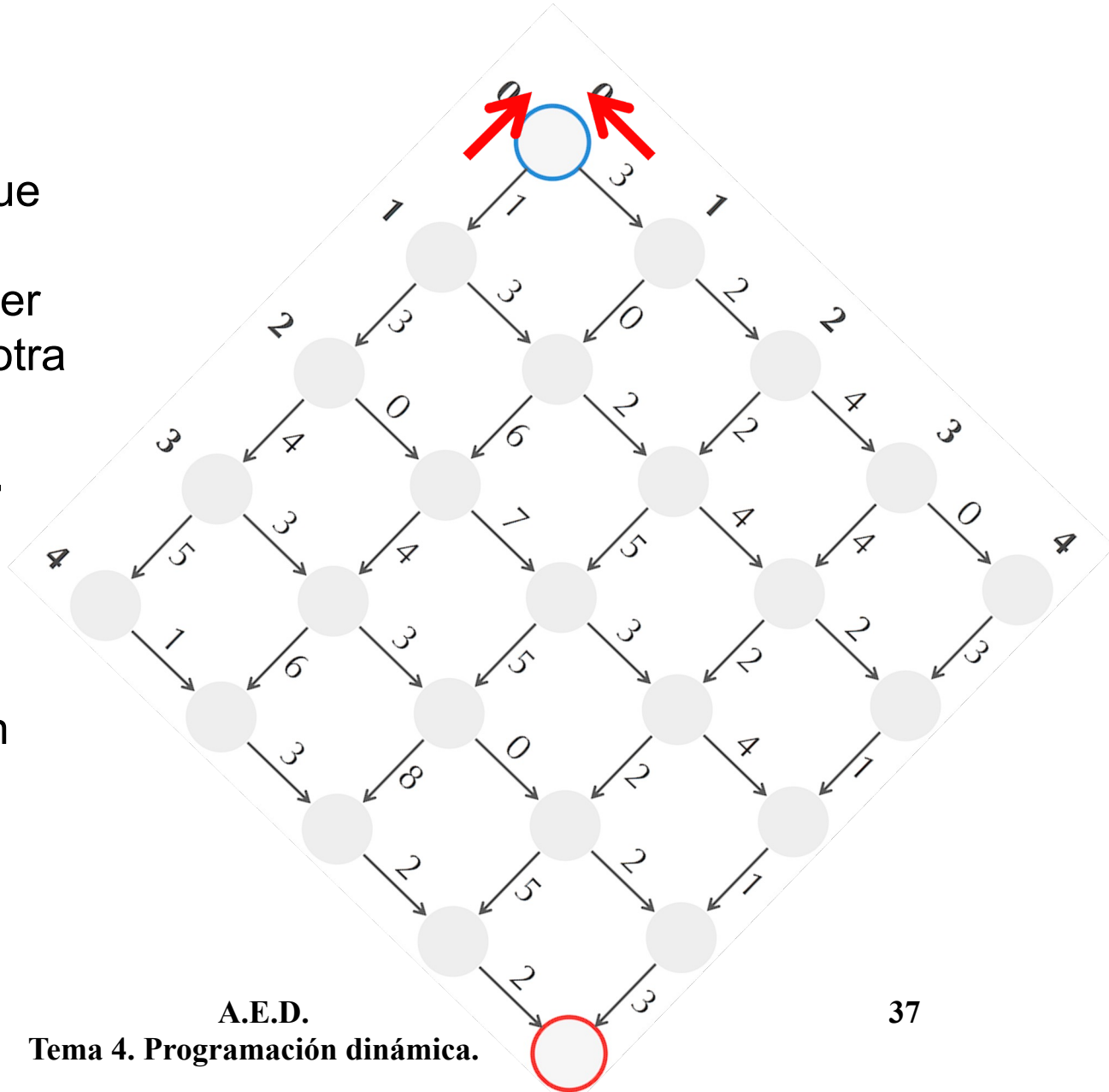
4.1. Método general.

La pregunta original es

Manhattan(n,m), así que para plantear la recurrencia vamos a ver las decisiones desde otra perspectiva...



Empezar por el final... ("salida del laberinto")

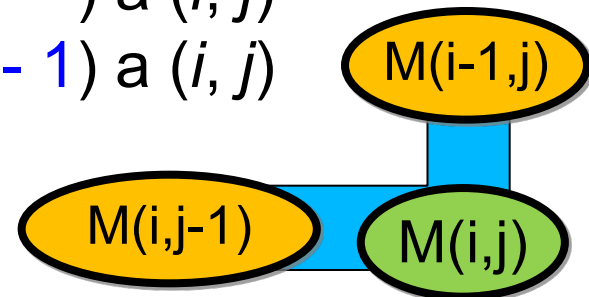
¿**desde dónde** llego al
nodo (i,j) en solución
óptima?



4.1. Método general.

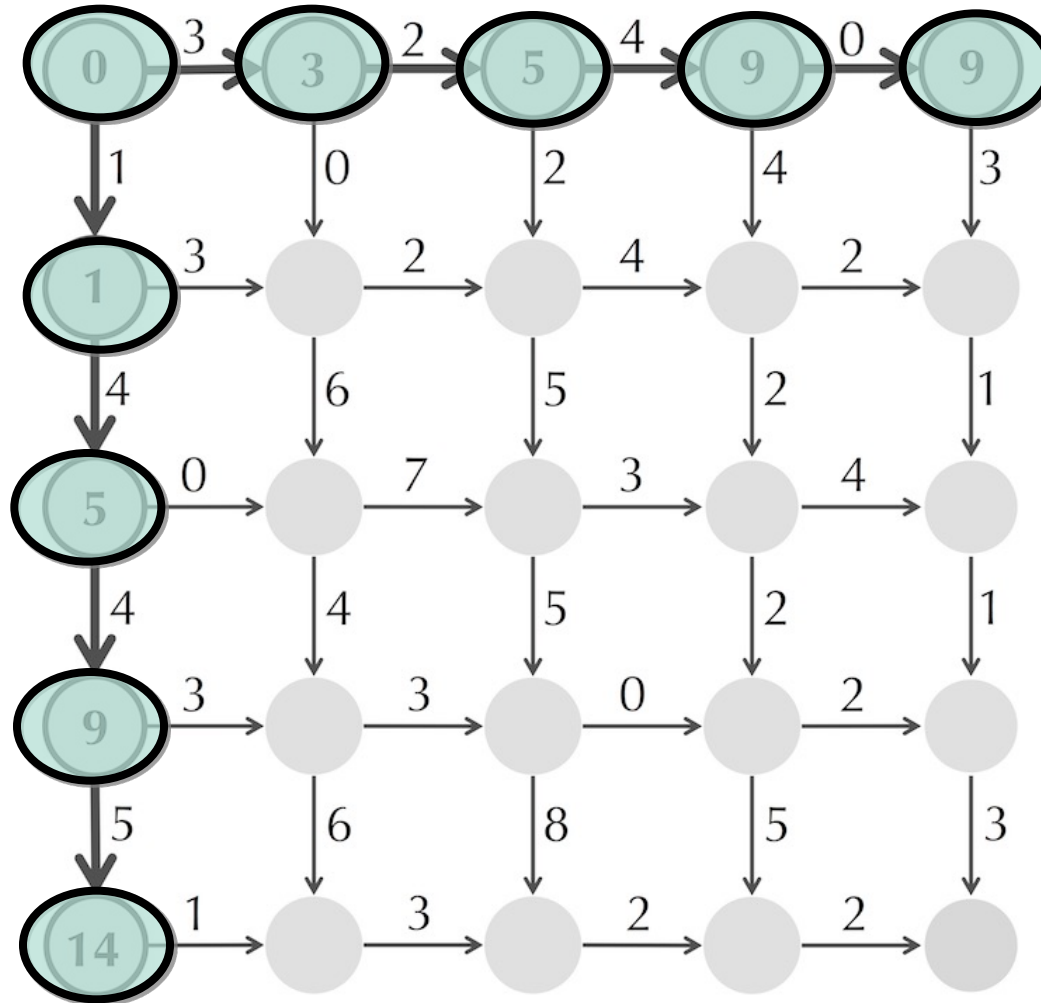
(Manhattan)

- **Decisiones**: en intersección elegir venir desde izq. o arriba.
- **Consecuencias** de una decisión:  
 - Añado cierto beneficio (n° lugares en esa calle).
 - Una decisión menos por tomar (total $n+m$ decisiones).
- **Tamaño** del problema:
 - n° decisiones pendientes = verticales + horiz. = $i + j$
 - Puede venir bien dejarlas por separado: tamaño $\approx (i, j)$
- **Recurrencia**: $s_{i,j}$ es el **máximo** entre:
 - $s_{i-1,j} + \text{peso arista vertical}$ de $(i-1, j)$ a (i, j)
 - $s_{i,j-1} + \text{peso arista horizontal}$ de $(i, j-1)$ a (i, j)
- **Casos base**: nodos con solución directa:
>> primera fila y columna...



(Manhattan)

4.1. Método general.



A.E.D.

Tema 4. Programación dinámica.

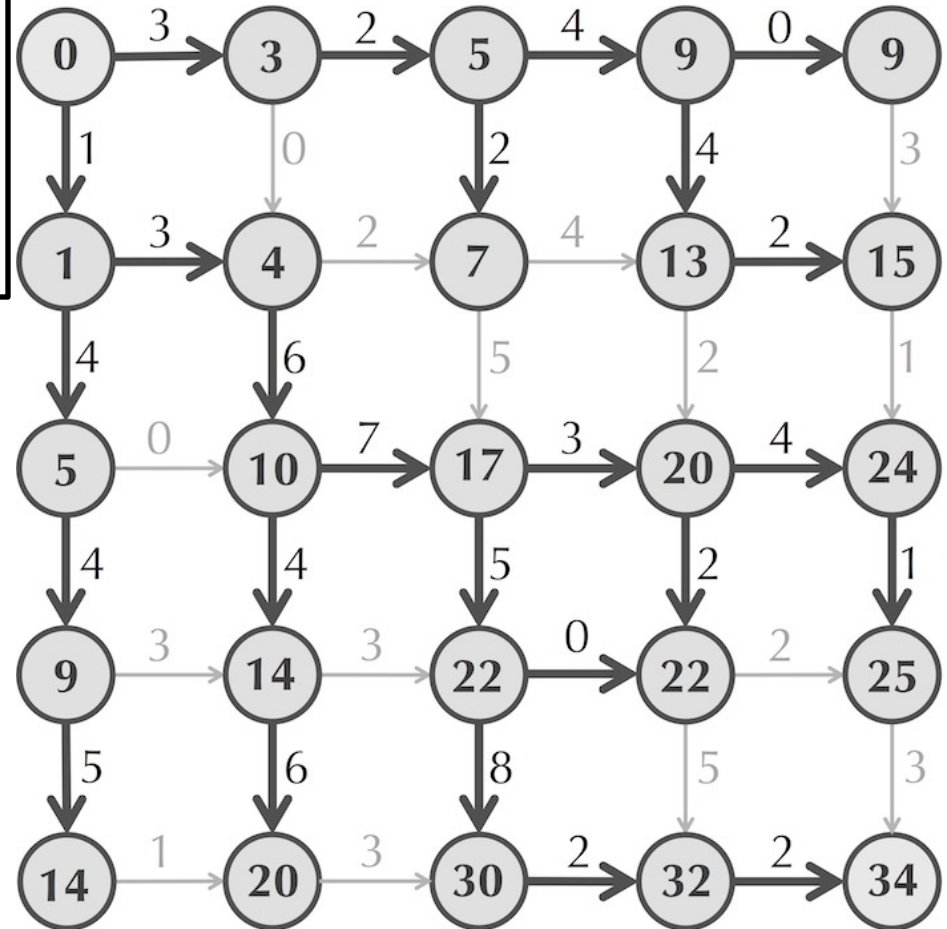
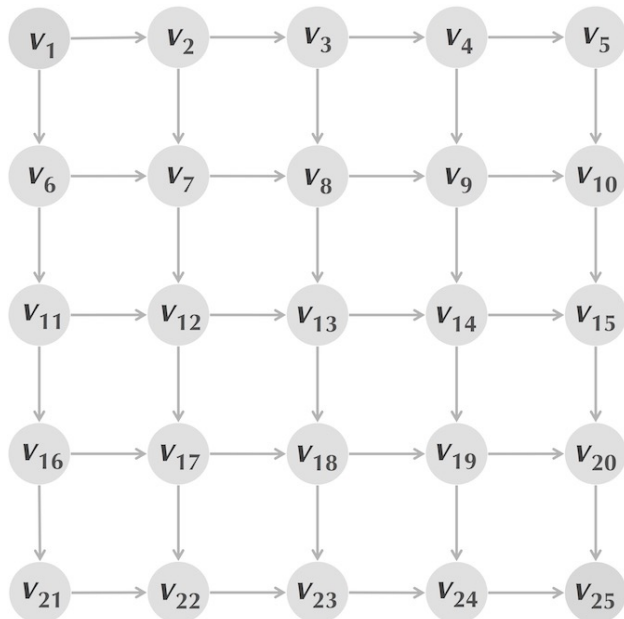
4.1. Método general.

(Manhattan)

- **Rellenar tabla:** tras casos base, por columnas, arriba a abajo

>> Usando este orden tenemos ya calculados los valores de los nodos previos, necesarios para calcular el valor del nodo actual.

- También posible por filas.

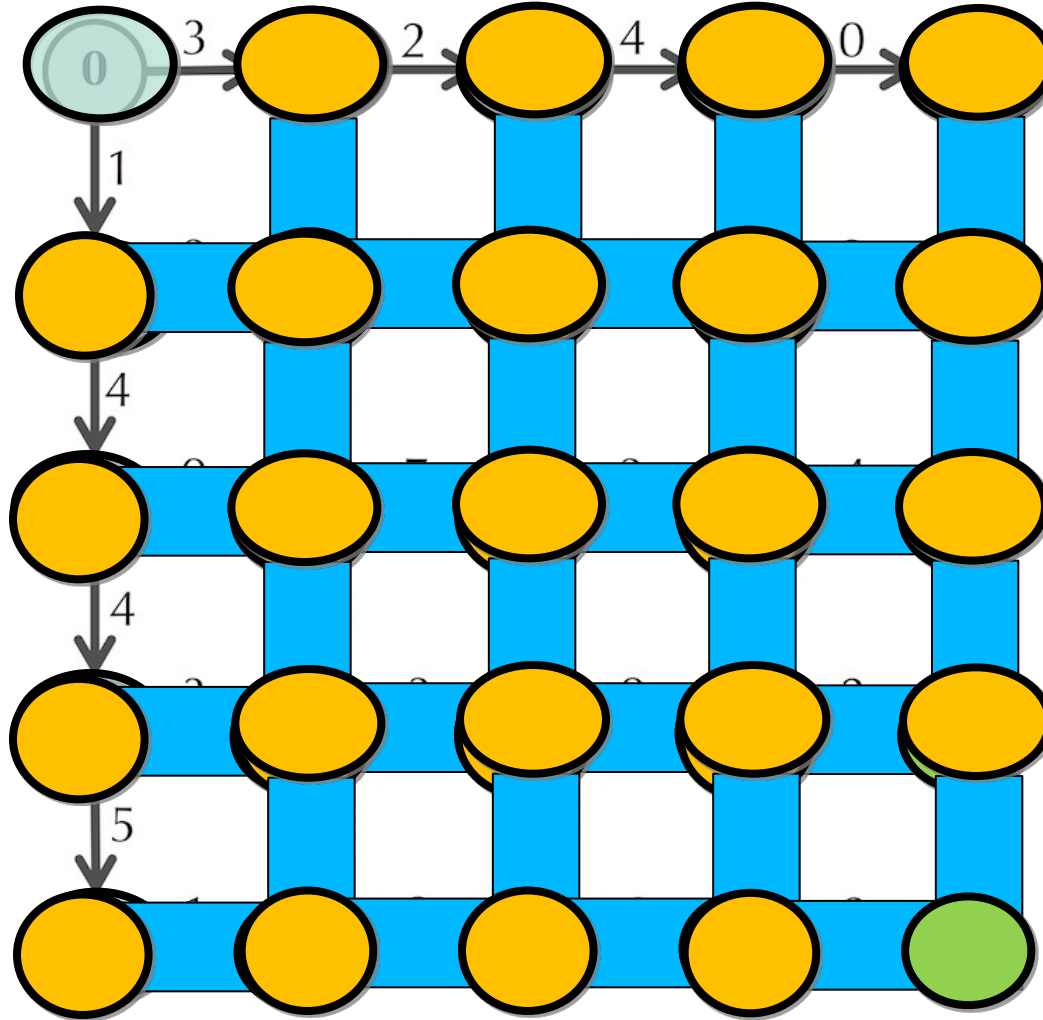


A.E.D.

ogramación dinámica.

(Manhattan)

4.1. Método general.



A.E.D.

Tema 4. Programación dinámica.

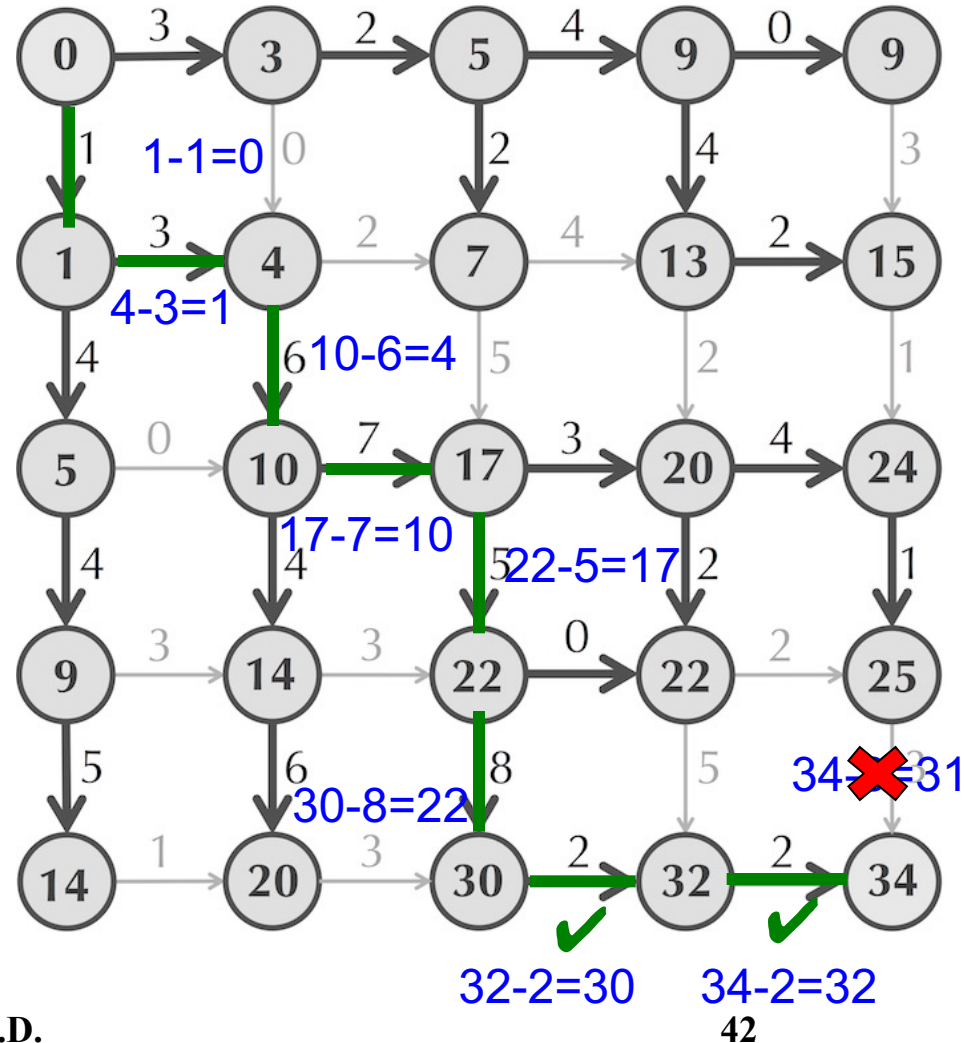
4.1. Método general.

(Manhattan)

– Reconstruir la solución:

>> Dos posibilidades:

- Reconstruir esa información a partir de la tabla de valores $s_{i,j}$.



A.E.D.

Tema 4. Programación dinámica.

4.1. Método general.

(Manhattan)

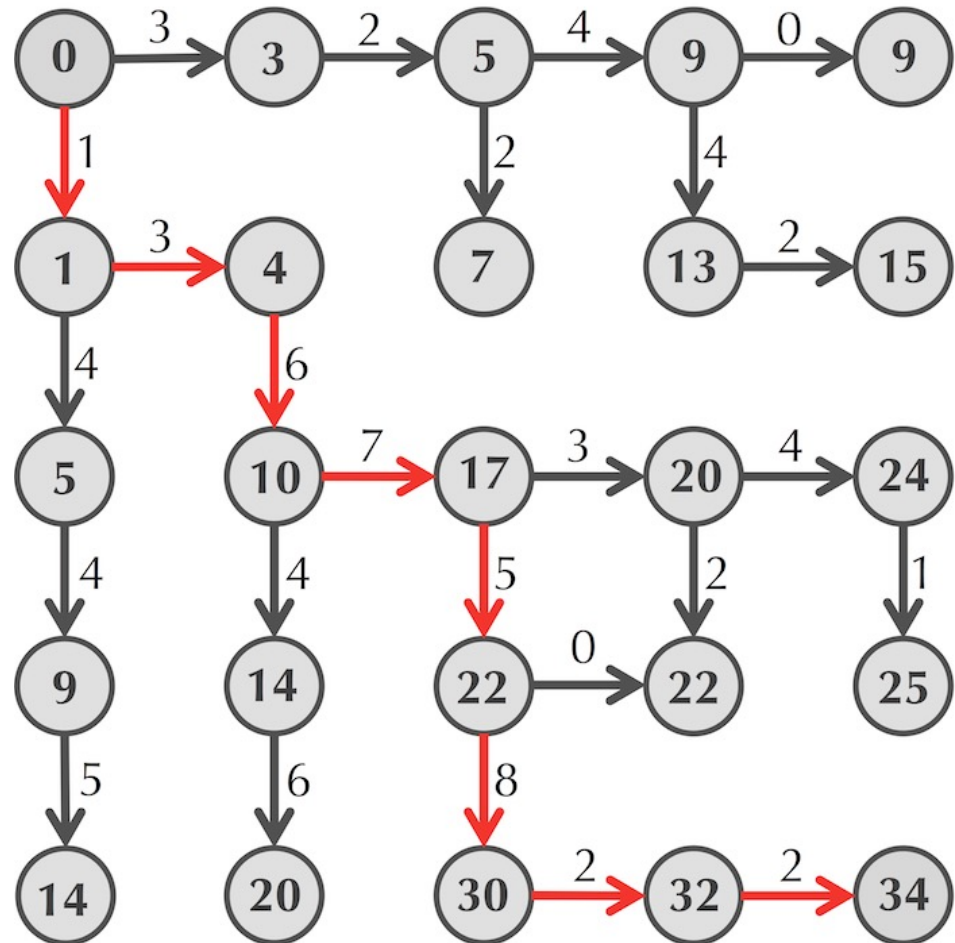
– Reconstruir la solución:

>> Dos posibilidades:

- Reconstruir esa información a partir de la tabla de valores $s_{i,j}$.
- Guardar decisiones en 2ª tabla

Generalización a DAGs:

Todo este proceso visto para Manhattan es similar para otros problemas en que también se recorra un DAG (grafo dirigido acíclico).



4.1. Método general.

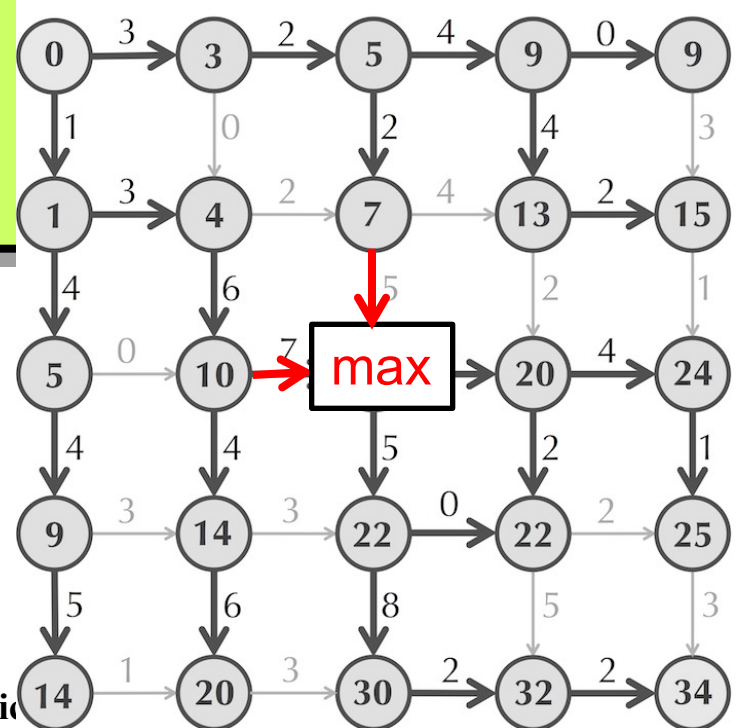
- ¿Cómo garantiza un algoritmo PD la solución correcta?
- Una descomposición recurrente es correcta si cumple el **Principio de optimalidad de Bellman**:

La solución óptima de un problema se obtiene combinando

soluciones óptimas
de subproblemas.



- ¿Se cumple para Manhattan?



A.E.D.

Tema 4. Programación dinámica

4.2. Ejemplos de aplicación.

4.2.1. Problema de la mochila 0/1.

(NP-completo clásico)

•Datos del problema:

- **n**: número de objetos disponibles.
- **M**: capacidad de la mochila.
- **p** = (**p**₁, **p**₂, ..., **p**_n) pesos de los objetos.
- **b** = (**b**₁, **b**₂, ..., **b**_n) beneficios de los objetos.

•Formulación matemática:

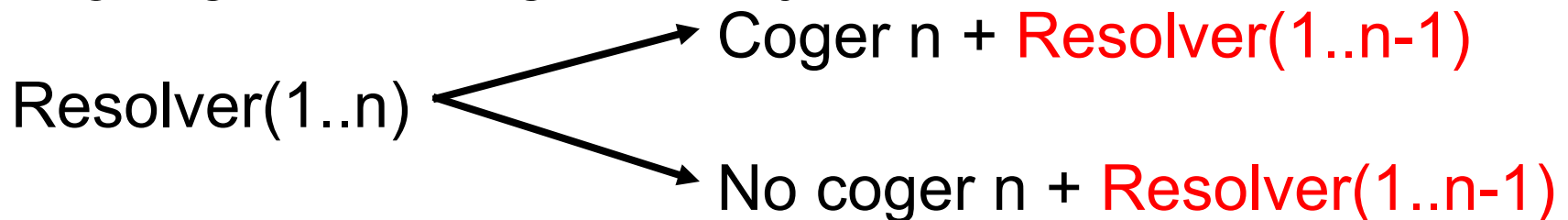
Maximizar $\sum_{i=1..n} x_i b_i$; sujeto a la restricción $\sum_{i=1..n} x_i p_i \leq M$, y $x_i \in \{0,1\}$

4.2.1. 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 sobre un objeto, nos queda un **problema de menor *tamaño*** (con un objeto menos) = **SUBPROBLEMA**.
- ¿Coger o no coger un objeto?



4.2.1. Problema de la mochila 0/1.

- ¿Coger o no coger un objeto k ?
 - **Si SÍ 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 cambia en subproblemas? (respecto original)
 - Número de objetos por decidir.
 - Peso disponible en la mochila.
- **Parametrizar** problema en función de lo que cambia...

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.

4.2.1. Problema de la mochila 0/1.

- **Definición de Mochila(k, m: entero): entero**
 - Si **NO** se coge el objeto k:
$$\text{Mochila}(k, m) = \text{Mochila}(k - 1, m)$$
 - Si **SÍ** se coge:
$$\text{Mochila}(k, m) = b_k + \text{Mochila}(k - 1, m - p_k)$$
 - **Valor óptimo:** el que dé mayor beneficio:
$$\text{Mochila}(k, m) = \max \{ \text{Mochila}(k - 1, m), \quad (\text{NO})$$

$$b_k + \text{Mochila}(k - 1, m - p_k) \} \quad (\text{SÍ})$$
- **Casos base:**
 - Si $m=0$, no se pueden incluir objetos: $\text{Mochila}(k, 0) = 0$
 - Si $k=0$, tampoco se pueden incluir: $\text{Mochila}(0, m) = 0$
 - ¿Y si m o k son negativos?

4.2.1. Problema de la mochila 0/1.

- **Casos base:**
 - Si **m** o **k** son negativos, el problema es irresoluble:
 $\text{Mochila}(k, m) = -\infty$
- **Resultado.** La siguiente **ecuación recurrente** obtiene la solución óptima del problema:

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

- ¿**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.

4.2.1. Problema de la mochila 0/1.

Paso 2) Definición de las tablas y cómo rellenarlas

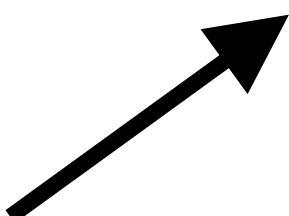
2.1) Dimensiones y tamaño de la tabla

- Definimos la tabla **V**, para guardar los resultados de los subproblemas: **$V[k, m] = \text{Mochila}(k, m)$**
- La solución del problema original es **$\text{Mochila}(n, M)$** .
- 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 $-\infty$.

4.2.1. Problema de la mochila 0/1.

2.2) Forma de rellenar las tablas:

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



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

4.2.1. Problema de la mochila 0/1.

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

j

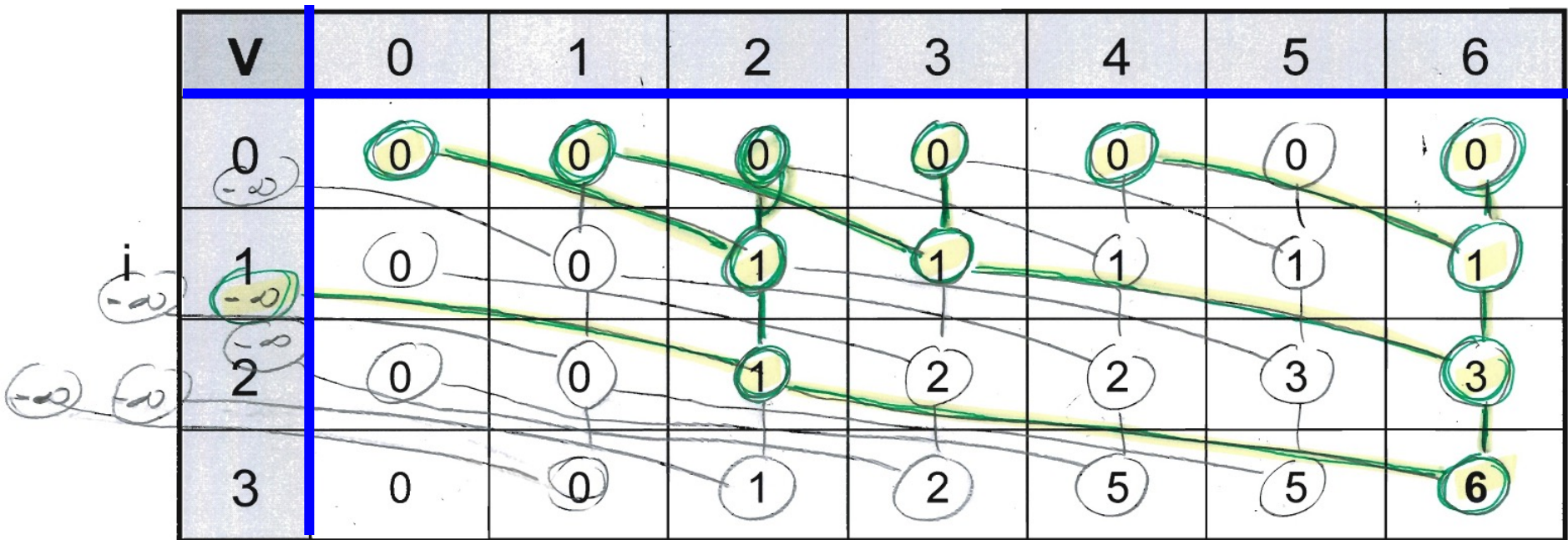
i	V	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
	1	0	0	1	1	1	1	1
	2	0	0	1	2	2	3	3
	3	0	0	1	2	5	5	6

- ¿Cuánto es el orden de complejidad del algoritmo?

4.2.1. Problema de la mochila 0/1.

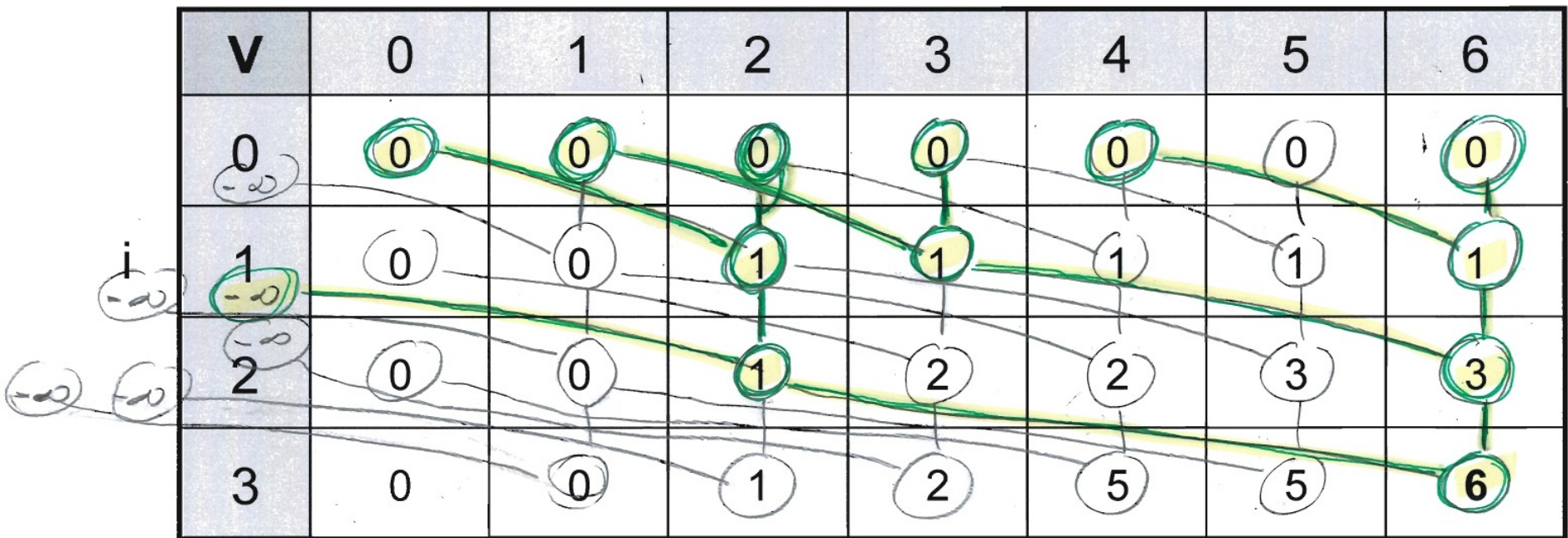
- Ejemplo. $n = 3$, $M = 6$, $p = (2, 3, 4)$, $b = (1, 2, 5)$

RESOLUCIÓN:



4.2.1. Problema de la mochila 0/1.

- Dale la vuelta...



4.2.1. Problema de la mochila 0/1.

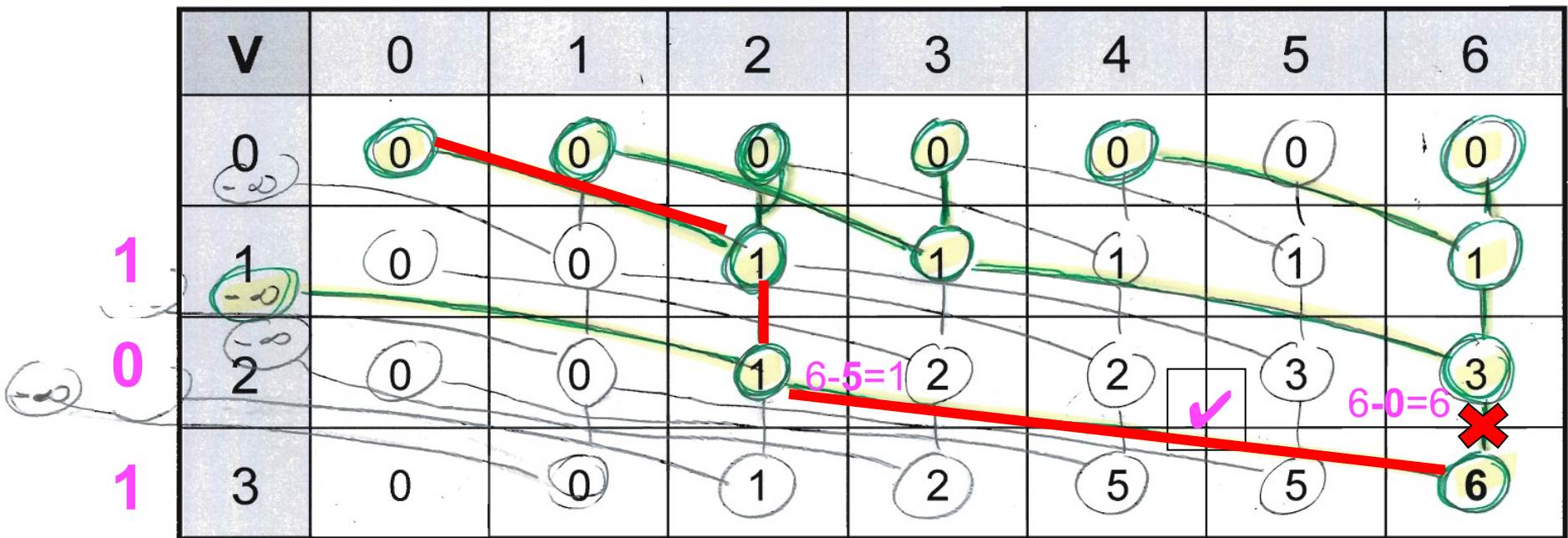
Paso 3) Recomponer la solución óptima

- $V[n, M]$ almacena el beneficio óptimo, pero ¿cuáles 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 k .
 - Si $V[k, m] = V[k-1, m]$, entonces la solución no usa el objeto $k \rightarrow x_k := 0$
 - Si $V[k, m] = V[k-1, m - p_k] + b_k$, entonces sí se usa el objeto $k \rightarrow x_k := 1$
 - Si se cumplen ambas, entonces podemos usar el objeto k o no (existe más de una solución óptima).

4.2.1. Problema de la mochila 0/1.

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

Reconstruir solución:



4.2.1. Problema de la mochila 0/1.

3) Cómo recomponer la solución óptima

$m := M$

para $k := n, \dots, 1$ **hacer**

si $V[k, m] == V[k-1, m]$ **entonces**

$x[k] := 0$

sino

// $V[k, m] == V[k-1, m-p_k] + b_k$

$x[k] := 1$

$m := m - p_k$

finsi

finpara

- Aplicar sobre el ejemplo anterior.

4.2.1. Problema de la mochila 0/1.

Código C++

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <climits>

using namespace std;

vector<vector<int>> mochila01_paso2_rellenar_tabla(const vector<int>& P, int M, int n,
    const vector<int>& B) {

    vector<vector<int>> V(n + 1, vector<int>(M + 1, 0));
    // date cuenta de que V se inicializa a ceros, por lo que los casos base ya están rellenos
    for (int k = 1; k <= n; ++k) {
        for (int m = 1; m <= M; ++m) {
            int m1 = (m - P[k - 1] >= 0) ? B[k - 1] + V[k - 1][m - P[k - 1]] : INT_MIN;
            int m2 = V[k - 1][m];
            V[k][m] = max(m1, m2);
        }
    }
    return V;
}
```

4.2.1. Problema de la mochila 0/1.

Código C++

```
pair<int, vector<int>> mochila01_paso3_reconstruir_solucion(const vector<int>& P, int M,
    int n, const vector<int>& B, const vector<vector<int>>& V) {

    int k_actual = n;
    int m_actual = M;
    vector<int> S(n, 0); // S es inicializado a ceros
    while (k_actual > 0) {
        int m1 = (m_actual - P[k_actual - 1] >= 0) ? B[k_actual - 1] + V[k_actual - 1][m_actual -
            P[k_actual - 1]] : INT_MIN;
        int m2 = V[k_actual - 1][m_actual];
        if (m1 != m2) {
            S[k_actual - 1] = 1;
            m_actual -= P[k_actual - 1];
        }
        //else... S ha sido inicializado a ceros
        k_actual--;
    }
    return {V[n][M], S};
}
```

4.2.1. Problema de la mochila 0/1.

Código C++

```
int main() {
    int n = 3;
    int M = 6;
    vector<int> P = {2, 3, 4};
    vector<int> B = {1, 2, 5};
    vector<vector<int>> V = mochila01_paso2_rellenar_tabla(P, M, n, B);
    pair<int, vector<int>> resultado = mochila01_paso3_reconstruir_solucion(P, M, n, B, V);
    cout << "Valor máximo: " << resultado.first << "\nSelección: ";
    for (int i : resultado.second) {
        cout << i << " ";
    }
    cout << endl;
    return 0;
}
```

4.2.1. Problema de la mochila 0/1.

Algunas cuestiones interesantes:

- ¿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?
- **En lugar de max se puede usar min, suma...**
 - >> Por ejemplo: calcular número de formas de elegir los objetos a meter en la mochila... **SUMA**

4.2.2. 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

4.2.2. 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.

4.2.2. 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: $\text{Cambio}(q, k) = +\infty$

4.2.2. 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.



4.2.2. Problema del cambio de monedas.

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

j

i

D	0	1	2	3	4	5	6	7	8
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?

4.2.2. 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

4.2.2. 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 $(i \neq 0) \text{ AND } (j \neq 0)$ **hacer**

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

$i := i - 1$

sino

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

$j := j - C_i$

finsi

finmientras

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

4.2.3. Problemas para profundizar.

Variaciones de la mochila:

- ¿De cuántas formas puedo llenar la mochila?
- Llenar completamente la mochila.
- ¿Y si tengo 2 mochilas? ¿O más?
- ¿Y si hay soluciones que empatan?

4.2.3. Problemas para profundizar.

Con bolsas y a lo loco:

- Concurso: gastar E euros en un día.
- n tiendas en las que hacer las compras
- En cada una puedes gastar cantidad c_i .
- Algoritmo PD que elija el conjunto de tiendas para gastar el máximo sin superar presupuesto E .

>> Parecido a la mochila, pero maximizo el gasto (como si en la mochila intento maximizar los kg usados)

4.2.3. Problemas para profundizar.

Patatas dinámicas:

- Queremos comprar G kg de patatas. G es un entero.
- Disponemos de n patatas, cada una con un peso p_i .
- Algoritmo PD que elija el conjunto de patatas que pesa al menos G gramos, minimizando exceso de peso.

>> OJO: no puedo resolverlo como hasta ahora, con lo cojo/no lo cojo (inténtalo y mira qué ocurre).

>> Idea... ¿y si la tabla lo que responde es... es posible conseguir rellenar el peso G ?

4.2.3. Problemas para profundizar.

2 montones lo más equilibrados posible:

- Tenemos n objetos
- Cada uno con un peso p_i
- Objetivo: repartir entre dos montones diferentes lo más equilibrados posible en peso, es decir, minimizar la diferencia de peso total entre ambos montones.

>> OJO: si lo planteamos como $\min(\text{lo cojo}, \text{no lo cojo})$, no cumple principio de optimalidad de Bellman.

Ejemplo: 10 objetos de un kg y 1 objeto de 10 kg

>> Se puede resolver como patatas dinámicas... tabla responde, ¿es posible conseguir x diferencia de peso?

4.3. 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.

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.