



UNIVERSITAT DE
BARCELONA



Programació Dinàmica

Algorísmica Avançada | Enginyeria Informàtica

Santi Seguí | 2025-2026

Programació Dinàmica

- La programació dinàmica, inventada pel matemàtic **Richard Bellman el 1953**, és un mètode per a reduir el temps d'execució d'un algorisme mitjançant la utilització de **subproblemes superposats i subestructures òptimes**.
- Emmagatzema els càlculs per reutilitzar-los més tard.
 - **Programming**: en el sentit d'optimització ("programació lineal").
 - **Dynamic**: perquè els resultats es van actualitzant dinàmicament.
 - El terme Programació dinàmica va ser escollit per Richard Bellman per tal que sonés bé :).
 - Un terme més descriptiu podria ser 'taula de consulta' (**look-up table**).



Sessió

- Què és la **programació Dinàmica**?
- Alguns exemples:
 - Fibonacci
 - Levenshtein
 - Problema de la Motxilla

Programació Dinàmica

- A la pregunta: Es pot resoldre qualsevol problema mitjançant la **programació dinàmica**? La resposta és **NO**.
- Per poder resoldre un problema mitjançant programació dinàmica, aquest ha de complir una sèrie de condicions que anirem veient.

Programació Dinàmica

- La programació dinàmica, inventada pel matemàtic **Richard Bellman el 1953**, és un mètode per a reduir el temps d'execució d'un algorisme mitjançant la utilització de **subproblemes superposats i subestructures òptimes**.
- El concepte de subestructura òptima vol dir que es poden fer servir solucions òptimes de subproblemes per a trobar la solució òptima del problema en el seu conjunt.
- Podem dir que l'**objectiu bàsic de la programació dinàmica** consisteix en **“descompondre” un problema d'optimització en k variables** a una **sèrie de problemes amb un nombre menor de variables** (i per tant més fàcils de resoldre).
 - Què vol dir això? Si posem com a exemple el **problema de la motxilla** el que farem és reduir el **problema inicial de n variables** (objectes diferents) a **n problemes amb $n-1$ variables**. En aquest sentit, podem dir que la programació dinàmica es basa en un mètode de descomposició.

Greedy vs. Programació Dinàmica

- **Greedy**
 - Una única seqüència de la solució és generada
- **Programació dinàmica**
 - Múltiples seqüències són generades

Divide & Conquer vs. Programació dinàmica

- **Divide and Conquer**

- Dividir el problema en subproblemes, solucionar els problemes recursivament, i finalment combinar les seves solucions per tal de solucionar el problema original

- **Programació dinàmica**

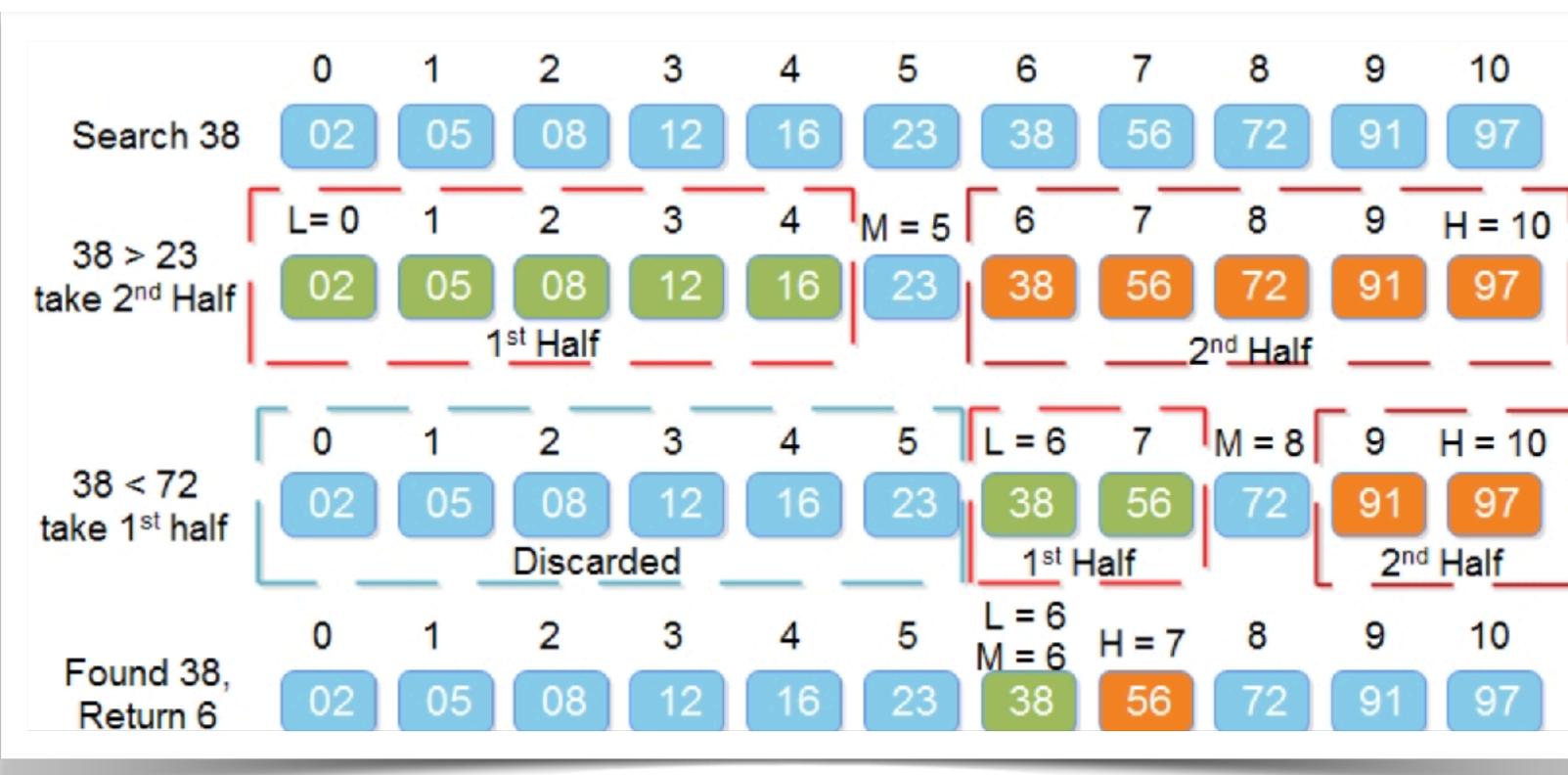
- Aplicable quan els subproblemes **no són independents**, és a dir, quan els **subproblemes comparteixen subproblemes**.
- Soluciona cada subproblema un únic cop i emmagatzema la solució dins una taula, evitant així tornar-ho a calcular si un altre cop es necessita resoldre el subproblema

Programació Dinàmica

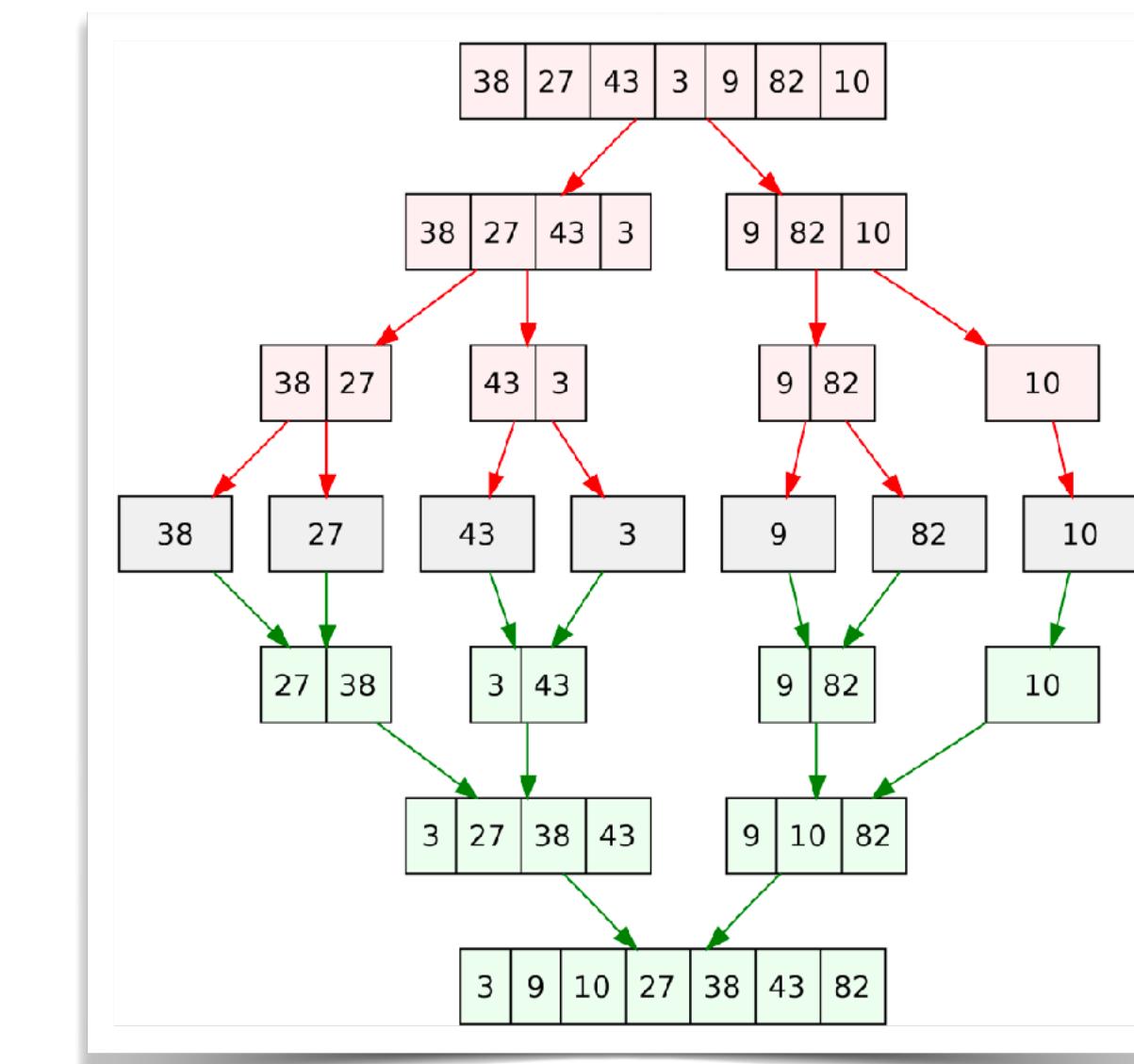
Exemples

- Quins dels següents problemes es poden resoldre amb algoritmes de **programació dinàmica**?

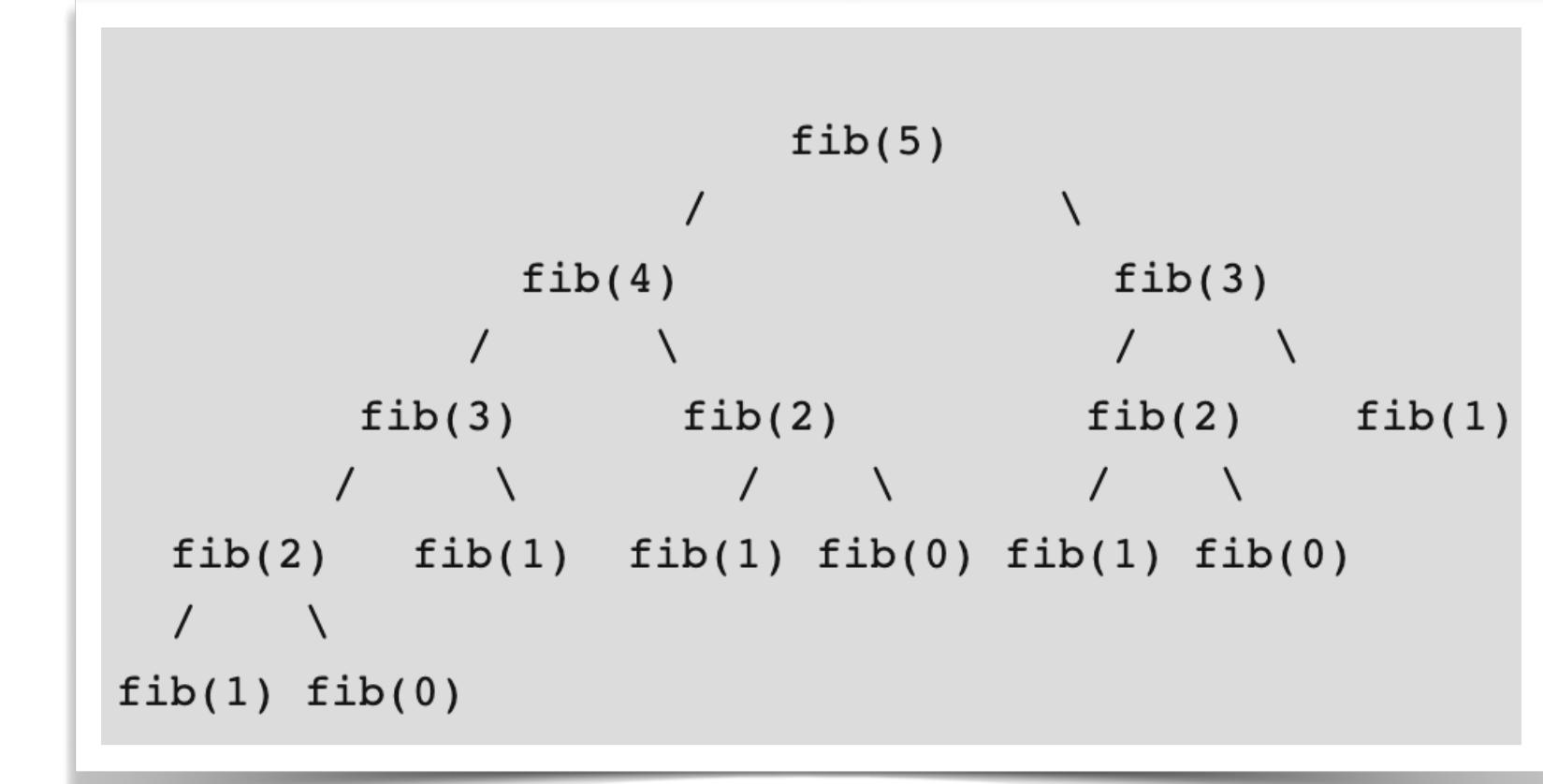
- Binary search



|| Quick Sort



|| Trobar número de Fibonacci



Programació Dinàmica

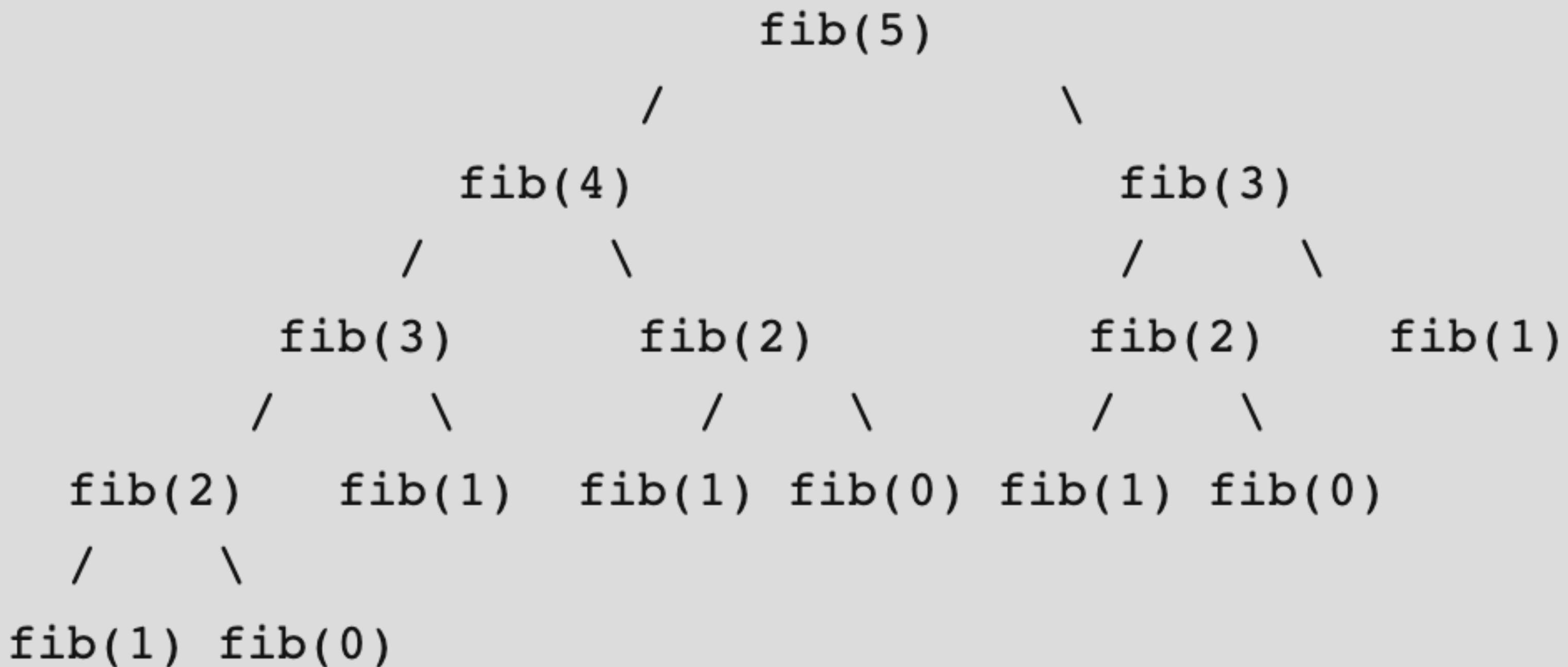
- En general, es poden resoldre problemes amb subestructures òptimes seguint aquests tres passos:
 1. *Definir funció de recursivitat*
 2. *Trobar subestructura òptima*
 3. *Afegir casos base*

Els subproblems es resolen al seu torn dividint-los en subproblems més petits fins que s'assoleixi el cas fàcil (anomenat cas base), on la solució al problema és trivial.

La programació dinàmica no hauria de ser un concepte nou per a vosaltres, de fet ja heu utilitzat aquest concepte a diversos problemes. Per trobar la solució de la successió de **Fibonacci** o bé per trobar la distància d'edició d'una paraula P a una altra paraula Q (algoritmes de **Levenshtein**).

Programació Dinàmica

Successió de Fibonacci



Programació Dinàmica

Solució naïve

```
def fibonacci(n):
    if n<0:
        print("Incorrect input")
        return 0
    # First Fibonacci number is 0
    elif n==0:
        return 0
    # Second Fibonacci number is 1
    elif n==1:
        return 1
    else:
        return fibonacci(n-1)+fibonacci(n-2)

print(fibonacci(5))
```

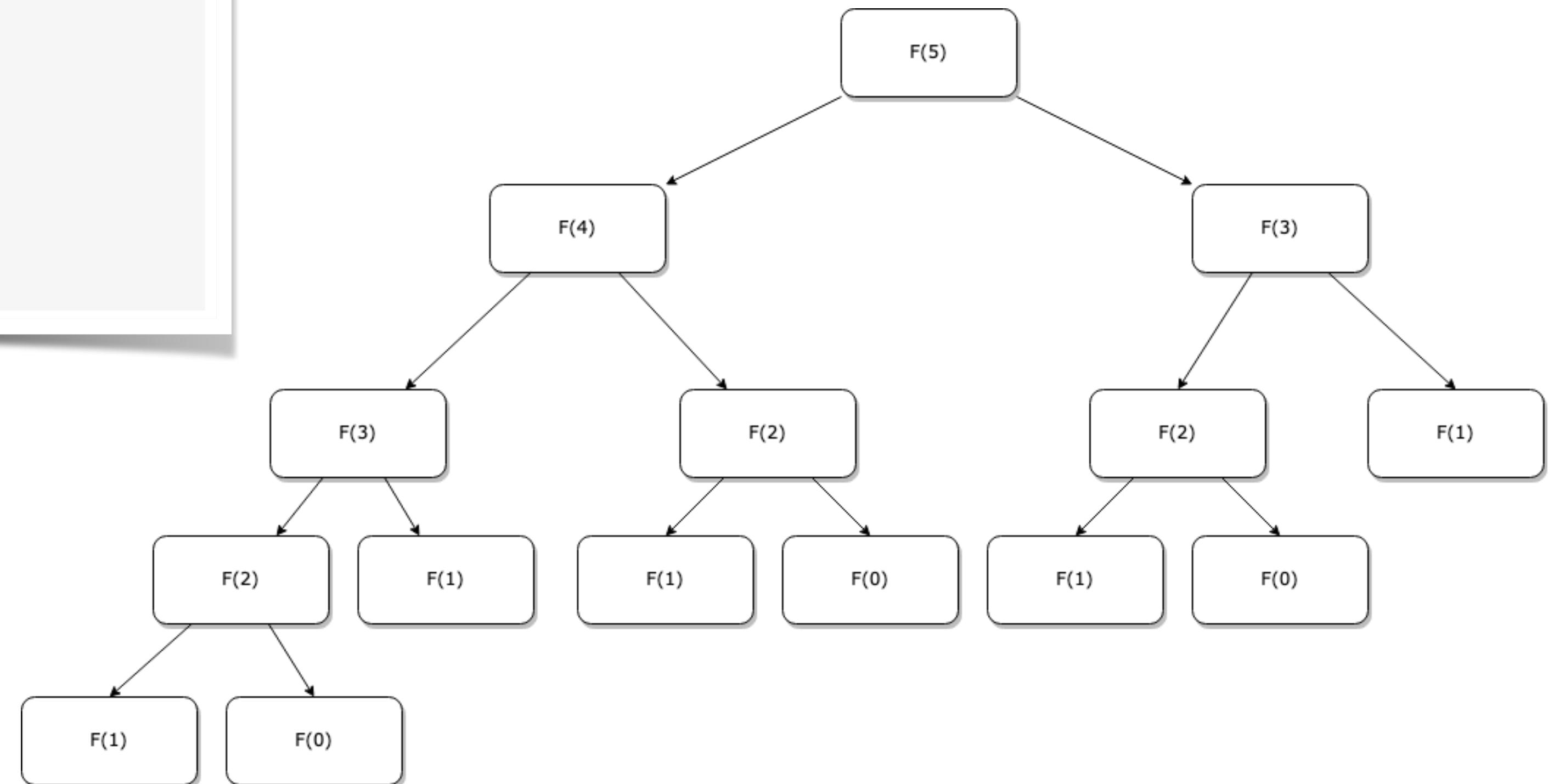
Programació Dinàmica

Successió de Fibonacci - Solució Naïve

Solució naïve

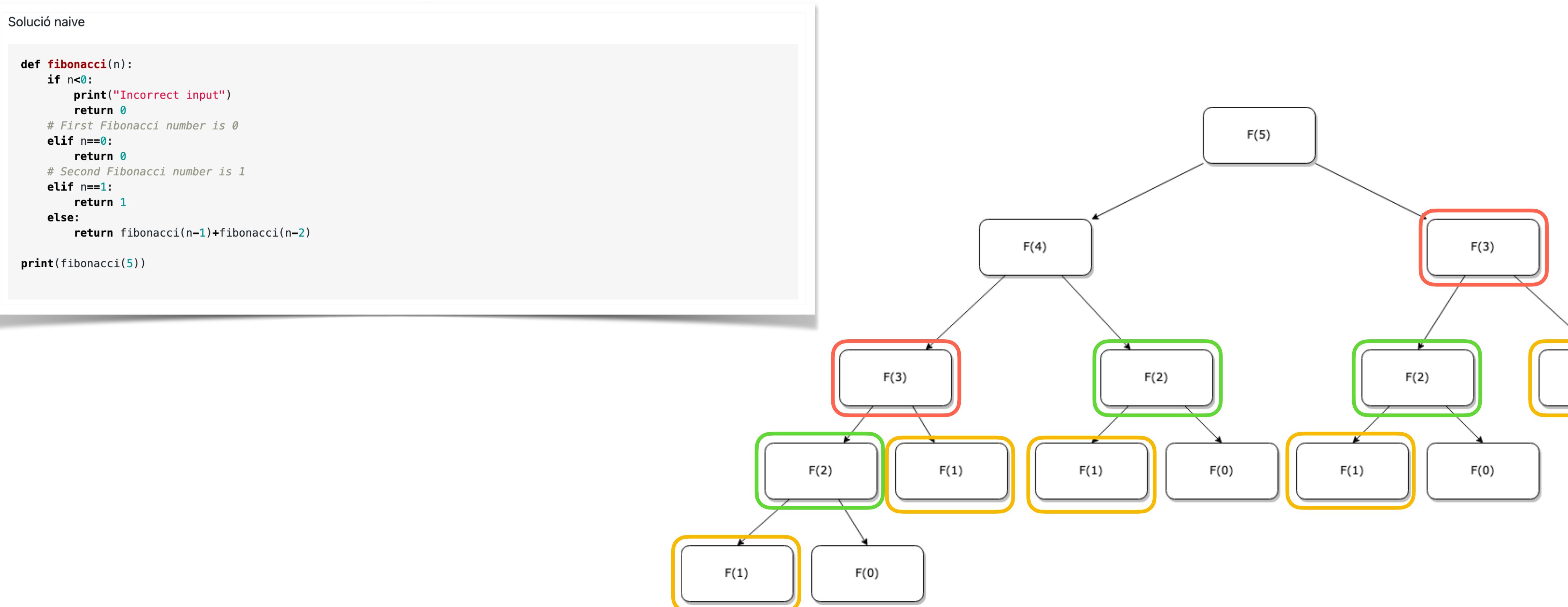
```
def fibonacci():
    if n<0:
        print("Incorrect input")
        return 0
    # First Fibonacci number is 0
    elif n==0:
        return 0
    # Second Fibonacci number is 1
    elif n==1:
        return 1
    else:
        return fibonacci(n-1)+fibonacci(n-2)

print(fibonacci(5))
```



Programació Dinàmica

Successió de Fibonacci - Solució **Naïve**



Programació Dinàmica

Successió de Fibonacci

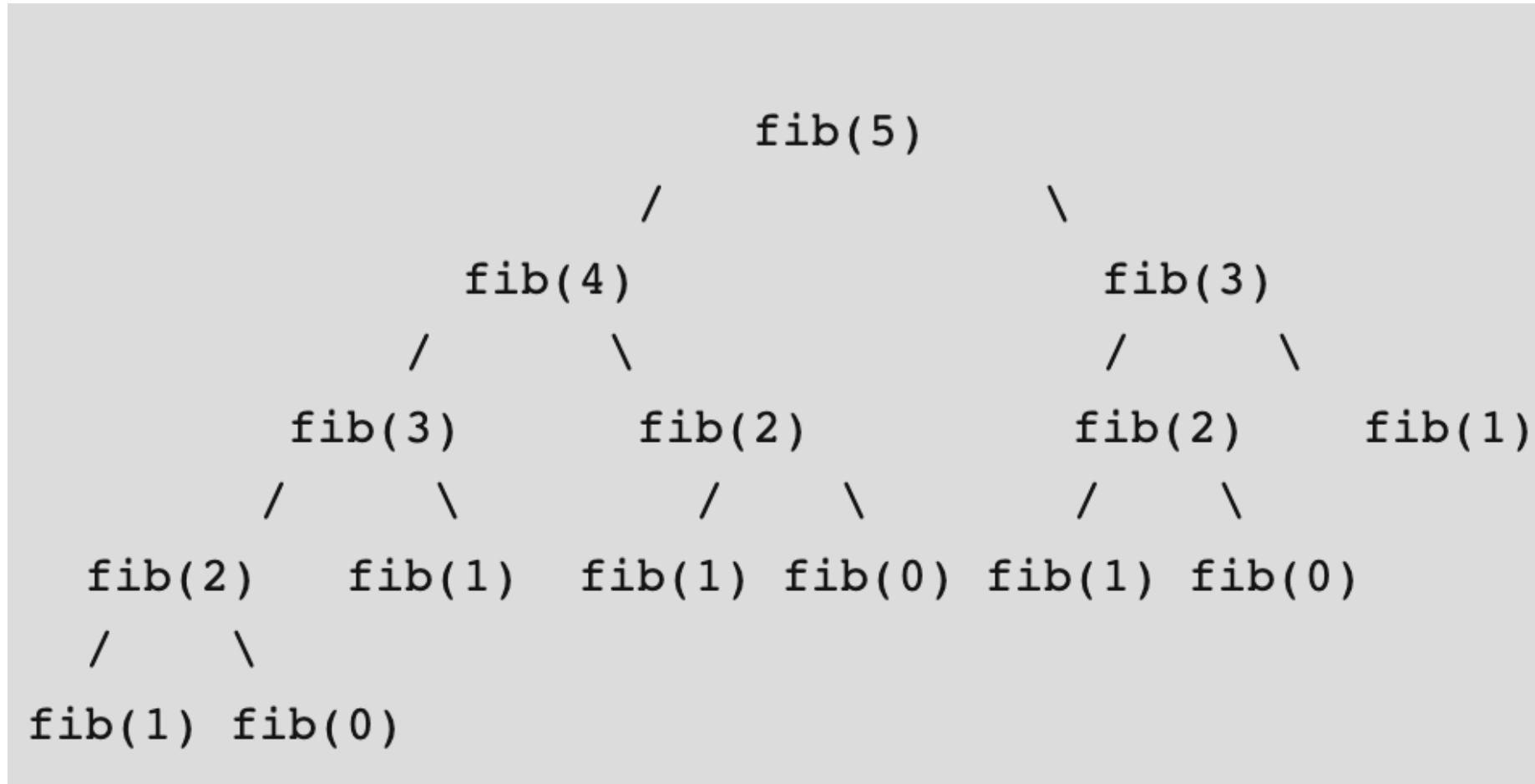
- Passos
 - Definir funció de recursivitat
 - Trobar subestructura òptima
 - Afegir casos base

Programació Dinàmica

Successió de Fibonacci

- Passos
 - Definir funció de recursivitat - $\rightarrow \text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$
 - Trobar subestructura òptima
 - Afegir casos base

$$\begin{aligned}\text{fib}(0) &= 0 \\ \text{fib}(1) &= 1\end{aligned}$$



Programació Dinàmica

Successió de Fibonacci

Solució naive

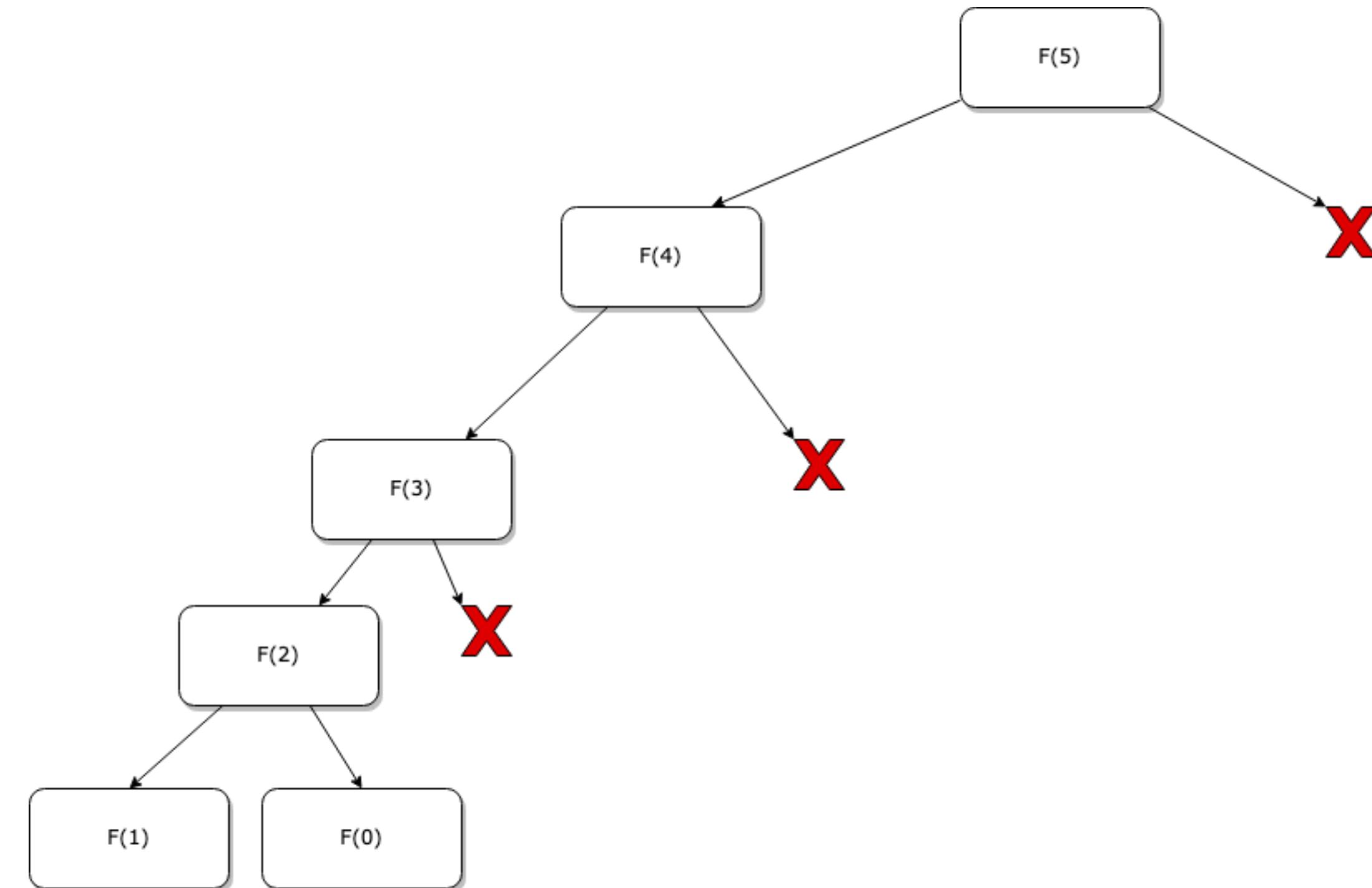
```
def fibonacci():
    if n<0:
        print("Incorrect input")
        return 0
    # First Fibonacci number is 0
    elif n==0:
        return 0
    # Second Fibonacci number is 1
    elif n==1:
        return 1
    else:
        return fibonacci(n-1)+fibonacci(n-2)

print(fibonacci(5))
```

Considerem una solució Top-Down

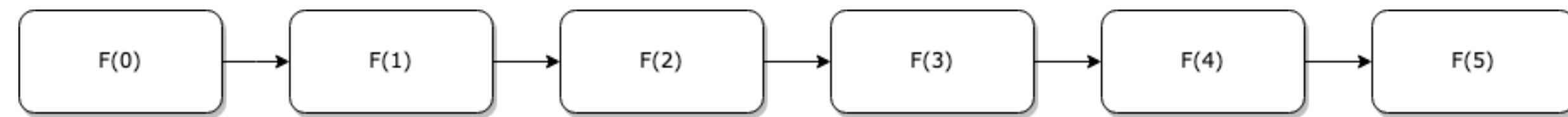
Programació Dinàmica

Successió de Fibonacci - Solució **TOP-DOWN** Programació Dinàmica



Programació Dinàmica

Successió de Fibonacci - Solució **BOTTOM-UP** Programació Dinàmica



Programació Dinàmica

Successió de Fibonacci

Solució amb programació dinàmica

```
def fibonacci(n):
    if n<0:
        print("Incorrect input")
        return 0
    # Taking 1st two fibonacci number as 0 and 1
    f = [0, 1]

    # compute Fibonacci from 2 to n
    for i in range(2, n+1):
        f.append(f[i-1] + f[i-2])
    return f[n]

print(fibonacci(5))
```

Solució naive

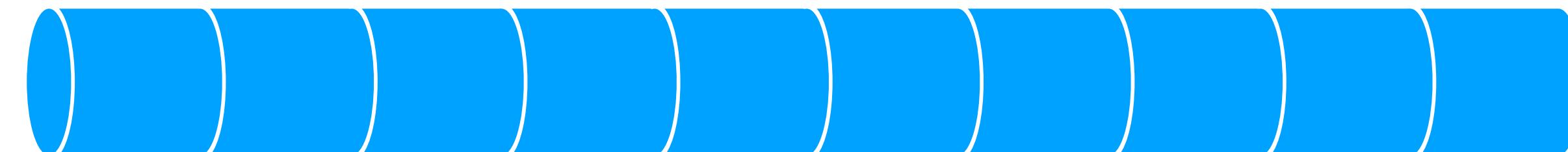
```
def fibonacci(n):
    if n<0:
        print("Incorrect input")
        return 0
    # First Fibonacci number is 0
    elif n==0:
```

Solució Bottom-up

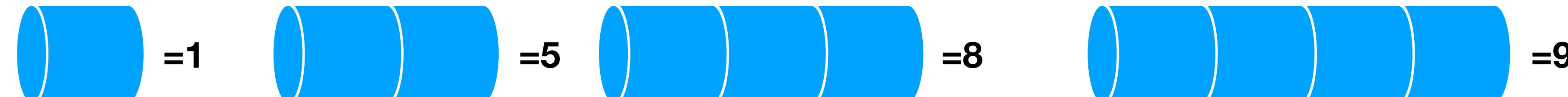
Programació Dinàmica

Rod-cutting problem

- Donada una **vareta de n** centímetres de longitud i una sèrie de preus que inclou els preus de totes les possibles peces de mida inferior a n .



- Volem **determinar el valor màxim** que es pot obtenir tallant la vareta i venent les peces resultants. Per exemple, si la longitud de la vareta és 10 i els valors de les diferents peces es donen de la següent manera, el valor màxim que es pot obtenir és 25 (tallant-la en 5 peces de longitud 2)



Programació Dinàmica

Rod-cutting problem

- Pensem una solució mitjançant un **algoritme exhaustiu**
 - **Alguna proposta?**
 - **Ens assegura la solució òptima?**
 - **Quina és la complexitat?**

Programació Dinàmica

Rod-cutting problem

- Pensem una solució mitjançant un **algoritme greedy**
 - **Alguna proposta?**
 - **Ens assegura la solució òptima?** justifica amb un contraexemple.
 - **Quina és la complexitat?**

Programació Dinàmica

Rod-cutting problem

- Pensem una solució mitjançant un **algoritme greedy**

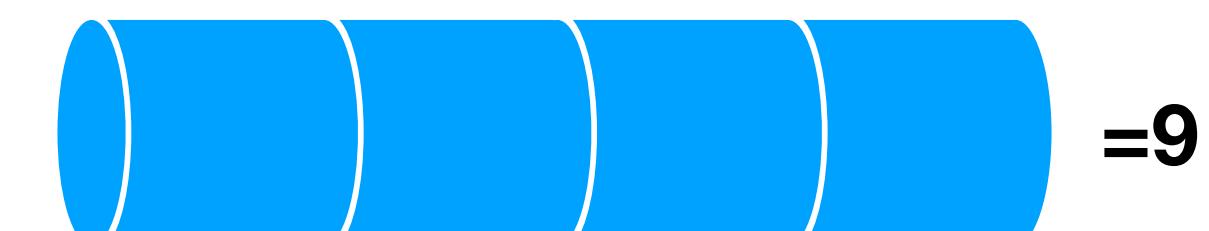
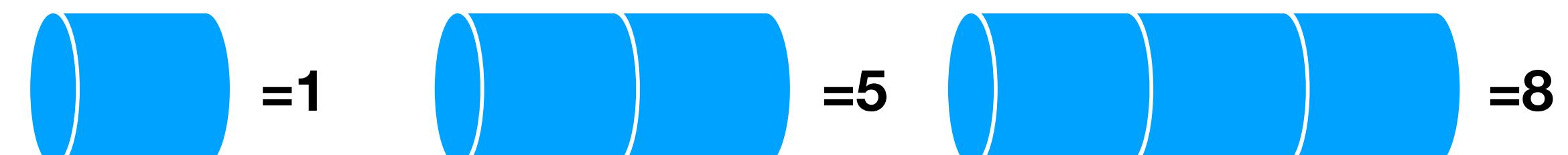
- **Alguna proposta?**

- **Ens assegura la solució òptima?** justifica amb un contraexemple.

- **Quina és la complexitat?**

Si la longitud de la vareta es 4 ->
segment 3 -> segment 1
Solució 8+1

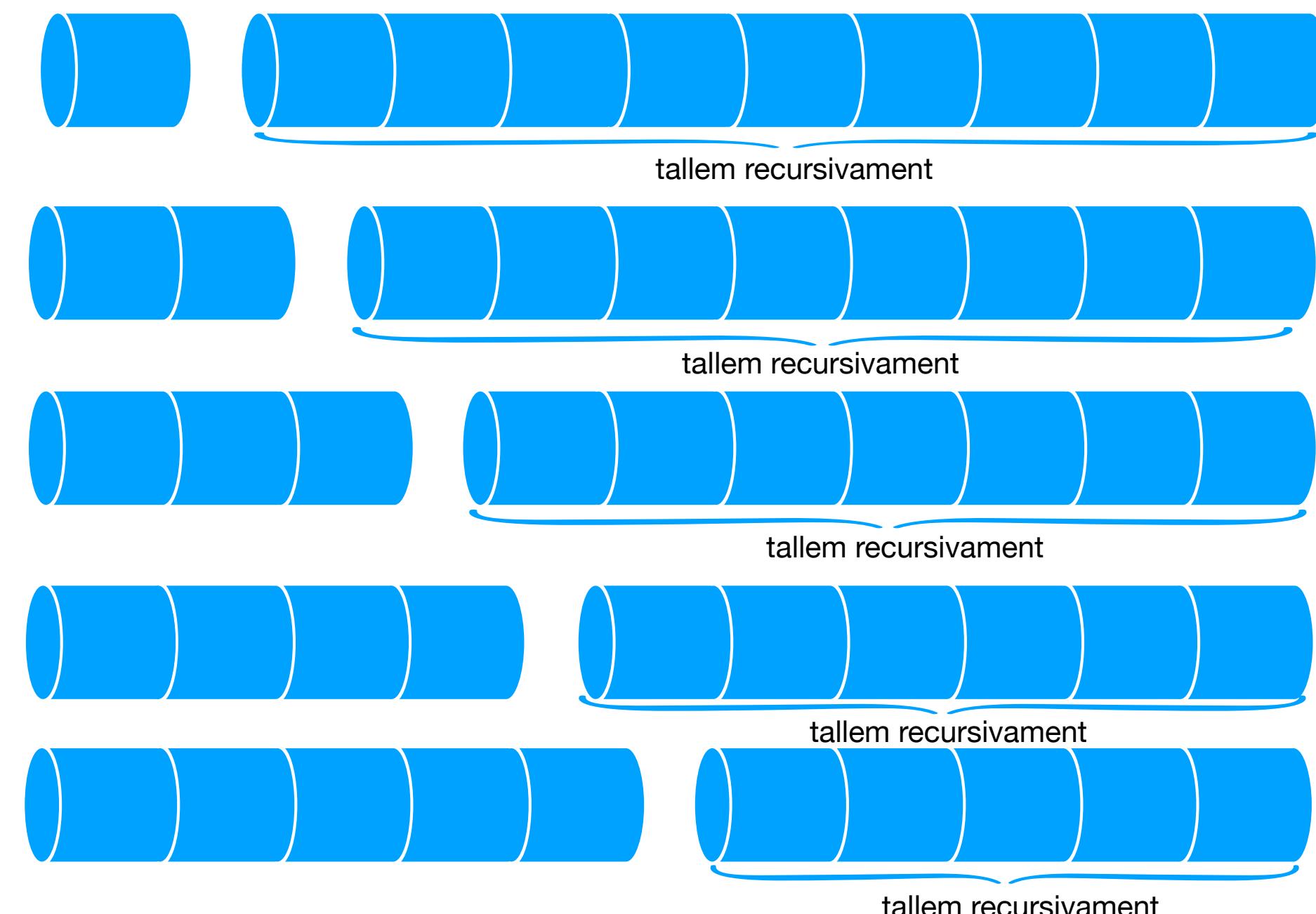
Solució òptima: 2 segments de 2
10



Programació Dinàmica

Rod-cutting problem

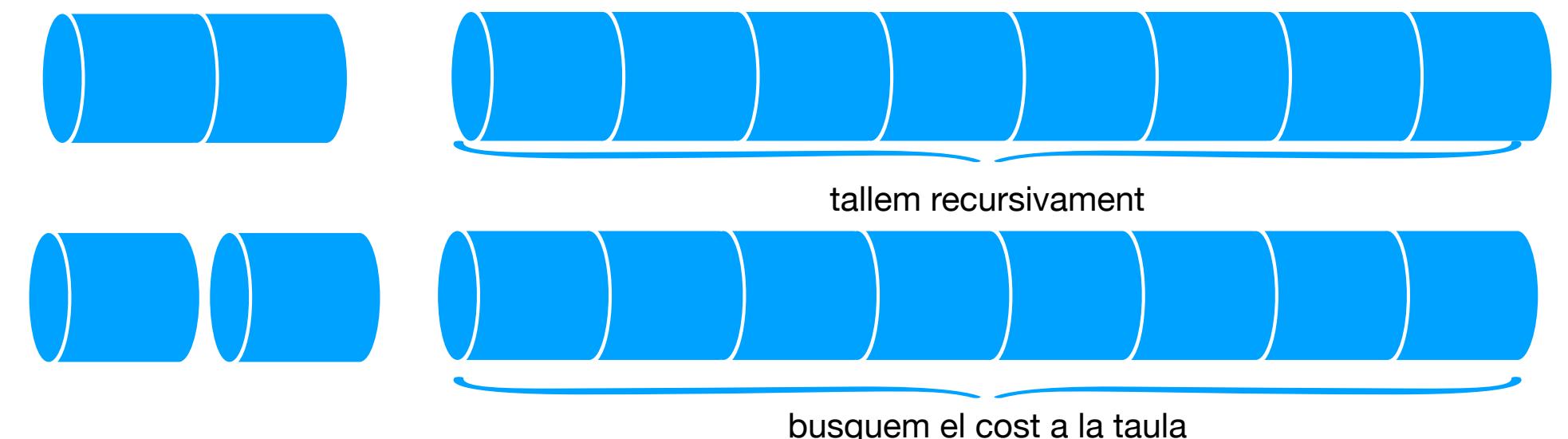
- Pensem ara una solució mitjançant un **algoritme amb programació dinàmica**
- **1a estratègia:** Per cada possible tall, calculem el valor de la part esquerra + el valor de la part dreta. Agafem el millor tall. (--> força bruta)



Programació Dinàmica

Rod-cutting problem

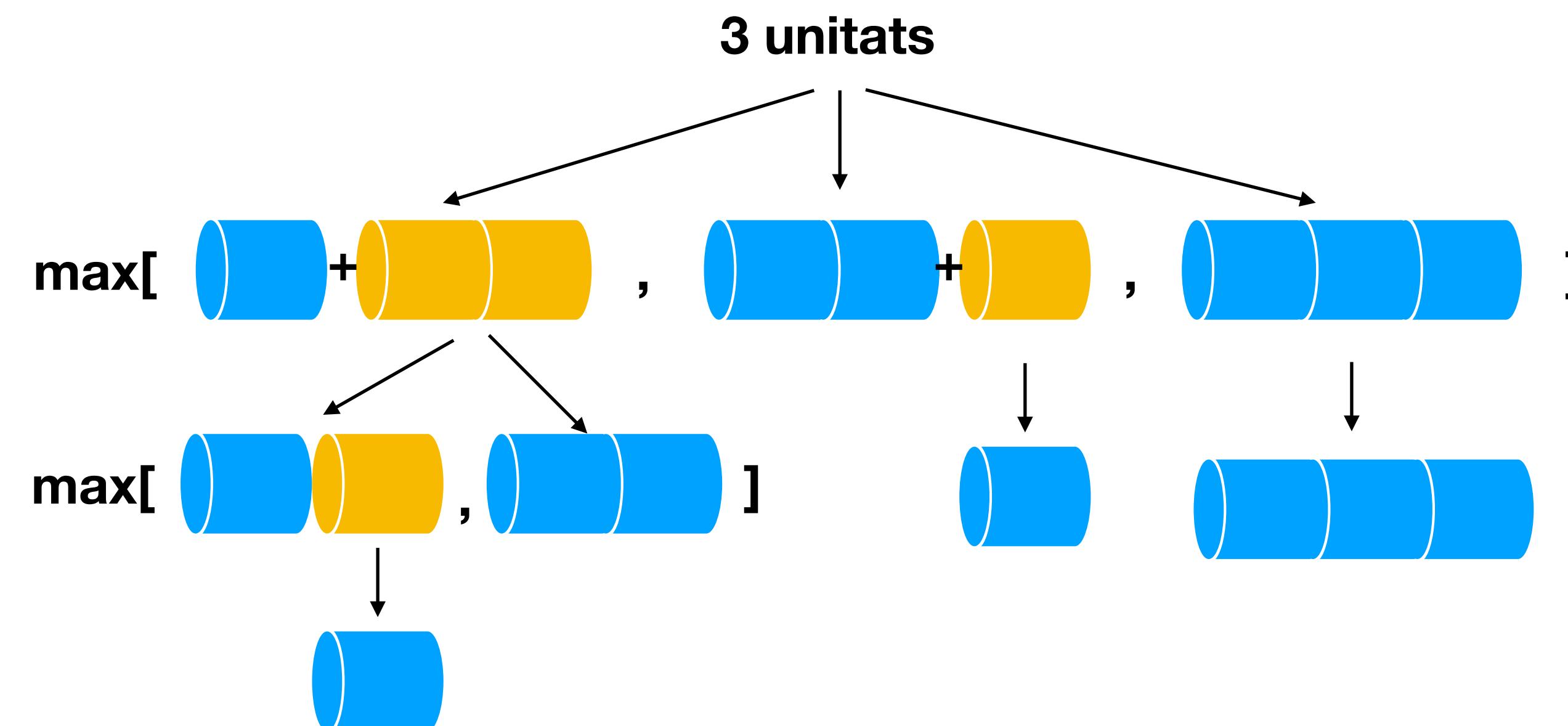
- Pensem ara una solució mitjançant un **algoritme amb programació dinàmica**
 - **2na estratègia (estratègia top-down):** Per cada possible tall, calculem el valor de la part esquerra i el guardem en una taula
 - Busquem el valor òptim del tall per la part dreta a la taula
 - El calculem de forma recursiva si encara no existeix
- Reduïm la complexitat de $O(2^n)$ a $O(n^2)$. Per què?
- Necessitem un array de longitud n per tal de guardar els càculs



Programació Dinàmica

Rod-cutting problem

- Pensem ara una solució mitjançant un **algoritme amb programació dinàmica**
- **2na estratègia (estratègia top-down):** Per cada possible tall, calculem el valor de la part esquerra i el guardem en una taula



Programació Dinàmica

Rod-cutting problem

- El valor optim d'una vareta de longitud n , r_n :
 - $r_n = \max(p_n, p_{n-1} + r_1, \dots, p_1 + r_{n-1})$
 - En altres paraules:
 - $r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i})$
 - On p_i és el preu d'una vareta de longitud i , r_{n-i} és el màxim benefici que podem tenir amb una vareta de longitud $n - 1$.

Programació Dinàmica

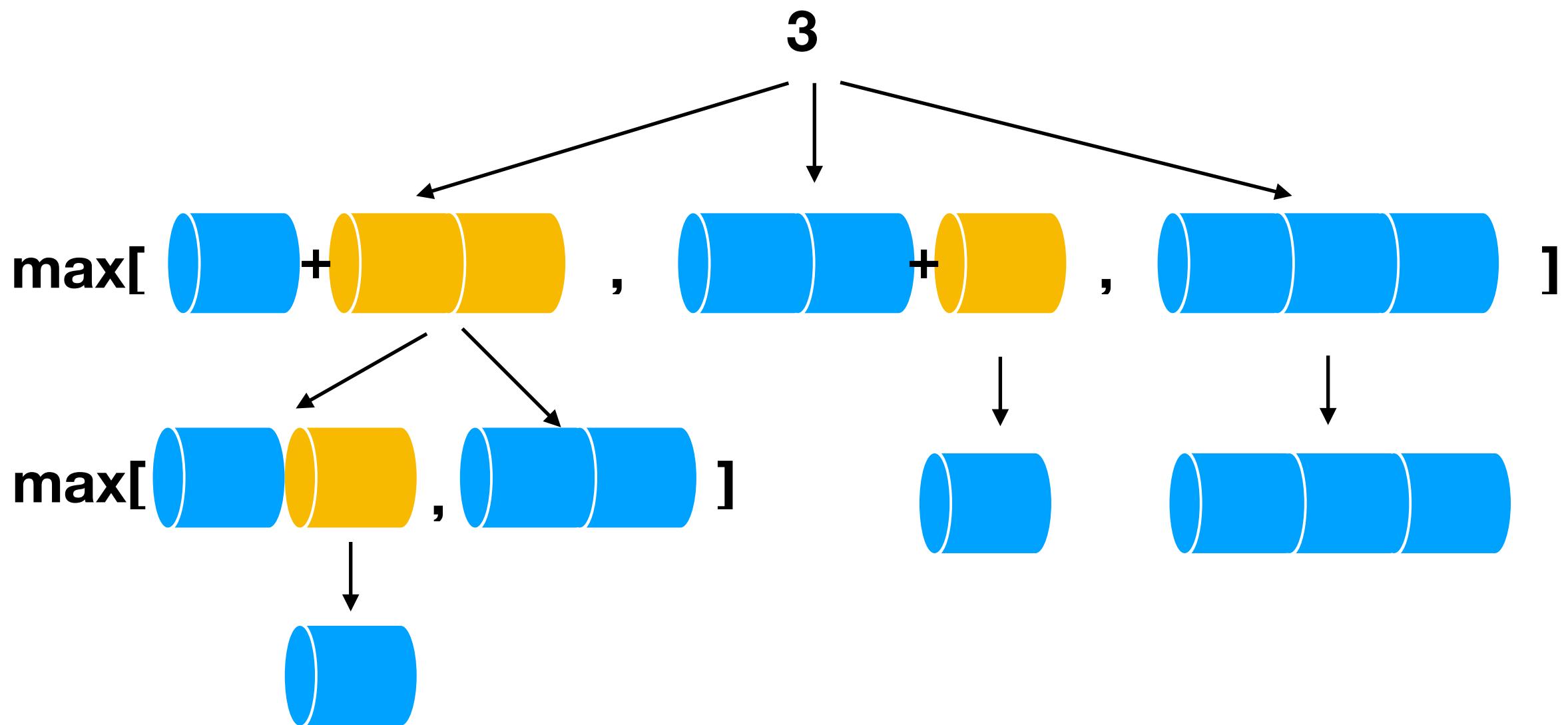
Rod-cutting - topDown solution

```
In [1]: def rodCut_top_down(price, n, memo = None):
    if memo == None:
        memo = [-1]*(n+1)
    if n <= 0:
        return 0
    # Check if the result already exists
    if memo[n]>0:
        return memo[n]

    # Recursively cut the road in different pieces
    # and compare different configurations
    max_val = -1
    for i in range(0,n):
        max_val = max(max_val,
                      price[i] + rodCut_top_down(price, n - i -1, memo))
    memo[n] = max_val
    return memo[n]

prices = [1, 5, 8, 9, 10, 17, 17, 20]
size = len(prices)
print("Maximum Obtainable value is", rodCut_top_down(prices, size))
```

Maximum Obtainable value is 22



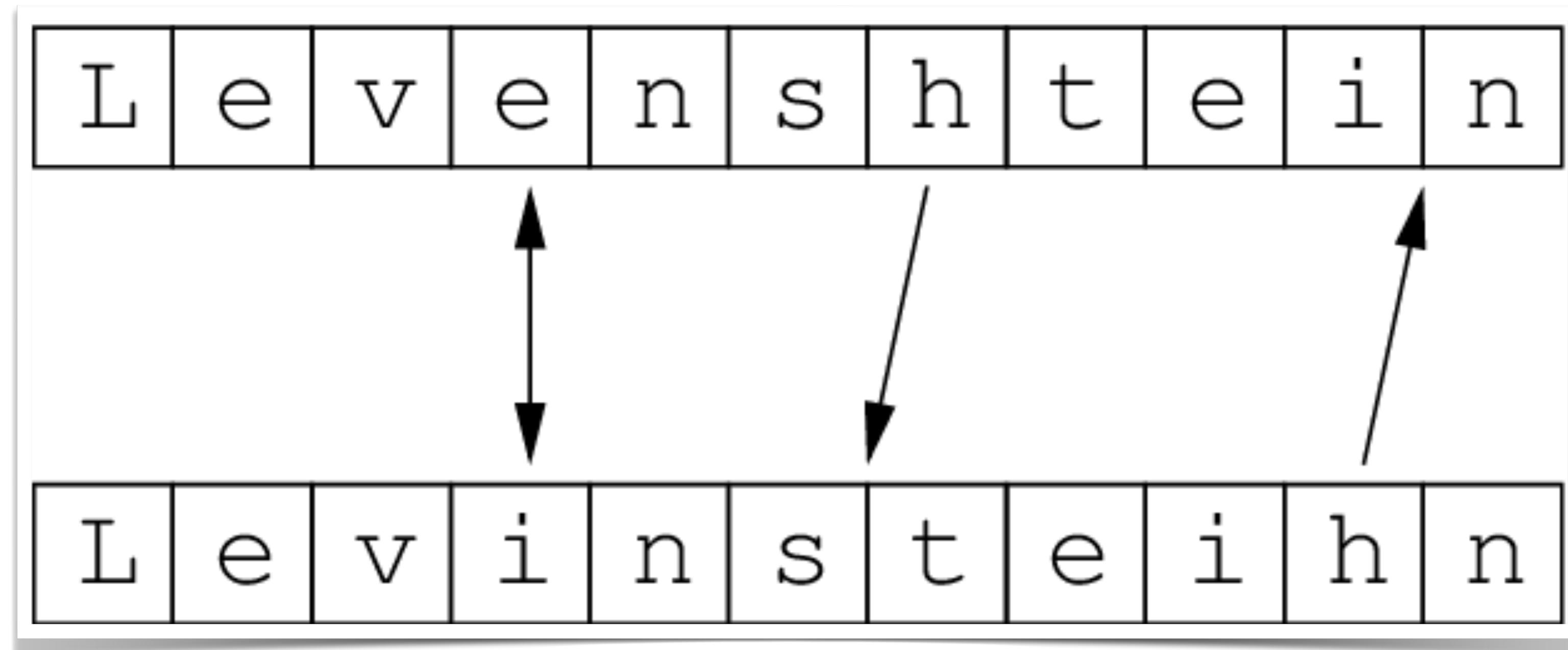
Programació Dinàmica

Rod-cutting problem

- Pensem ara una solució mitjançant un **algoritme amb programació dinàmica**
- **3ra estratègia (estratègia bottom-up):**
 - Calclem el cost de longitud 1 i el guardem a la taula
 - Calclem el cost de longitud 2. Únicament el podem tallar amb dos peces de cost 1. El cost ja està calculat i disponible a la taula. No hi ha cap crida recursiva.
 - Calclem el cost longituds successives fins a la longitud n. Els valors òptims de longituds inferior ja han estat calculats anteriorment
- Necessitem un vector de longitud n per tal de guardar els càlculs

Programació Dinàmica

Problema cost edició

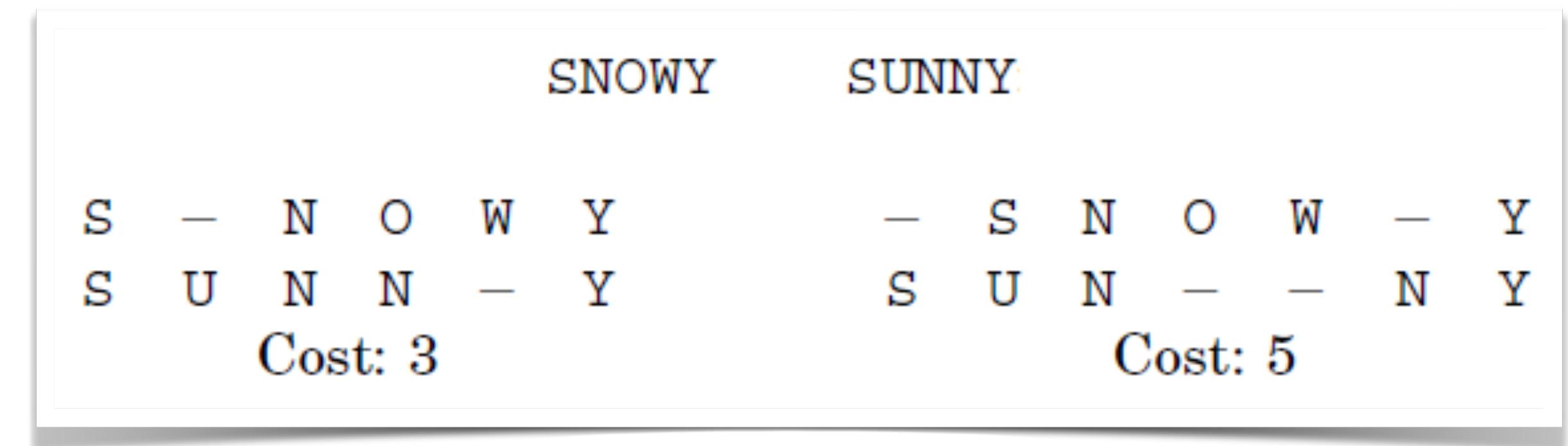


Programació Dinàmica

Problema cost edició

- Aplicació: **distància d'edició**
 - Donades dues paraules per exemple: **SNOWY** i **SUNNY** quina és la distància d'edició?
 - Tenim diferents accions: **Inserció, Eliminació i Substitució.**
 - Cada acció té un cost, suposem que el cost de cada acció és igual a 1. La distància consisteix en el nombre d'accions utilitzades per transformar una paraula p_1 a una paraula p_2 .
 - Sense dubtes hi ha moltes possibles solicions. Per exemple:

- S - S (mantenim la S: Cost 0)
- _ - U (inserim la U: Cost 1)
- N - N (mantenim la N: Cost 0)
- O - N (substituïm la O per N: Cost: 1)
- W - _ (eliminem la W: Cost 1)
- Y - Y (mantenim la Y: Cost 0)



- S - U (substituïm la S per U: Cost 1)
- N - N (mantenim la N: Cost 0)
- O - _ (eliminem la O Cost 1)
- W - _ (eliminem la W: Cost: 1)
- _ - N (inserim la N: Cost 1)
- Y - Y (mantenim Y: Cost 0)

Programació Dinàmica

Problema cost edició - Aplicacions

- **Correcció de text**

- L'usuari ha escrit: **graffe**
- Quina és la paraula més propera? graf | graft | grail | giraffe

- Biologia computacional

- Alinear dues seqüències d'ADN
 - AGGCTATCACCTGACCTCCAGGCCGATGCC
 - TAGCTATCACGACCGCGGTGATTGCCCGAC
- Resultat de l'alineació
 - **-AGGCTATCACCTGACCTCCAGGCCA -- TGCCC-**
 - **TAG- CTATCAC -- GACCGC- -GGTCGA**TTTGCCCCGAC
- Traducció de documents, reconeixement de veu,....

Programació Dinàmica

Problema cost edició

- Si cada operació té un cost de 1
 - La distància entre les paraules és 5
- Si les substitucions tenen cost 2
 - La distància entre les paraules és 8

INTENTION → EXECUTION

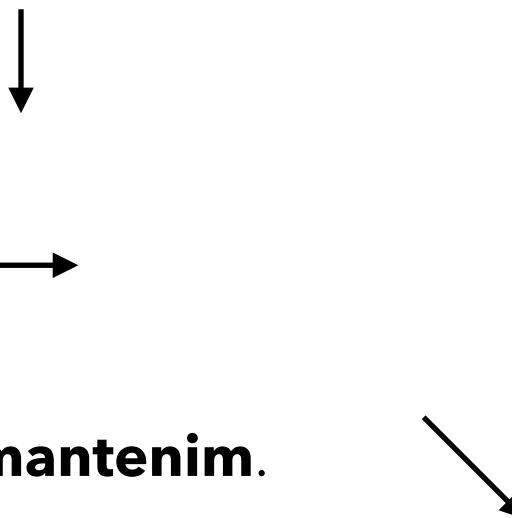
I	N	T	E	*	N	T	I	O	N
*	E	X	E	C	U	T	I	O	N
d	s	s		i	s				

Programació Dinàmica

Problema cost edició

- Podem definir la solució mitjançant una **funció recurrent** tenint en compte els següents punts:
 - Cost d'edició $E(i, j)$ serà el mínim entre:

- $E(i - 1, j) + 1 \rightarrow$ **afegir** una lletra
- $E(i, j - 1) + 1 \rightarrow$ **eliminar** una lletra
- $E(i - 1, j - 1) + diff(i, j) \rightarrow$ **substituïm** o **mantenim.**



$$E(i, j) = \min\{E(i - 1, j) + 1, E(i, j - 1) + 1, E(i - 1, j - 1) + \text{diff}(i, j)\}$$

	$j - 1$	j	n
$i - 1$			
i			
m			
			GOAL

	P	O	L	Y	N	O	M	I	A	L
E	0	1	2	3	4	5	6	7	8	9
X	1									10
P	2									
O	3									
Y	4									
N	5									
O	6									
M	7									
I	8									
A	9									
L	10									
L	11									

- **Exemple:** El cost d'edició de la paraula (POL) per (EXP) vindrà definit per: $E(POL, EXP) = \min\{E(PO, EXP) + 1, E(POL, EX) + 1, E(PO, EX) + \text{dif}(L, X)\}$
- Si executem aquesta funció recursiva d'esquerra a dreta i de dalt a sota, podem emplenar aquesta taula i obtindrem el següent resultat:

```
def editDistance(str1, str2, m, n):

    # Si el primer string té longitud 0 , l'única opció
    # és afegir els caràcters del segon string
    if m == 0:
        return n

    # Si el segon string té longitud 0, l'única opció
    # és eliminar els caràcters del primer string
    if n == 0:
        return m

    # Si l'últim caràcter dels dos string és el mateix
    if str1[m-1] == str2[n-1]:
        return editDistance(str1, str2, m-1, n-1)

    # Si l'últim caràcter dels dos string no és el mateix
    # cridem la funció recursiva de les diferents accions i agafem la de menor cost+1
    return 1 + min(editDistance(str1, str2, m, n-1),      # afegir
                  editDistance(str1, str2, m-1, n),      # eliminar
                  editDistance(str1, str2, m-1, n-1))   # reemplaçar
                    )

str1 = "sunny"
str2 = "snowy"
print editDistance(str1, str2, len(str1), len(str2))
```

Programació Dinàmica

Problema cost edició

- Pensem una solució amb **programació dinàmica**. Necessitem:
 1. *Definir funció de recursitat*
 2. *Trobar subestructura òptima*
 3. *Afegir casos base*

Programació Dinàmica

Problema cost edició

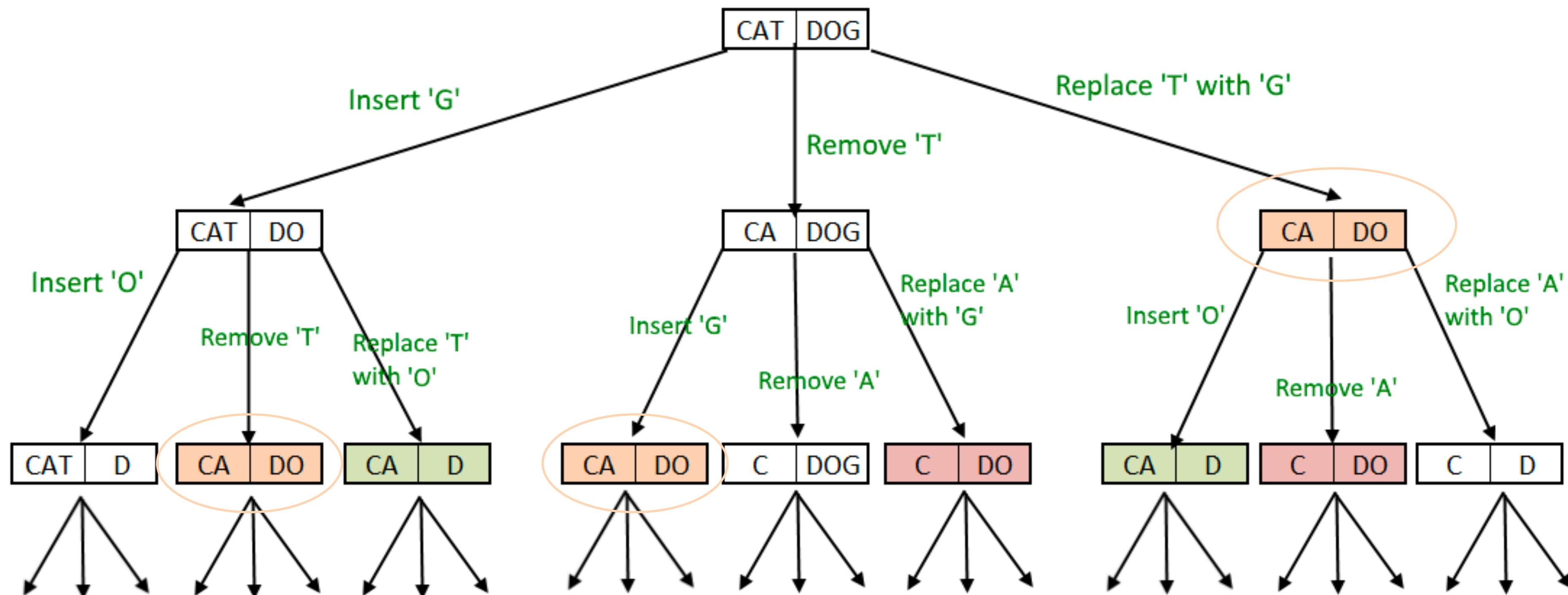
- Pensem una solució amb **programació dinàmica**. Necessitem:
 1. *Definir funció de recursitat.*

Ja l'hem vist abans:

```
# Si l'últim caràcter dels dos string no és el mateix
# cridem la funció recursiva de les diferents accions i agafem la de menor cost+1
return 1 + min(editDistance(str1, str2, m, n-1),      # afegir
               editDistance(str1, str2, m-1, n),      # eliminar
               editDistance(str1, str2, m-1, n-1))    # reemplaçar
)
```

Programació Dinàmica

Problema cost edició



Programació Dinàmica

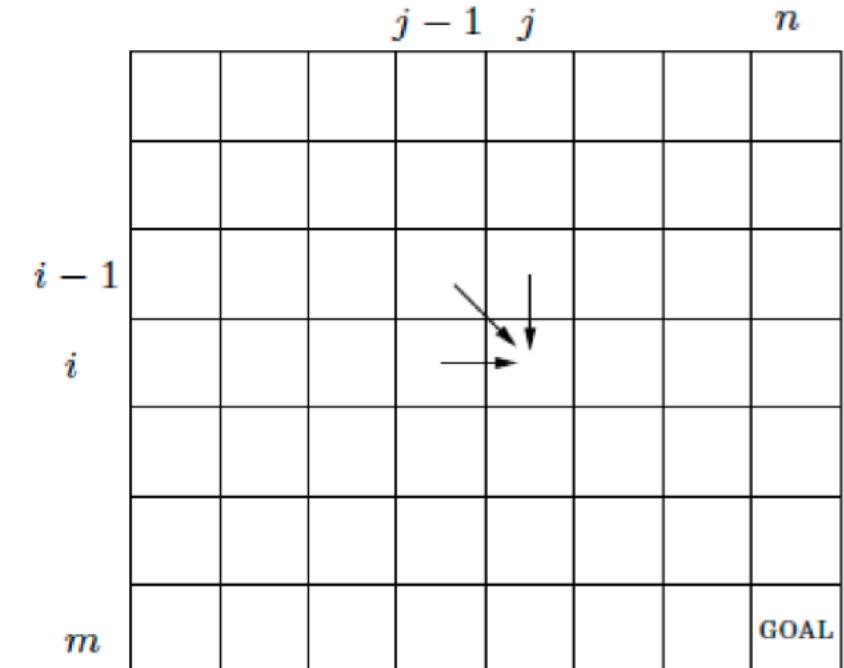
Problema cost edició

- Pensem una solució amb **programació dinàmica**. Necessitem:

1. *Definir funció de recursitat.*

$$E(i, j) = \min\{E(i - 1, j) + 1, E(i, j - 1) + 1, E(i - 1, j - 1) + \text{diff}(i, j)\}$$

2. *Trobar subestructura òptima*



	P	O	L	Y	N	O	M	I	A	L
E	0	1	2	3	4	5	6	7	8	9
X	1									
P	2									
O	3									
N	4									
E	5									
N	6									
T	7									
I	8									
A	9									
L	10									
GOAL	11									

Programació Dinàmica

Problema cost edició

- Pensem una solució amb **programació dinàmica**. Necessitem:

1. *Definir funció de recursitat.*
2. *Trobar subestructura òptima*
3. *Afegir casos base*

```
# Si el primer string té longitud 0 , l'única opció  
# és afegir els caràcters del segon string
```

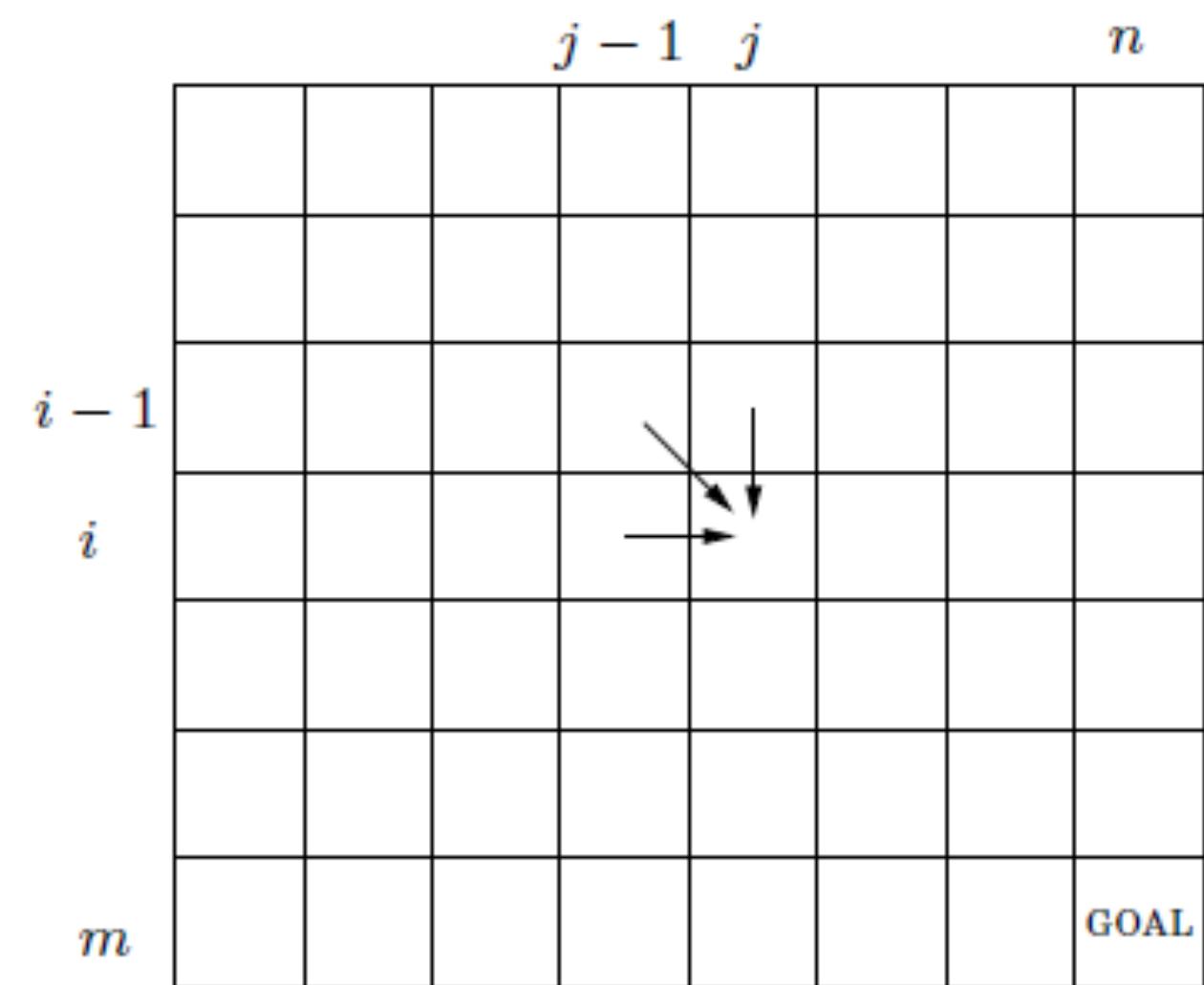
```
# Si el segon string té longitud 0, l'única opció  
# és eliminar els caràcters del primer string
```

Programació Dinàmica

Problema cost edició

- **Exemple**

$$E(i, j) = \min\{E(i - 1, j) + 1, E(i, j - 1) + 1, E(i - 1, j - 1) + \text{diff}(i, j)\}$$



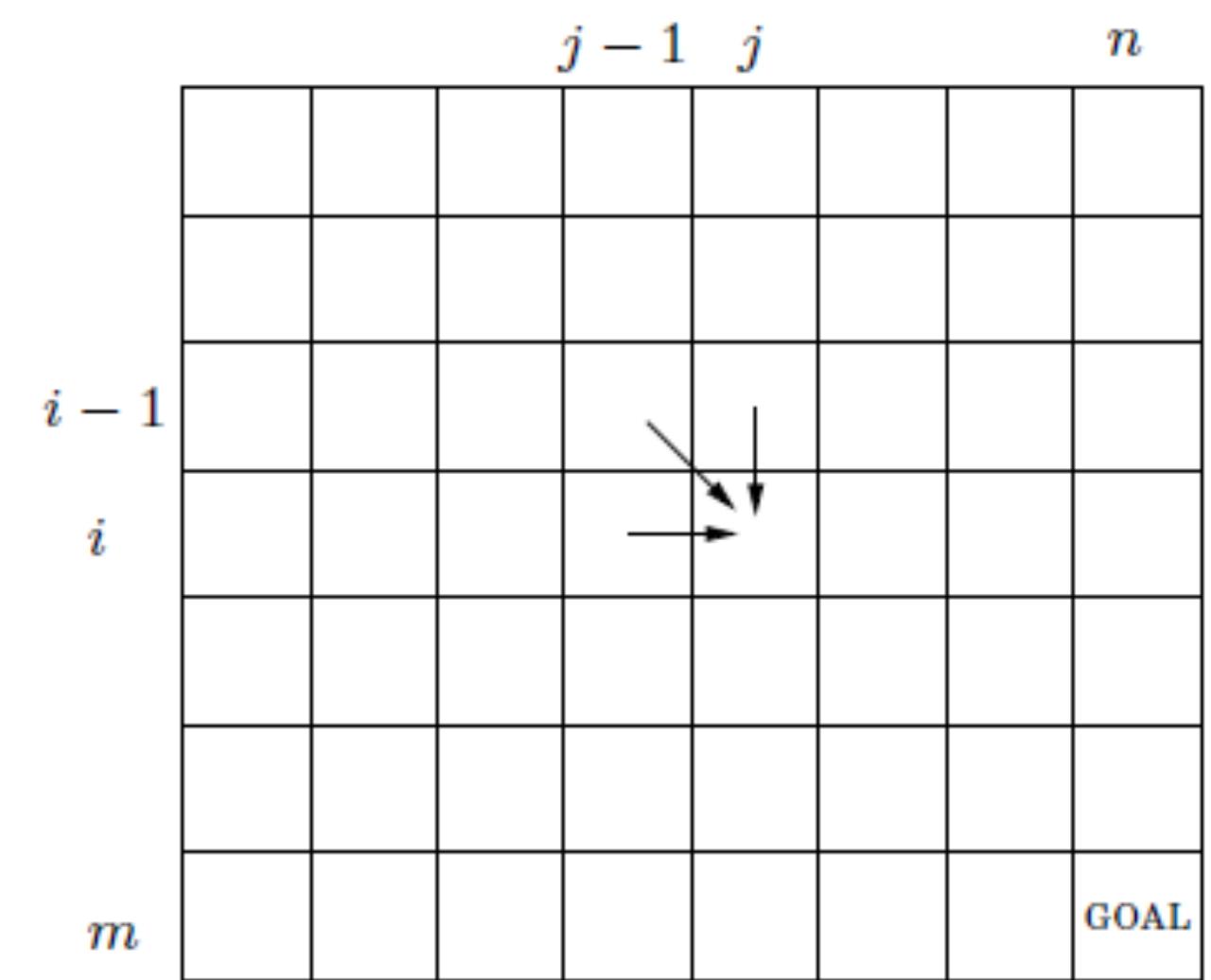
	P	O	L	Y	N	O	M	I	A	L
E	0	1	2	3	4	5	6	7	8	9
X	1									
P	2									
O	3									
N	4									
E	5									
N	6									
T	7									
I	8									
A	9									
L	10									
	11									

Programació Dinàmica

Problema cost edició

- Taula de subproblemes

$$E(i, j) = \min\{E(i - 1, j) + 1, E(i, j - 1) + 1, E(i - 1, j - 1) + \text{diff}(i, j)\}$$



	P	O	L	Y	N	O	M	I	A	L
E	0 1	1	2	3	4	5	6	7	8	9
X	1	2								
P	2	2								
O	3	2								
N	4	3								
E	5	4								
N	6	5								
T	7	6								
I	8	7								
A	9	8								
L	10	9								
L	11	10								

Programació Dinàmica

Problema cost edició

- Taula de subproblems

```
for i = 0, 1, 2, ..., m:  
    E(i, 0) = i  
for j = 1, 2, ..., n:  
    E(0, j) = j  
for i = 1, 2, ..., m:  
    for j = 1, 2, ..., n:  
        E(i, j) = min{E(i - 1, j) + 1, E(i, j - 1) + 1, E(i - 1, j - 1) + diff(i, j)}  
return E(m, n)
```

Depèn de l'objectiu del problemes

E	X	P	O	N	E	N	-	T	I	A	L
-	-	P	O	L	Y	N	O	M	I	A	L

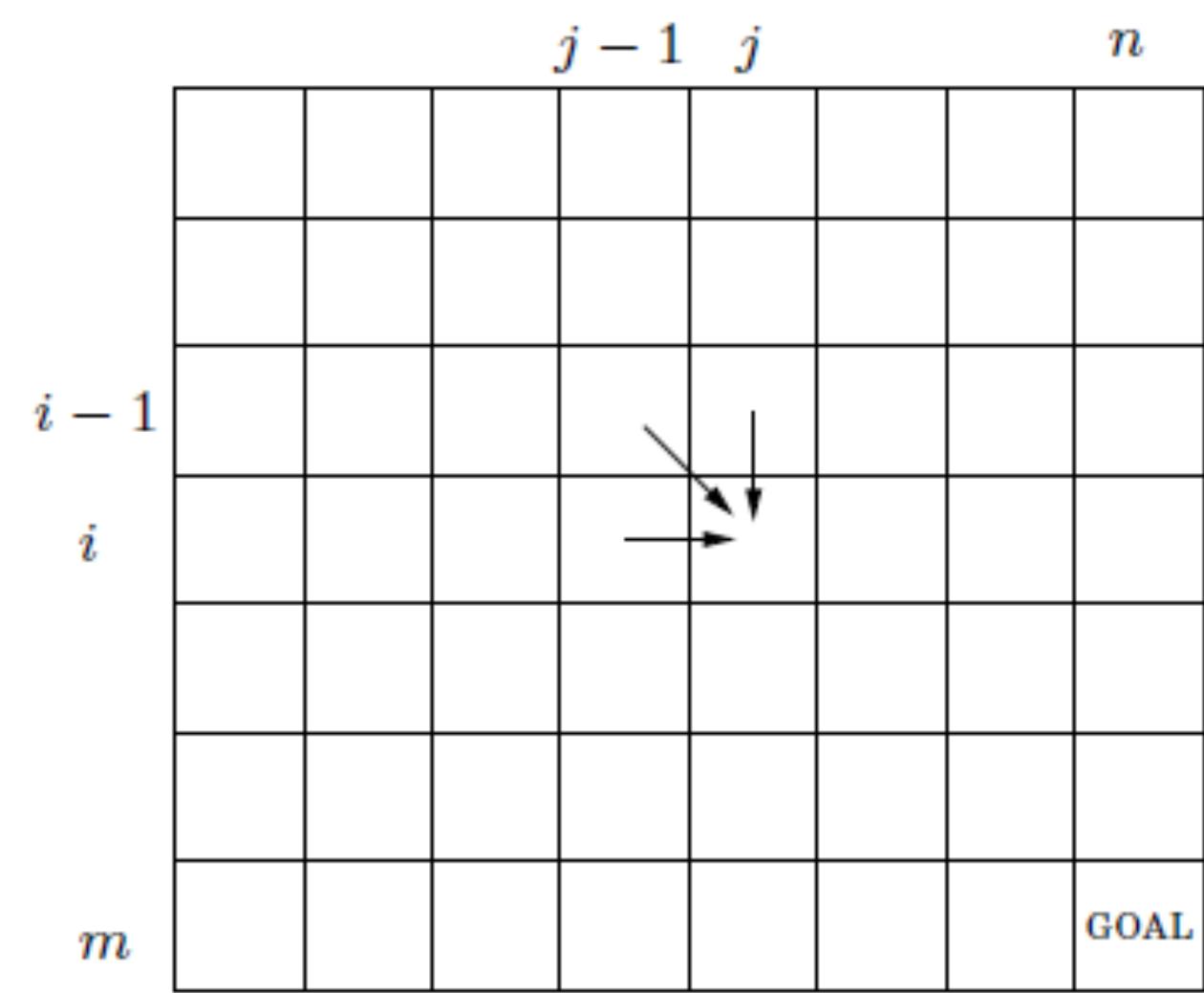
$O(mn)$

Programació Dinàmica

Problema cost edició

- Exemple

$$E(i, j) = \min\{E(i - 1, j) + 1, E(i, j - 1) + 1, E(i - 1, j - 1) + \text{diff}(i, j)\}$$



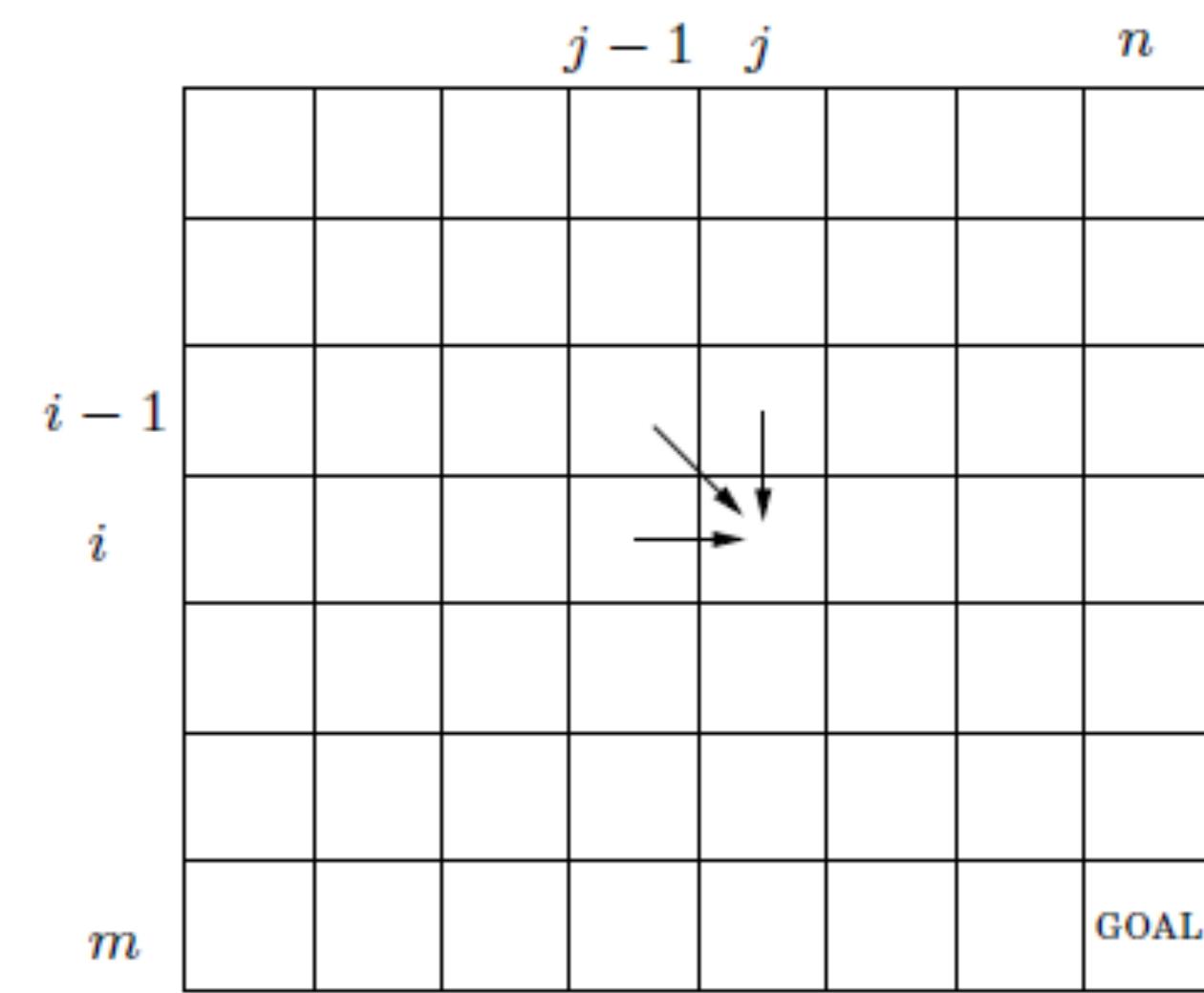
	P	O	L	Y	N	O	M	I	A	L
E	0	1	2	3	4	5	6	7	8	9
X	1	1								
X	2	2								
P	3	2								
O	4	3								
N	5	4								
E	6	5								
N	7	6								
T	8	7								
I	9	8								
A	10	9								
L	11	10								

Programació Dinàmica

Problema cost edició

- Exemple

$$E(i, j) = \min\{E(i - 1, j) + 1, E(i, j - 1) + 1, E(i - 1, j - 1) + \text{diff}(i, j)\}$$



	P	O	L	Y	N	O	M	I	A	L
E	0	1	2	3	4	5	6	7	8	9
E	1	1	2	3	4	5	6	7	8	9
X	2	2	2	3	4	5	6	7	8	9
P	3	2	3	3	4	5	6	7	8	9
O	4	3	2	3	4	5	5	6	7	8
N	5	4	3	3	4	4	5	6	7	8
E	6	5	4	4	5	5	6	7	8	9
N	7	6	5	5	4	5	6	7	8	9
T	8	7	6	6	5	5	6	7	8	9
I	9	8	7	7	6	6	6	6	7	8
A	10	9	8	8	8	7	7	7	6	7
L	11	10	9	8	9	8	8	8	7	6

Exercici: Programació dinàmica

- Associar la paraula ALGORISMICA amb la paraula AVANÇADA fent ús d'aquesta inicialització i funció de programació dinàmica

		A	L	G	O	R	I	S	M	I	C	A
	0	1										
A	1	0										
V												
A												
N												
Ç												
A												
D												
A												

Exercici: Programació dinàmica

- Associar la paraula ALGORISMICA amb la paraula AVANÇADA fent ús d'aquesta inicialització i funció de programació dinàmica

		A	L	G	O	R	I	S	M	I	C	A
	0	1	2	3	4	5	6	7	8	9	10	11
A	1	0	1									
V	2	1	1									
A	3	2	2									
N	4	3	3									
Ç	5	4	4									
A	6	5	5									
D	7	6	6									
A	8	7	7									

Exercici: Programació dinàmica

- Com modificaríeu el codi tal que el cost **d'afegir** una lletra tingui un **cost de 2 en lloc de 1**.
- Quin seria el resultat obtingut de transformar la paraula **Avançada** a **Algorísmica**.

		A	L	G	O	R	I	S	M	I	C	A
	0	1	2	3	4	5	6	7	8	9	10	11
A	2	0	1	2	3	4	5	6	7	8	9	10
V	4	2	1	2	3	4	5	6	7	8	9	10
A	6	4	3	2	3	4	5	6	7	8	9	9
N	8	6	5	4	3	4	5	6	7	8	9	10
Ç	10	8	7	6	5	4	5	6	7	8	9	10
A	12	10	9	8	7	6	5	6	7	8	9	9
D	14	12	11	10	9	8	7	6	7	8	9	10
A	16	14	13	12	11	10	9	8	7	8	9	9

Programació Dinàmica

Hem trobat el cost, com trobem el camí?

- **backtrace**

	P	O	L	Y	N	O	M	I	A	L	
E	0	1	2	3	4	5	6	7	8	9	10
X	1	1	2	3	4	5	6	7	8	9	10
P	2	2	3	4	5	6	7	8	9	10	
O	3	2	3	4	5	6	7	8	9	10	
N	4	3	2	3	4	5	6	7	8	9	
E	5	4	3	3	4	5	6	7	8	9	
N	6	5	4	4	4	5	6	7	8	9	
T	7	6	5	5	5	4	6	7	8	9	
I	8	7	6	6	6	5	5	7	8	9	
A	9	8	7	7	7	6	6	6	7	8	
L	10	9	8	8	8	7	7	7	6	7	
	11	10	9	8	9	8	8	8	7	6	

Programació Dinàmica

El camí més curt



Programació Dinàmica

El camí més curt

Quins algoritmes coneixem?

Dijkstra Bellman Ford

Programació dinàmica

Floyd Warshall



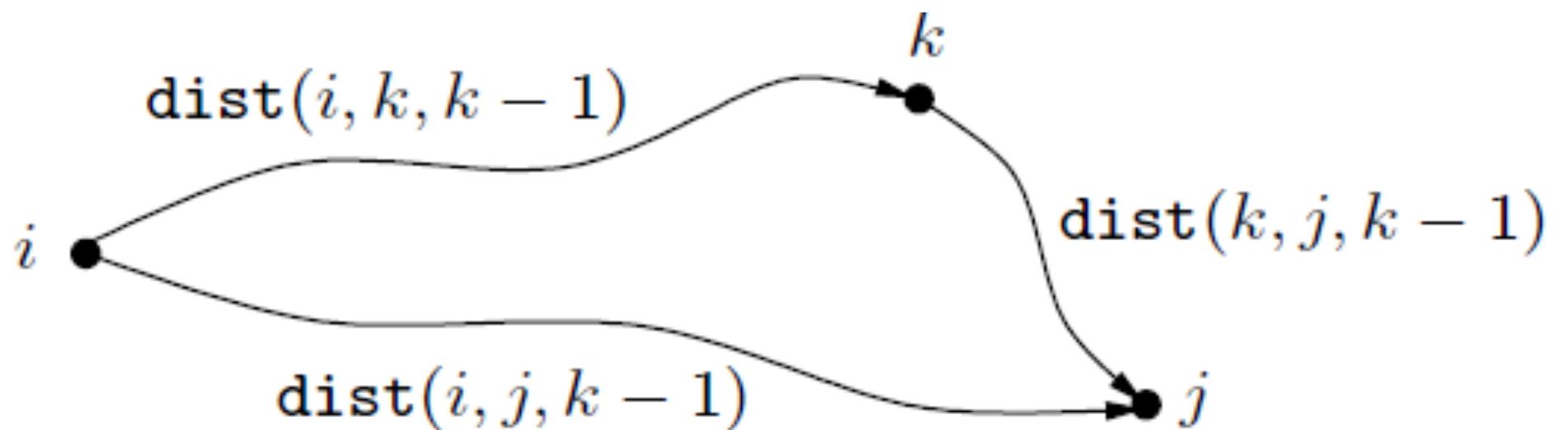
Programació Dinàmica

Algoritme **Floyd Warshall**

- Fem ús d'una matriu tridimensional, on:

$$dist(i, j, k) \{1, 2, \dots, k\}$$

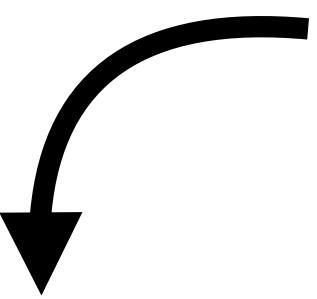
és la distància del camí més curt entre i i j tenint en compte només k nodes.



$$dist(i, k, k - 1) + dist(k, j, k - 1) < dist(i, j, k - 1),$$

Programació Dinàmica

Algoritme Floyd Warshall



```
1 let dist be a |V| × |V| array of minimum distances initialized to ∞ (infinity)
2 for each edge (u,v)
3   dist[u][v] ← w(u,v) // the weight of the edge (u,v)
4 for each vertex v
5   dist[v][v] ← 0
6 for k from 1 to |V|
7   for i from 1 to |V|
8     for j from 1 to |V|
9       if dist[i][j] > dist[i][k] + dist[k][j]
10      dist[i][j] ← dist[i][k] + dist[k][j]
11    end if
```

```
for  $i = 1$  to  $n$ :
  for  $j = 1$  to  $n$ :
     $\text{dist}(i, j, 0) = \infty$ 

  for all  $(i, j) \in E$ :
     $\text{dist}(i, j, 0) = \ell(i, j)$ 

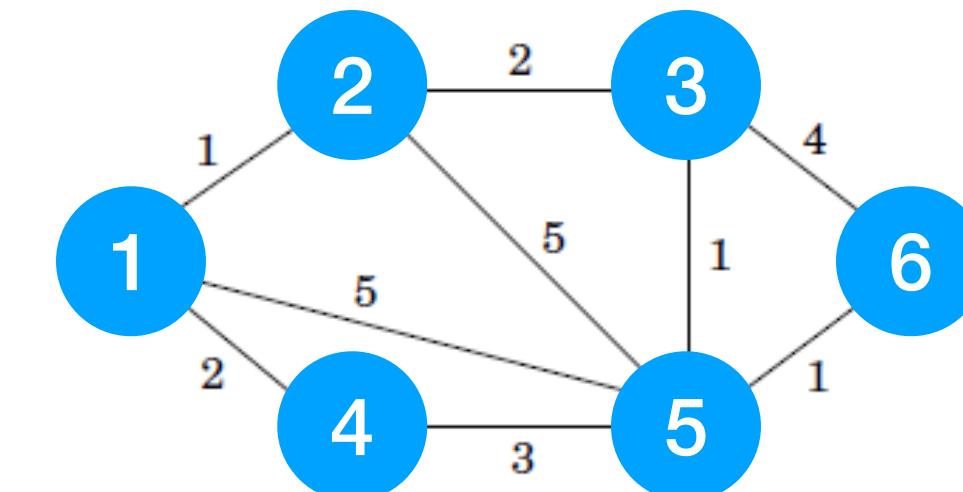
for  $k = 1$  to  $n$ :
  for  $i = 1$  to  $n$ :
    for  $j = 1$  to  $n$ :
       $\text{dist}(i, j, k) = \min\{\text{dist}(i, k, k - 1) + \text{dist}(k, j, k - 1), \text{dist}(i, j, k - 1)\}$ 
```

Programació Dinàmica

Algoritme Floyd Warshall

```
for  $i = 1$  to  $n$ :  
    for  $j = 1$  to  $n$ :  
         $\text{dist}(i, j, 0) = \infty$   
  
    for all  $(i, j) \in E$ :  
         $\text{dist}(i, j, 0) = \ell(i, j)$   
    for  $k = 1$  to  $n$ :  
        for  $i = 1$  to  $n$ :  
            for  $j = 1$  to  $n$ :  
                 $\text{dist}(i, j, k) = \min\{\text{dist}(i, k, k - 1) + \text{dist}(k, j, k - 1), \text{dist}(i, j, k - 1)\}$ 
```

	1	2	3	4	5	6
1	-	1	-	2	5	-
2	1	-	2	-	5	-
3	-	2	-	-	1	4
4	2	-	-	-	3	-
5	5	5	1	3	-	1
6	-	-	4	-	1	-



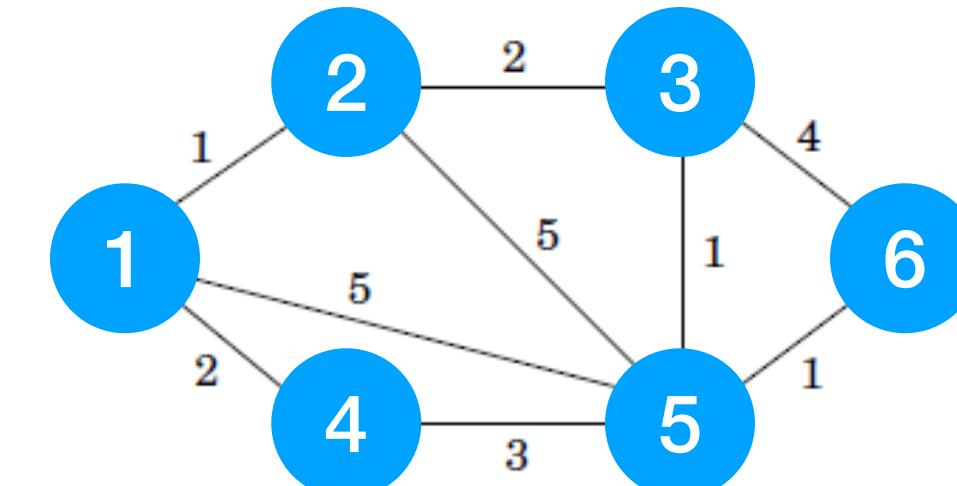
Programació Dinàmica

Algoritme Floyd Warshall

```
for  $i = 1$  to  $n$ :  
    for  $j = 1$  to  $n$ :  
         $\text{dist}(i, j, 0) = \infty$   
  
    for all  $(i, j) \in E$ :  
         $\text{dist}(i, j, 0) = \ell(i, j)$   
    for  $k = 1$  to  $n$ :  
        for  $i = 1$  to  $n$ :  
            for  $j = 1$  to  $n$ :  
                 $\text{dist}(i, j, k) = \min\{\text{dist}(i, k, k - 1) + \text{dist}(k, j, k - 1), \text{dist}(i, j, k - 1)\}$ 
```

$k = 1$

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



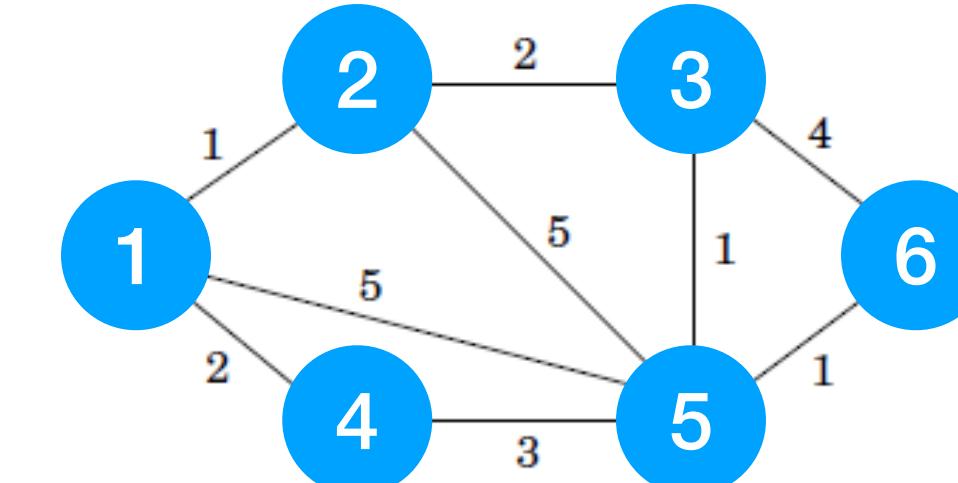
Programació Dinàmica

Algoritme Floyd Warshall

```
for  $i = 1$  to  $n$ :  
    for  $j = 1$  to  $n$ :  
         $\text{dist}(i, j, 0) = \infty$   
  
    for all  $(i, j) \in E$ :  
         $\text{dist}(i, j, 0) = \ell(i, j)$   
    for  $k = 1$  to  $n$ :  
        for  $i = 1$  to  $n$ :  
            for  $j = 1$  to  $n$ :  
                 $\text{dist}(i, j, k) = \min\{\text{dist}(i, k, k - 1) + \text{dist}(k, j, k - 1), \text{dist}(i, j, k - 1)\}$ 
```

$k = 2$

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



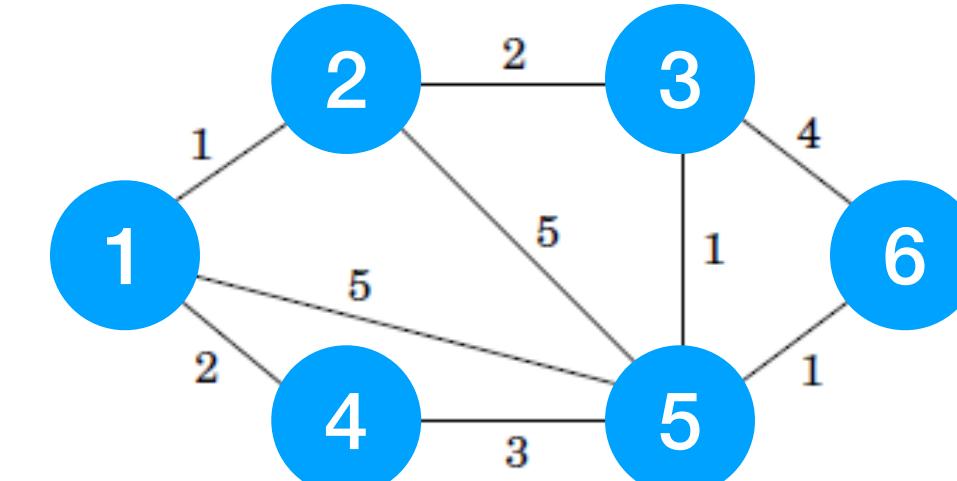
Programació Dinàmica

Algoritme Floyd Warshall

```
for  $i = 1$  to  $n$ :  
    for  $j = 1$  to  $n$ :  
         $\text{dist}(i, j, 0) = \infty$   
  
    for all  $(i, j) \in E$ :  
         $\text{dist}(i, j, 0) = \ell(i, j)$   
    for  $k = 1$  to  $n$ :  
        for  $i = 1$  to  $n$ :  
            for  $j = 1$  to  $n$ :  
                 $\text{dist}(i, j, k) = \min\{\text{dist}(i, k, k - 1) + \text{dist}(k, j, k - 1), \text{dist}(i, j, k - 1)\}$ 
```

$k = 3$

	1	2	3	4	5	6
1	2	1	3	2	4	7
2	1	2	2	3	3	6
3	3	2	4	5	1	4
4	2	3	5	4	3	9
5	4	3	1	3	2	1
6	7	6	4	9	1	8



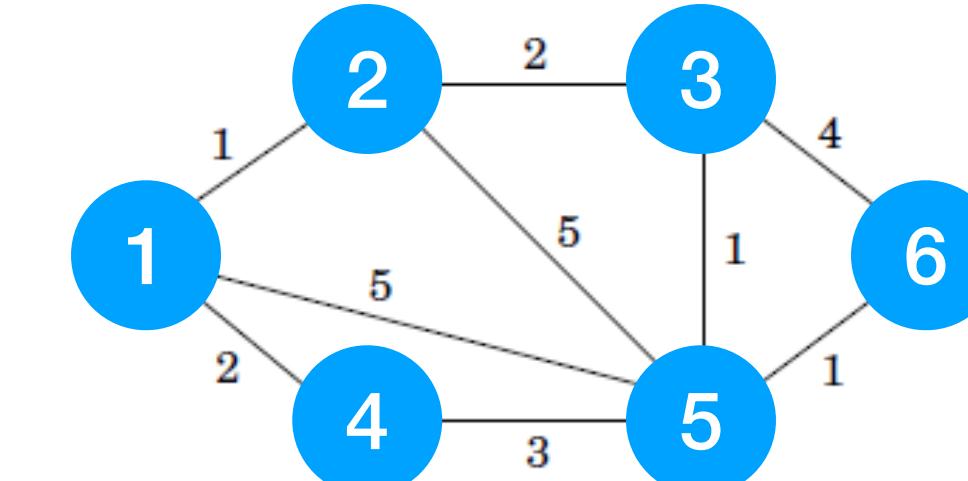
Programació Dinàmica

Algoritme Floyd Warshall

```
for  $i = 1$  to  $n$ :  
    for  $j = 1$  to  $n$ :  
         $\text{dist}(i, j, 0) = \infty$   
  
    for all  $(i, j) \in E$ :  
         $\text{dist}(i, j, 0) = \ell(i, j)$   
    for  $k = 1$  to  $n$ :  
        for  $i = 1$  to  $n$ :  
            for  $j = 1$  to  $n$ :  
                 $\text{dist}(i, j, k) = \min\{\text{dist}(i, k, k - 1) + \text{dist}(k, j, k - 1), \text{dist}(i, j, k - 1)\}$ 
```

$k = 4$

	1	2	3	4	5	6
1	2	1	3	2	4	7
2	1	2	2	3	3	6
3	3	2	4	5	1	4
4	2	3	5	4	3	9
5	4	3	1	3	2	1
6	7	6	4	9	1	8



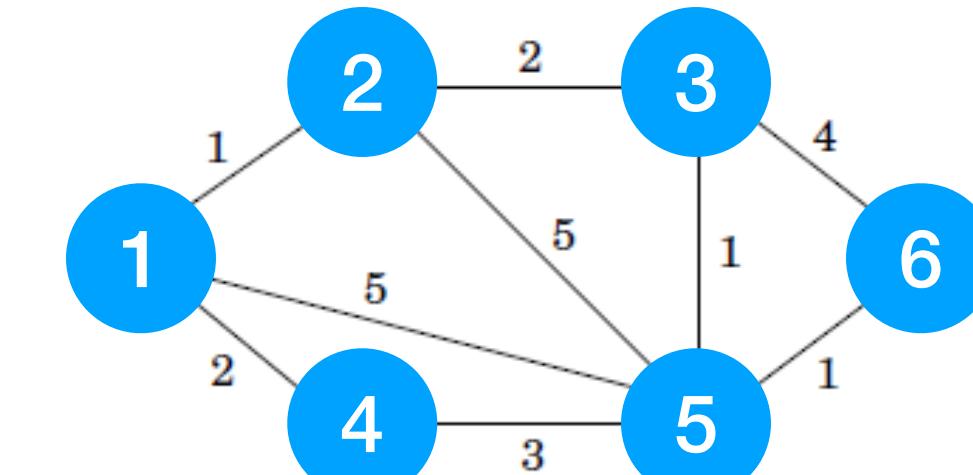
Programació Dinàmica

Algoritme Floyd Warshall

```
for  $i = 1$  to  $n$ :  
    for  $j = 1$  to  $n$ :  
         $\text{dist}(i, j, 0) = \infty$   
  
    for all  $(i, j) \in E$ :  
         $\text{dist}(i, j, 0) = \ell(i, j)$   
    for  $k = 1$  to  $n$ :  
        for  $i = 1$  to  $n$ :  
            for  $j = 1$  to  $n$ :  
                 $\text{dist}(i, j, k) = \min\{\text{dist}(i, k, k - 1) + \text{dist}(k, j, k - 1), \text{dist}(i, j, k - 1)\}$ 
```

$k = 5$

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



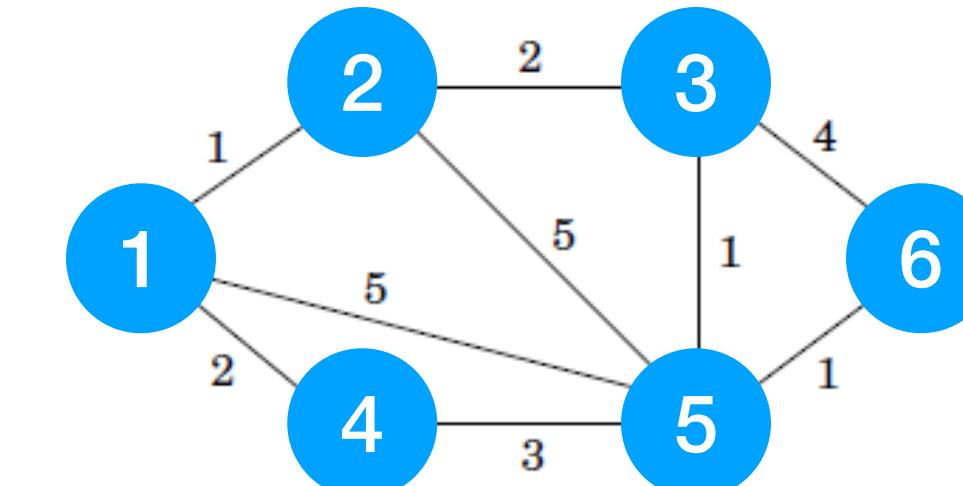
Programació Dinàmica

Algoritme Floyd Warshall

```
for  $i = 1$  to  $n$ :  
    for  $j = 1$  to  $n$ :  
         $\text{dist}(i, j, 0) = \infty$   
  
    for all  $(i, j) \in E$ :  
         $\text{dist}(i, j, 0) = \ell(i, j)$   
    for  $k = 1$  to  $n$ :  
        for  $i = 1$  to  $n$ :  
            for  $j = 1$  to  $n$ :  
                 $\text{dist}(i, j, k) = \min\{\text{dist}(i, k, k - 1) + \text{dist}(k, j, k - 1), \text{dist}(i, j, k - 1)\}$ 
```

$k = 6$

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



Programació Dinàmica

Algoritme **Floyd Warshall**

- Quina és la complexitat de l'algoritme?

```
for i = 1 to n:  
    for j = 1 to n:  
        dist(i, j, 0) = ∞  
  
    for all (i, j) ∈ E:  
        dist(i, j, 0) = ℓ(i, j)  
    for k = 1 to n:  
        for i = 1 to n:  
            for j = 1 to n:  
                dist(i, j, k) = min{dist(i, k, k - 1) + dist(k, j, k - 1), dist(i, j, k - 1)}
```

Programació Dinàmica

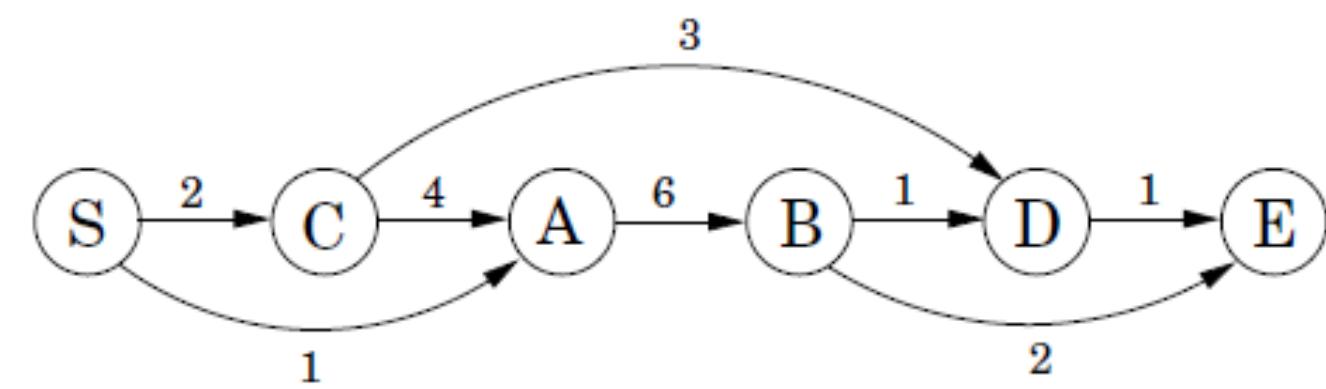
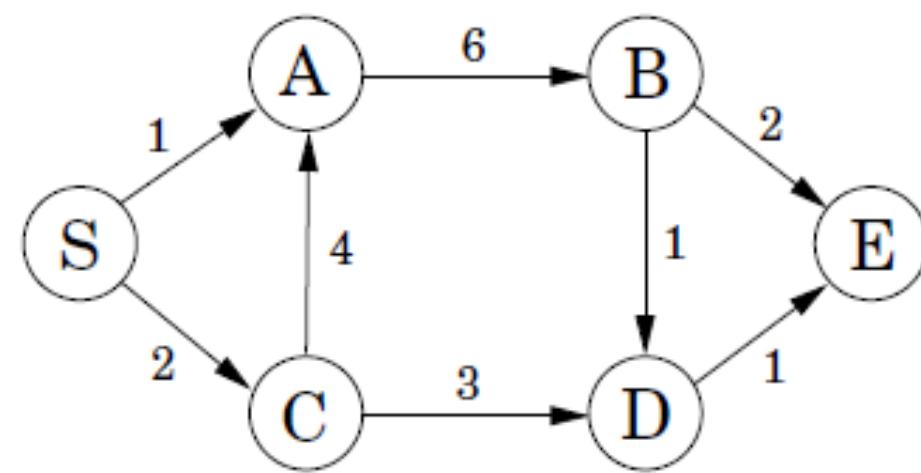
Algoritme **Floyd Warshall**

- Quina és la complexitat de l'algoritme?
 - **Complexitat :** $O(|V|^3)$

```
for i = 1 to n:  
    for j = 1 to n:  
        dist(i, j, 0) = ∞  
  
    for all (i, j) ∈ E:  
        dist(i, j, 0) = ℓ(i, j)  
    for k = 1 to n:  
        for i = 1 to n:  
            for j = 1 to n:  
                dist(i, j, k) = min{dist(i, k, k - 1) + dist(k, j, k - 1), dist(i, j, k - 1)}
```

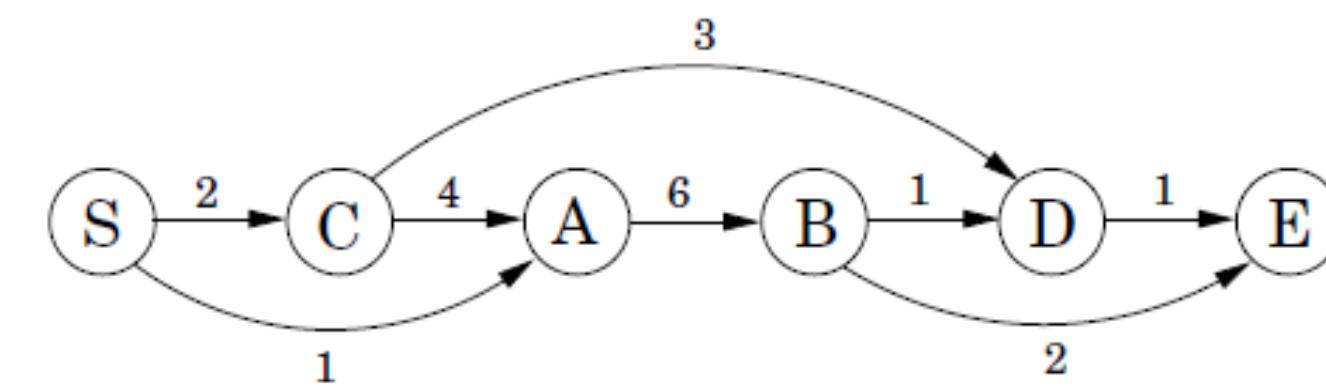
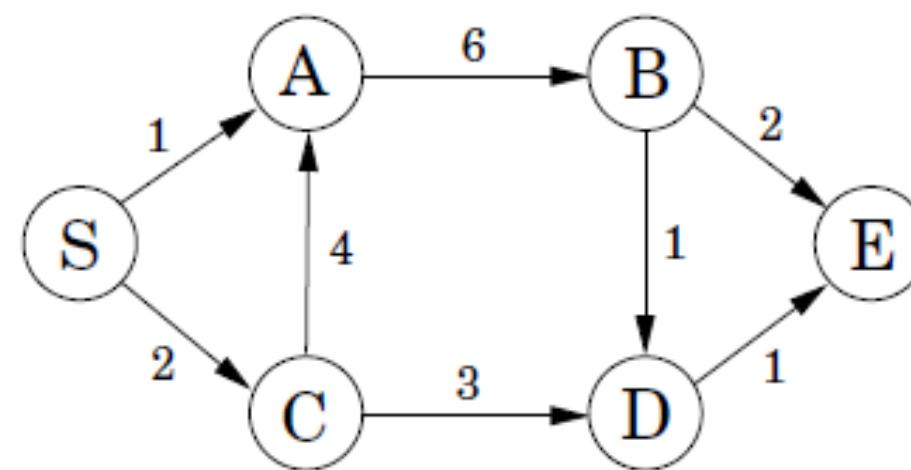
Programació Dinàmica

Linearització de grafs



Programació Dinàmica

Linearització de grafs



- Distància més curta a D:

$$\text{dist}(D) = \min\{\text{dist}(B) + 1, \text{dist}(C) + 3\}$$

Programació Dinàmica

Linearització de grafs

- Podem calcular la distància a tots els nodes en un pas:

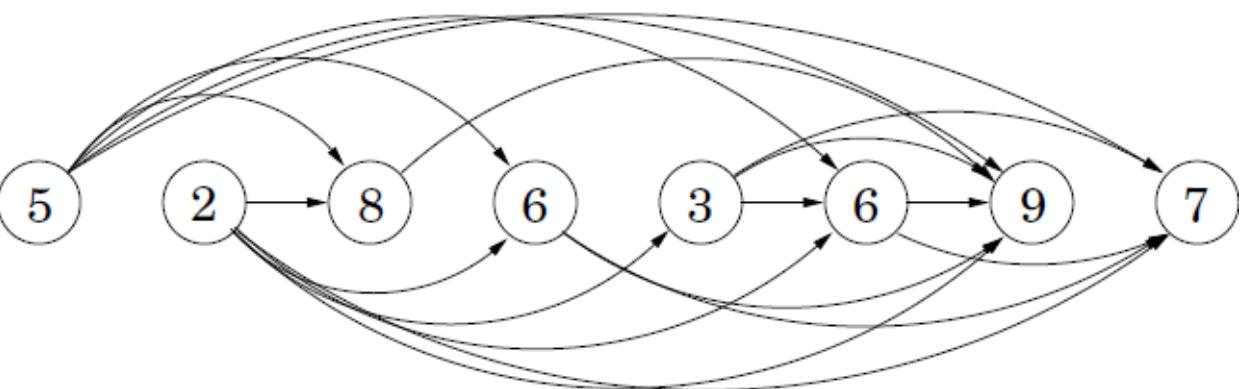
```
initialize all dist(.) values to ∞
dist(s) = 0
for each  $v \in V \setminus \{s\}$ , in linearized order:
    dist(v) = min(u,v) ∈ E {dist(u) + l(u,v)}
```

- Solucionem un conjunt de **subproblems** fins que arribem a una solució final: tècnica molt general!!
- Tenim una funció sobre els nodes anteriors per actualitzar una resposta al node actual:
 - → Aquesta funció podria ser qualsevol! (i.e. Màxim en lloc de mínim)

Programació Dinàmica

Linearització de grafs

- Per obtenir una representació de programació dinàmica, suposem que el graf disposa de totes les connexions entre un node i els seus predecessors:



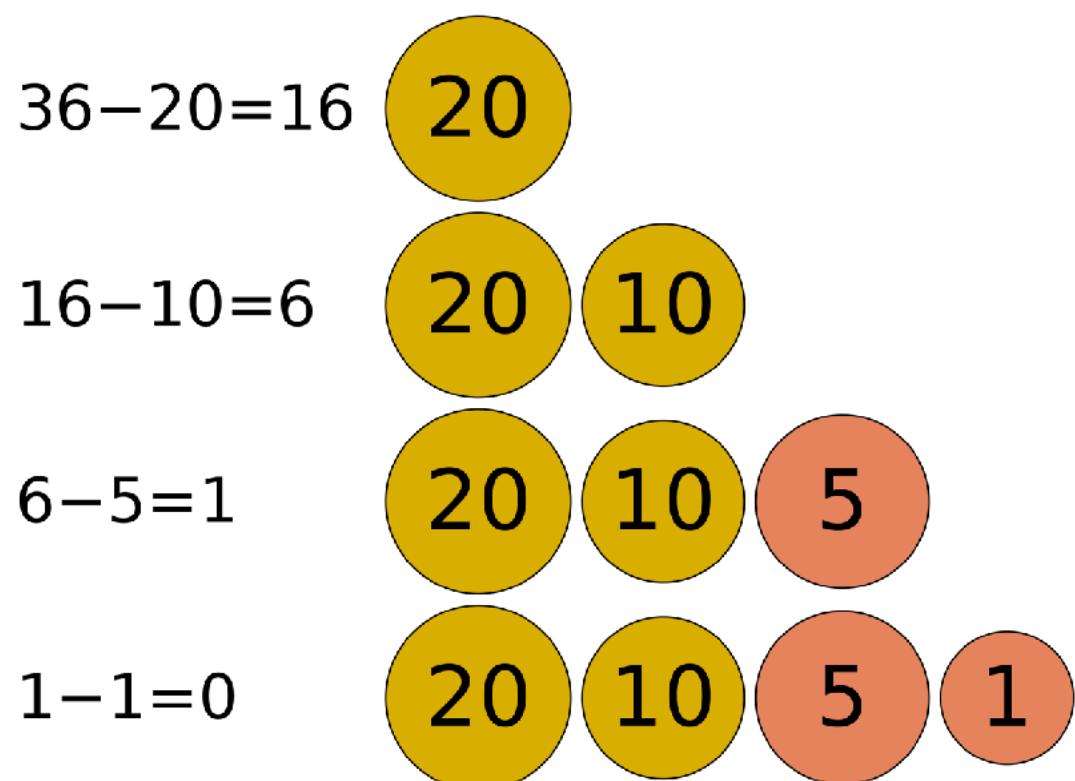
- Exemple: trobar el camí de longitud màxima

```
for j = 1, 2, ..., n:  
    L(j) = 1 + max{L(i) : (i, j) ∈ E}  
return maxj L(j)
```

Programació Dinàmica

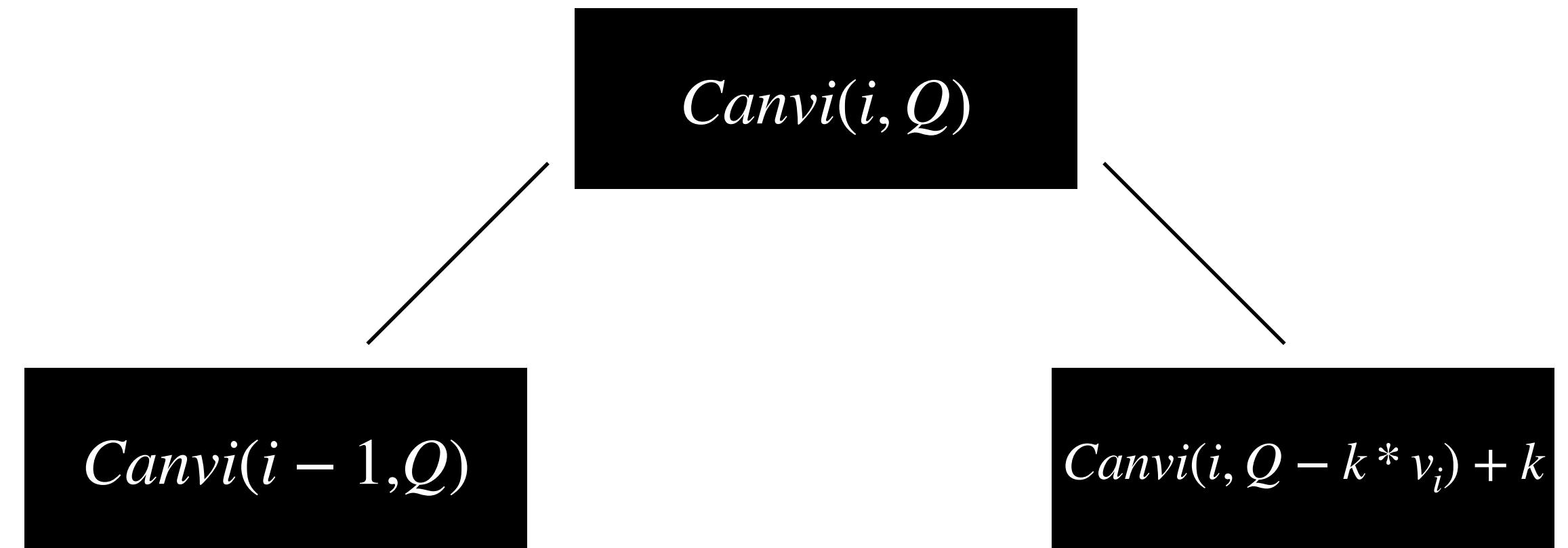
Problema canvi monedes

- **Problema:** Donat un conjunt n tipus de monedes, cadascuna d'elles amb un valor v_i , i una quantitat Q , trobar el número mínim de monedes a utilitzar per obtenir la quantitat exacta.
- Alguna proposta amb un algoritme greedy? Solució molt eficient, però no sempre l'òptima



Programació Dinàmica

Funció recurrent



Tornem la mateixa
quantitat de diners Q
sense utilitzar la moneda
amb i

Utilitzem k cops la
moneda i amb valor v_i

Programació Dinàmica

Problema canvi monedes

- **Definició de l'equació recurrent:**

- $Canvi(i, Q)$: calcula el número mínim de monedes necessàries per a retornar una quantitat Q , utilitzant els i primer tipus de monedes (és a dir, 1...i),
 - La solució de $Canvi(i, Q)$ pot utilitzar k monedes de tipus i o pot ser que no utilitzi cap.
 - Si no s'utilitza cap moneda d'aquest tipus: $Canvi(i, Q) = Canvi(i - 1, Q)$.
 - Si s'utilitza k monedes de tipus i: $Canvi(i, Q) = Canvi(i, Q - k * v_i) + k$
 - En qualsevol cas, el valor sempre serà el mínim:
 - $Canvi(i, Q) = \min_{k=0,1,\dots,Q/v_i} \{Canvi(i - 1, Q - k * v_i) + k\}$

- **Casos base:**

- Si $(i < 1) \circ (Q < 0)$ no hi ha cap solució al problema i $Canvi(i, Q) = +\inf$
- Per a qualsevol $i > 0$, $Canvi(i, 0) = 0$

Programació Dinàmica

Problema canvi monedes

- Definició de les taules utilitzades:
 - Necessitem emmagatzemar els resultats de tots els subproblemes.
 - El problema a resoldre és: $\text{Canvi}(n, Q)$
 - Necessitem una taula de $n \times Q$, d'enters, que anomenarem D , on $D[i, j] = \text{Canvi}(i, j)$
 - Exemple: $n = 3, P = 8, v = (1, 4, 6)$

		Quantitat a retornar								
		0	1	2	3	4	5	6	7	8
V1 = 1	0	1	2	3	4	5	6	7	8	
V2 = 4	0									
V3 = 6	0									

- De dalt a baix, i d'esquerra a dreta, aplicar la següent funció de recurrència:

$$\bullet D[i, j] = \min_{k=0,1,\dots,Q/v_i} \{D[i - 1, Q - k*v_i] + k\}$$

Programació Dinàmica

Problema canvi monedes

- Definició de les taules utilitzades:
 - Necessitem emmagatzemar els resultats de tots els subproblemes.
 - El problema a resoldre és: $Canvi(n, Q)$
 - Necessitem una taula de nxQ , d'enters, que anomenarem D , on $D[i, j] = Canvi(i, j)$
 - Exemple: $n = 3, Q = 8, v = (1, 4, 6)$

		Quantitat a retornar								
		0	1	2	3	4	5	6	7	8
V1 = 1		0	1	2	3	4	5	6	7	8
V2 = 4		0	1	2	3	1	2	3	4	2
V3 = 6		0	1	2	3	1	2	1	2	2

- De dalt a baix, i d'esquerra a dreta, aplicar la següent funció de recurrència:
 - $D[i, j] = \min_{k=0,1,\dots,Q/v_i} \{D[i - 1, Q - k*v_i] + k\}$



Programació Dinàmica

Exercici: Viatge pel Riu

- **El viatge més barat pel riu**

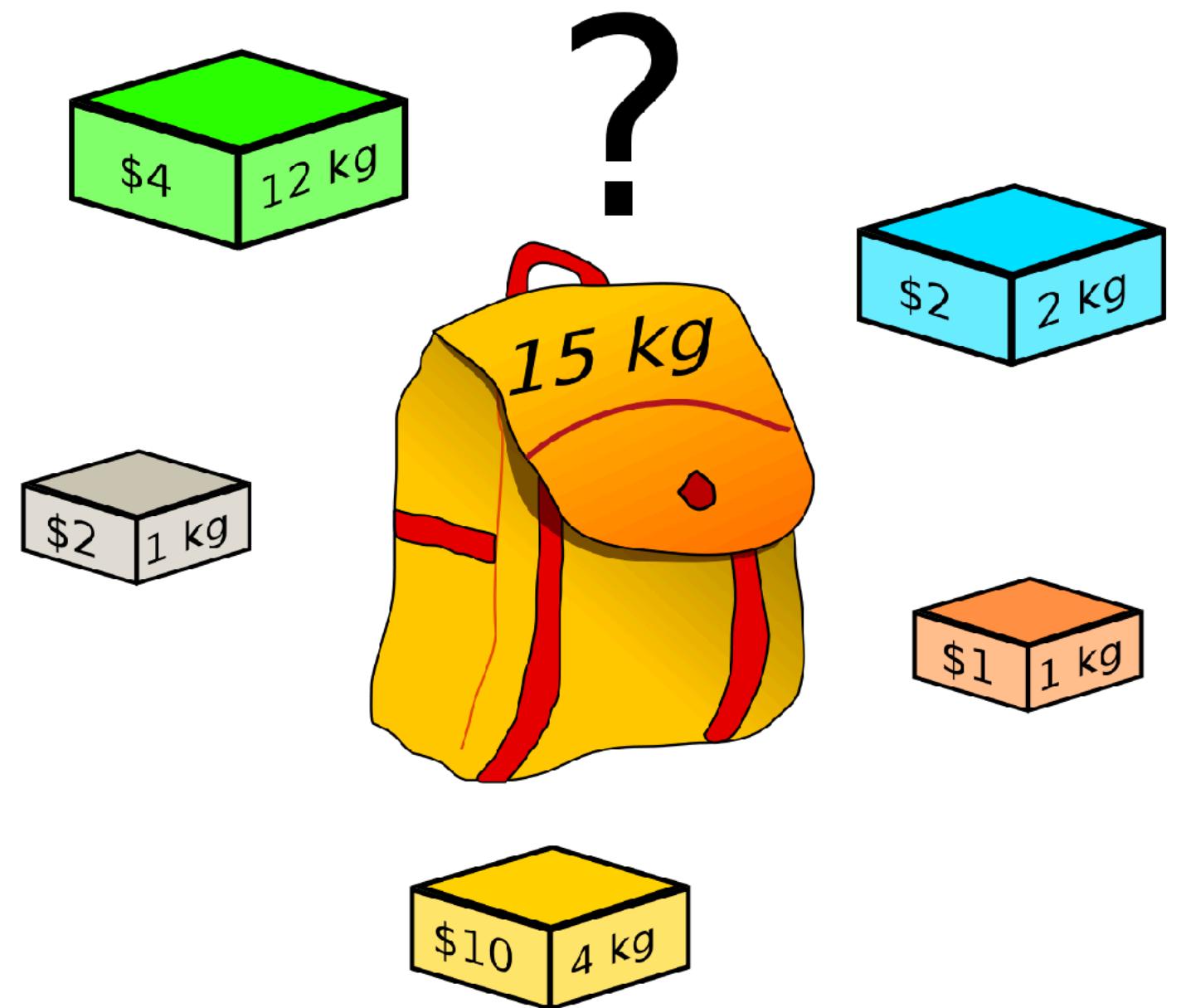
- En un riu hi ha **n** embarcadors, en cadascun d'ells es pot llogar un bot per anar a un altre embarcador situat a una zona més baixa del riu.
Suposem que no es pot remuntar el riu. Una taula de tarifes indica els costos de viatjar entre els diferents embarcadors. Se suposa que un viatge entre i i j pot sortir més barat fent diverses escales que no pas directament.
- El problema consisteix amb determinar el **cost mínim** per a anar d'un embarcador a un altre. Els costos vénen definits per la taula de tarifes, T . Així, $T[i, j]$ serà el cost d'anar de l'embarcador i a l' j . La matriu serà triangular superior d'ordre n , on n és el nombre d'embarcadors.
- La idea recursiva és que el cost es calcula de la següent manera:
 - $C(i, j) = T[i, k] + C(k, j)$
- La expressió recurrent per a la solució pot ser definida com:

$$C(i, j) = \begin{cases} 0 & \text{si } i = j \\ \min(T[i, k] + C(k, j), T[i, j]) & \text{si } i < k \leq j \end{cases}$$

- **Troba l'algoritme !!**

Programació Dinàmica

Problema de la motxilla



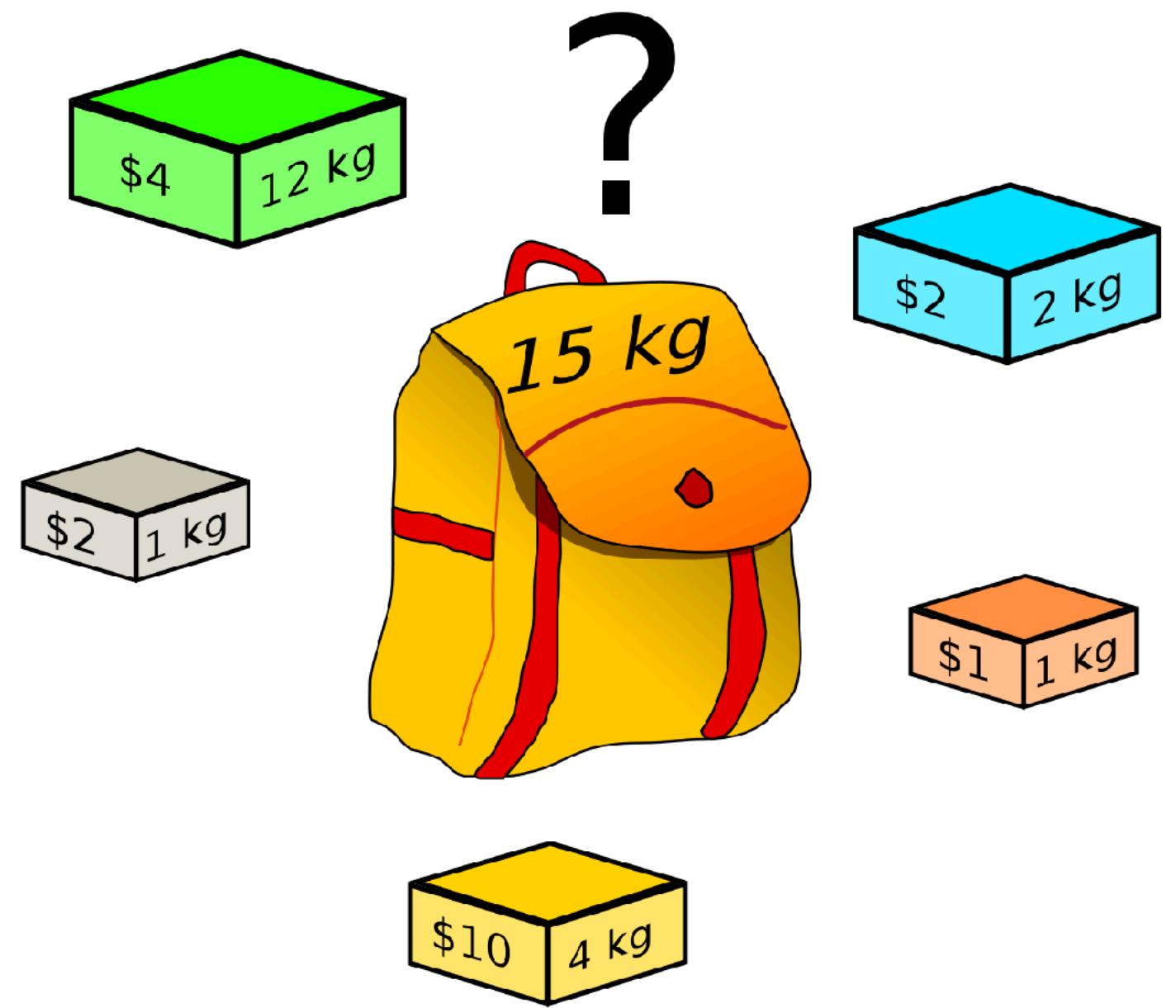
Tenim diversos objectes, amb el seu pes, i tenim la capacitat total de la motxilla. Quins objectes hauríem de posar per tal portar el màxim valor v possible amb el mínim pes w ?

$$\begin{aligned} & \text{maximize} \sum_{i=1}^n v_i x_i \\ & \text{subject to} \sum_{i=1}^n w_i x_i \leq W \text{ and } x_i \in \{0, 1\}. \end{aligned}$$

Given a set of items numbered from 1 up to n , each with a weight w_i and a value v_i , along with a maximum weight capacity W .

Programació Dinàmica

Problema de la motxilla



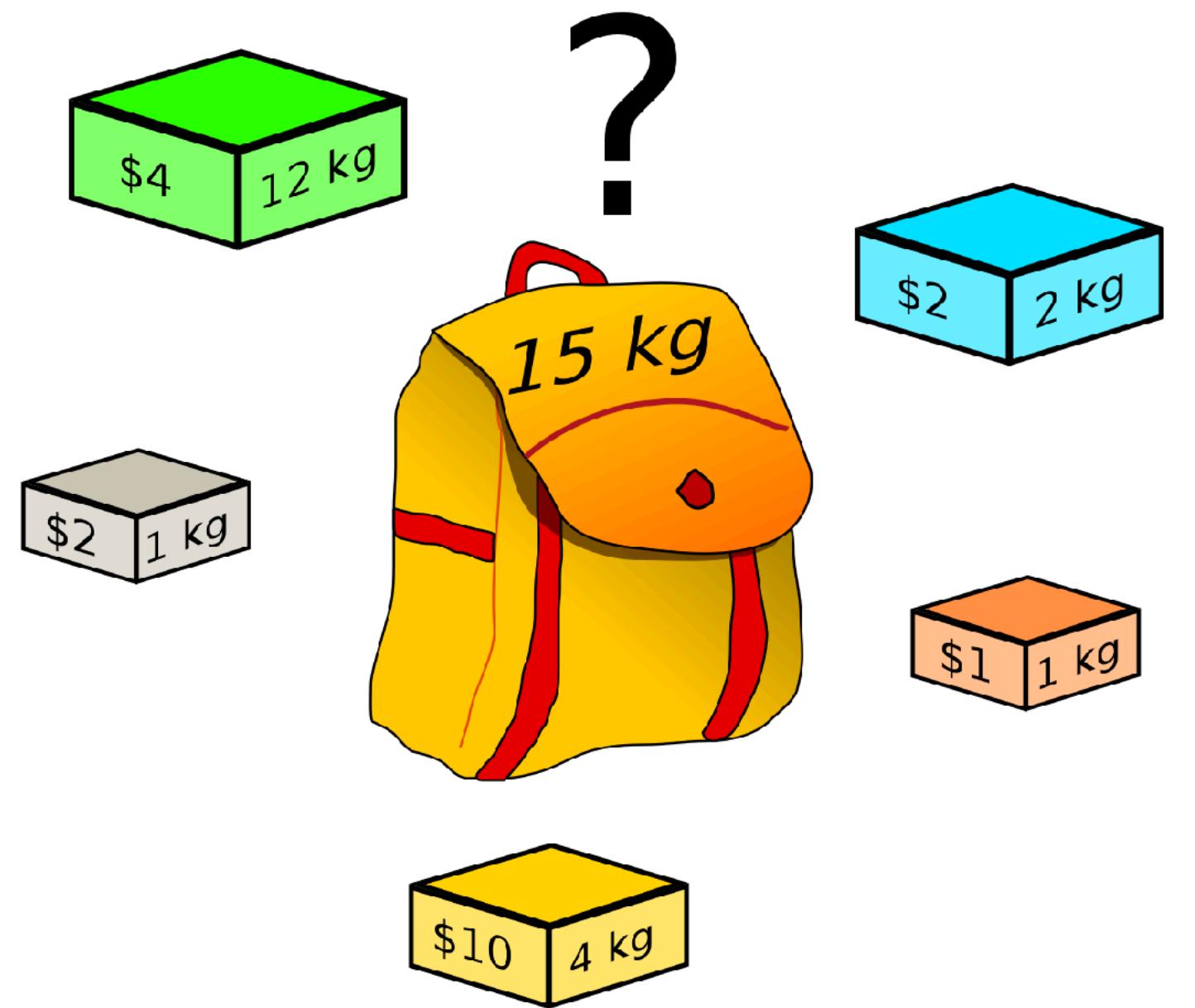
Tenim diversos objectes, amb el seu pes, i tenim la capacitat total de la motxilla. Quins objectes hauríem de posar per tal portar el màxim valor possible amb el mínim pes?

Proposem ara una solució amb un mètode:

- a) Greedy
- b) Programació dinàmica

Programació Dinàmica

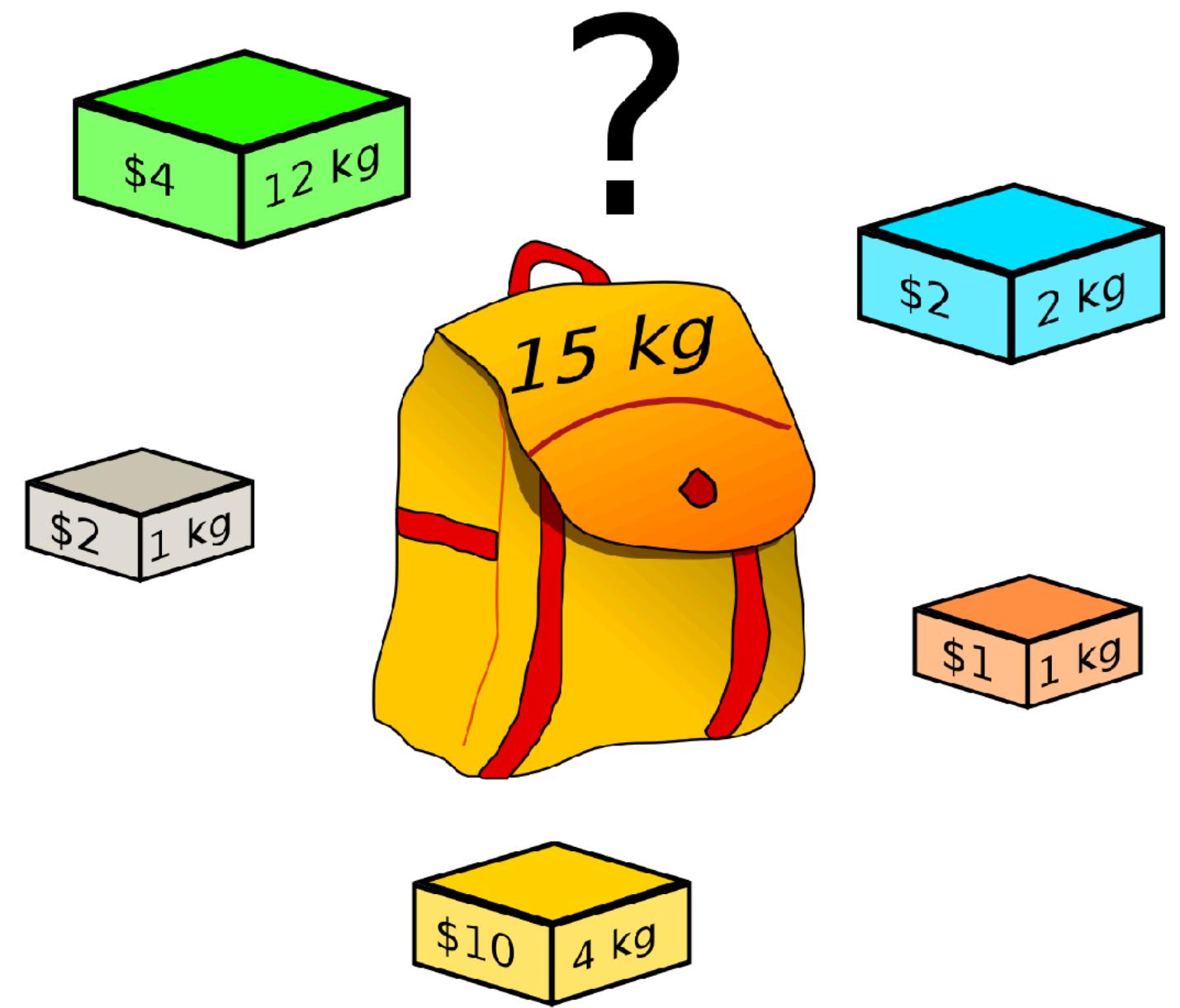
Problema de la motxilla



Solució (1) Greedy: Agafa els objectes amb un major valor fins que quedi capacitat dins la motxilla

Programació Dinàmica

Problema de la motxilla



Solució (1) Greedy: Agafa els objectes amb un major valor fins que quedi capacitat dins la motxilla

Solució (2) Greedy: Agafa els objectes on els seu valor/pes sigui major fins que quedi capacitat