

Algorísmica  
**Dividir i vèncer**  
Jordi Vitrià

## Algorismes de dividir i vèncer



«*Veni, vidi, vici*»  
Juli Cèsar

Dividir i vèncer és una estratègia de resolució de problemes consistent en:

- Dividir un problema en subproblemes que són instàncies del més petites del mateix problema.
- Resoldre **recursivament** aquests subproblemes.
- Combinar adequadament les seves solucions.

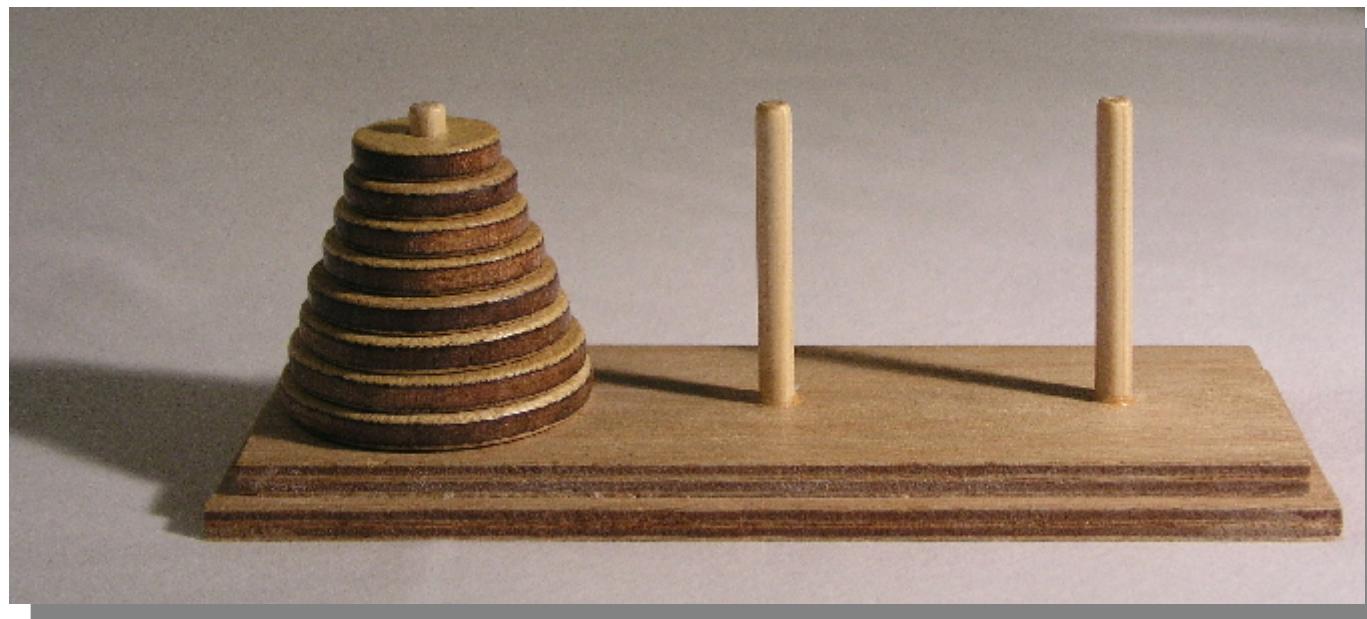
## Algorismes de dividir i vèncer

Les qüestions a resoldre són tres:

- Com dividim el problema?
- Quina és la solució al darrer pas de la recursió?
- Com combinem les solucions recursives?

There is a legend about an Indian temple which contains a large room with three time-worn posts in it surrounded by 64 golden disks. Brahmin priests, acting out the command of an ancient prophecy, have been moving these disks, in accordance with the rules of the puzzle, since that time. The puzzle is therefore also known as the Tower of Brahma puzzle. According to the legend, when the last move of the puzzle is completed, the world will end.

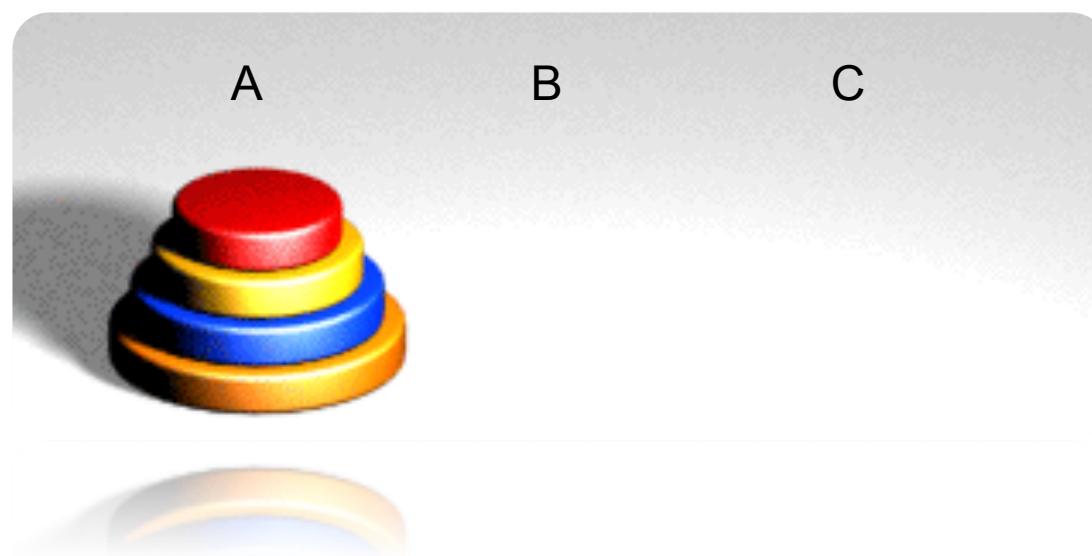
If the legend were true, and if the priests were able to move disks at a rate of one per second, using the smallest number of moves, it would take them  $2^{64}-1$  seconds or roughly 585 billion years; it would take 18,446,744,073,709,551,615 turns to finish.



## Recursivitat

Les **torres de Hanoi** és un joc usat típicament com a exemple de recursivitat.

A l'inici estan col·locats de més gran a més petit en la primera vareta. El joc consisteix en passar tots els discs a la tercera vareta tenint en compte que només es pot canviar de vareta un disc cada vegada i que mai no podem tenir un disc col·locat sobre un que sigui més petit.



A -> B  
A -> C  
B -> C  
A -> B  
C -> A  
C -> B  
A -> B  
A -> C  
B -> C  
B -> A  
C -> A  
B -> C  
A -> B  
A -> C  
B -> C

## Recursivitat

La idea bàsica és que:

- 1) Per poder passar la peça grossa de A a C cal passar les que estan a sobre de A a B amb l'ajut de C.
- 2) Llavors puc passar la peça que queda a A a C i oblidar-me d'ella, ja està ben col·locada!
- 3) Ara tinc la pila a B. Per tant el que queda és passar les peces de B a C amb l'ajuda de A i ja hauré acabat.

Aquesta idea bàsica es pot repetir recursivament!

# Recursivitat

```
def hanoi(n, a='A', b='B', c='C'):  
    if n == 0:  
        return  
    hanoi(n-1, a, c, b)  
    print a, '->', c  
    hanoi(n-1, b, a, c)
```

```
hanoi(4)
```

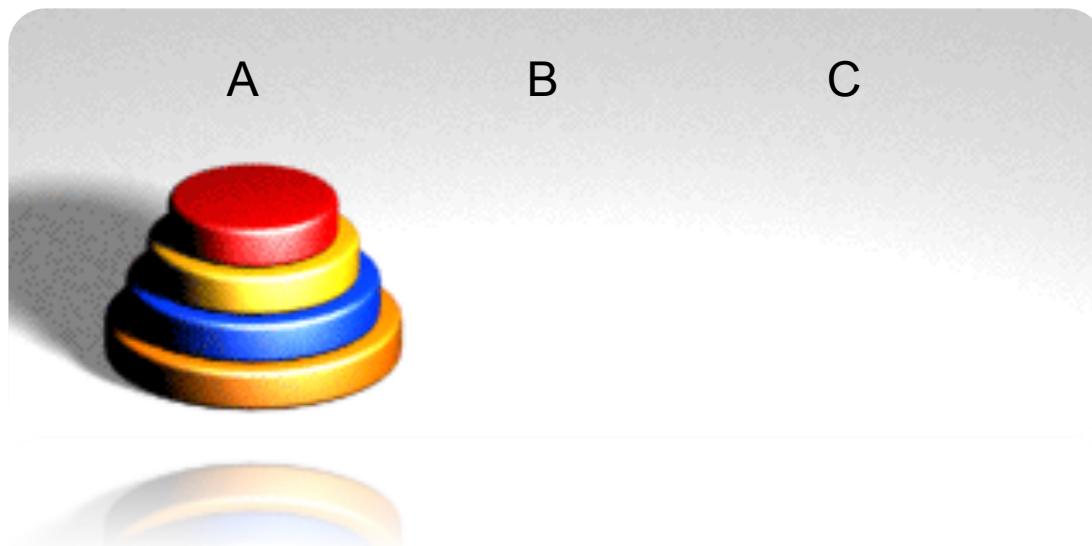
Mou els  $n$  discs que hi ha a la posició que indica  $a$  a la que indica  $c$  fent servir  $b$  d'auxiliar.

Mou els  $n-1$  discs de dalt des de  $a$  a  $b$  fent servir  $c$  d'auxiliar.

Mou el disc d' $a$  a  $c$

Mou els  $n-1$  discs de dalt des de  $b$  a  $c$  fent servir  $a$  d'auxiliar.

La complexitat és exponencial  $O(2^n)$



A -> B  
A -> C  
B -> C  
A -> B  
C -> A  
C -> B  
A -> B  
A -> C  
B -> C  
B -> A  
C -> A  
B -> C  
A -> B  
A -> C  
B -> C

D'on a on van les peces en  
aquesta versió?

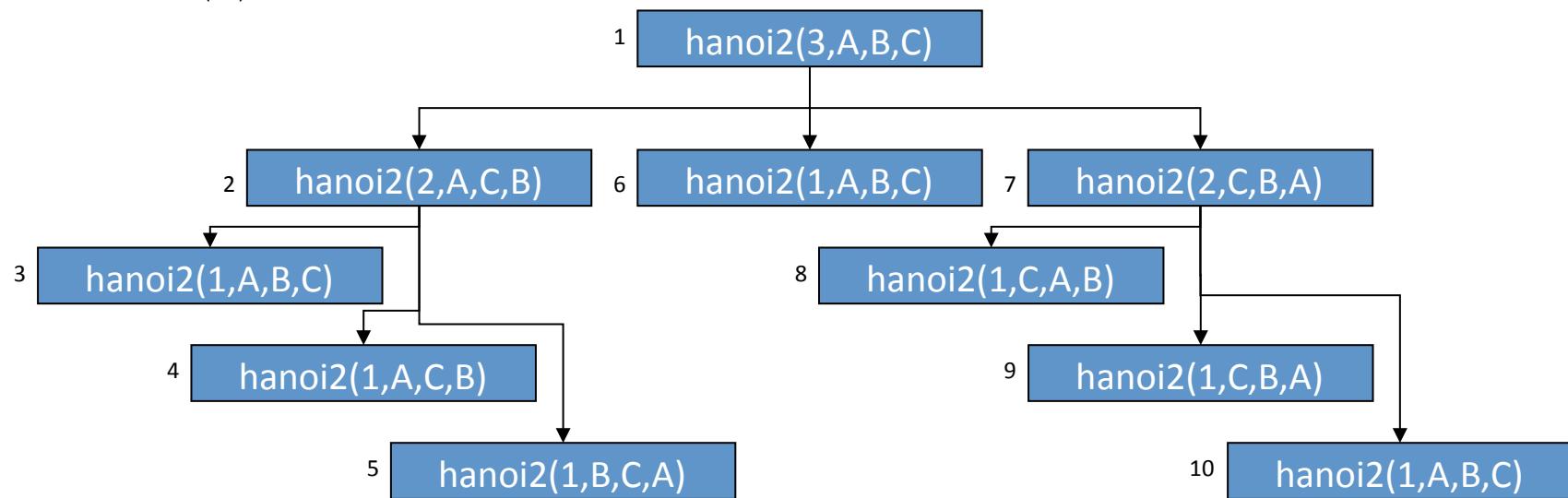
```
def hanoi2(n,source="A",spare="B",dest="C"):  
    if n==1: print source+"->"+spare  
    else:  
        hanoi2(n-1, source, dest, spare)  
        hanoi2(1, source, spare, dest)  
        hanoi2(n-1,dest, spare, source)  
  
hanoi2(3)
```

```

def hanoi2(n,source="A",spare="B",dest="C") :
    if n==1: print source+"->"+spare
    else:
        hanoi2(n-1, source, dest, spare)
        hanoi2(1, source, spare,dest)
        hanoi2(n-1,dest, spare, source)

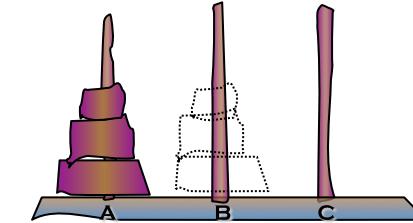
```

hanoi2 (3)



The initial call 1 is made, and `solveTowers` begins execution:

```
count = 3
source = A
dest = B
spare = C
```

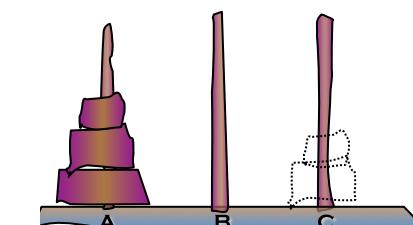


At point X, recursive call 2 is made, and the new invocation of the method begins execution:

```
count = 3
source = A
dest = B
spare = C
```

X

```
count = 2
source = A
dest = C
spare = B
```



At point X, recursive call 3 is made, and the new invocation of the method begins execution:

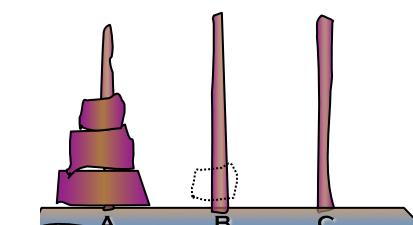
```
count = 3
source = A
dest = B
spare = C
```

X

```
count = 2
source = A
dest = C
spare = B
```

X

```
count = 1
source = A
dest = B
spare = C
```



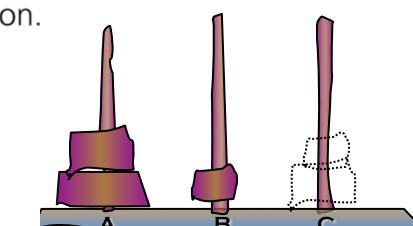
This is the base case, so a disk is moved, the return is made, and the method continues execution.

```
count = 3
source = A
dest = B
spare = C
```

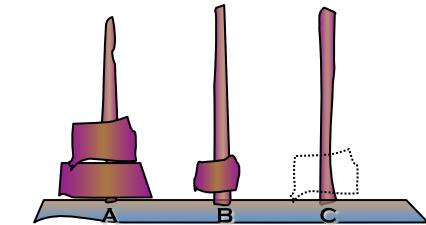
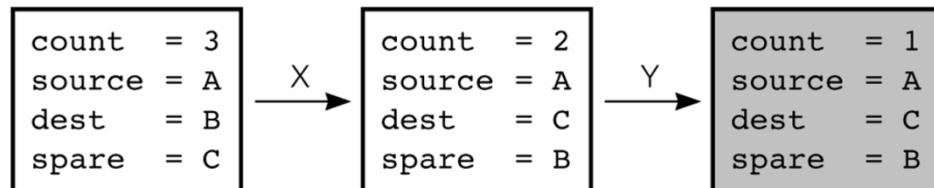
X

```
count = 2
source = A
dest = C
spare = B
```

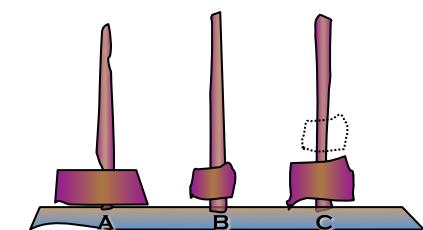
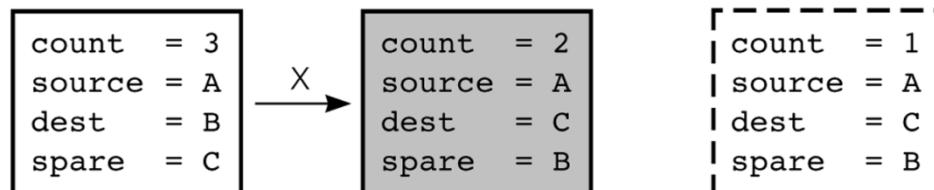
```
-----|
| count = 1
| source = A
| dest = B
| spare = C
|-----|
```



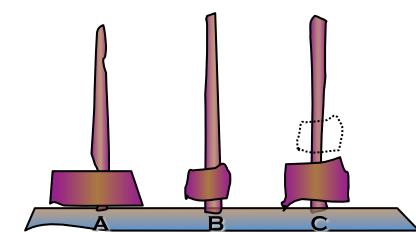
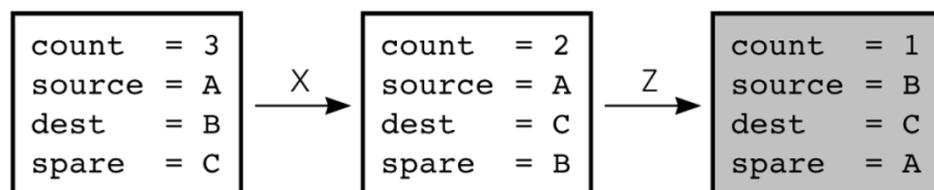
At point Y, recursive call 4 is made, and the new invocation of the method begins execution:



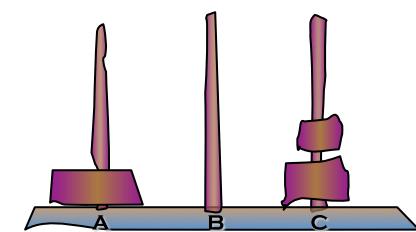
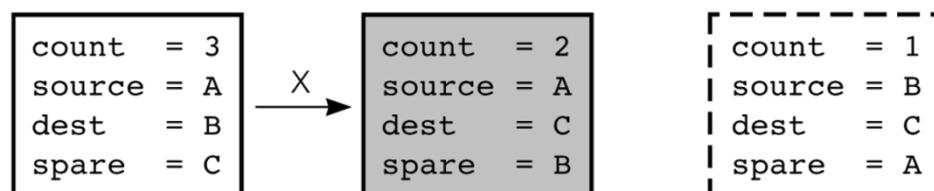
This is the base case, so a disk is moved, the return is made, and the method continues execution.



At point Z, recursive call 5 is made, and the new invocation of the method begins execution:



This is the base case, so a disk is moved, the return is made, and the method continues execution.

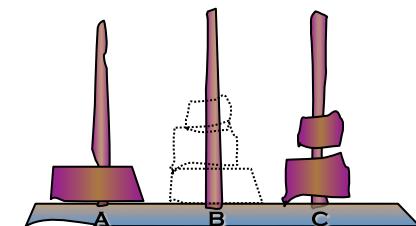


This invocation completes, the return is made, and the method continues execution.

```
count = 3  
source = A  
dest = B  
spare = C
```

```
count = 2  
source = A  
dest = C  
spare = B
```

```
count = 1  
source = B  
dest = C  
spare = A
```

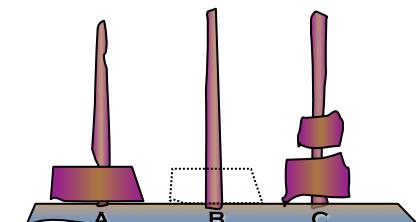


At point Y, recursive call 6 is made, and the new invocation of the method begins execution:

```
count = 3  
source = A  
dest = B  
spare = C
```

```
count = 1  
source = A  
dest = B  
spare = C
```

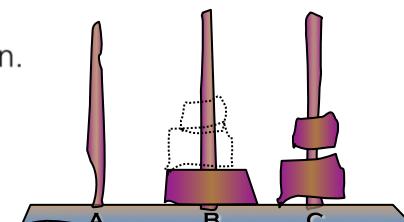
Y



This is the base case, so a disk is moved, the return is made, and the method continues execution.

```
count = 3  
source = A  
dest = B  
spare = C
```

```
count = 1  
source = A  
dest = B  
spare = C
```

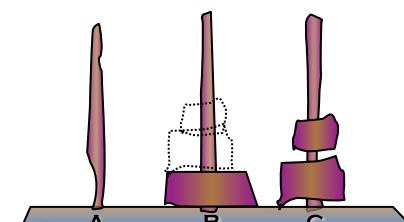


At point Z, recursive call 7 is made, and the new invocation of the method begins execution:

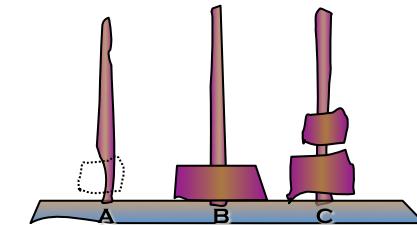
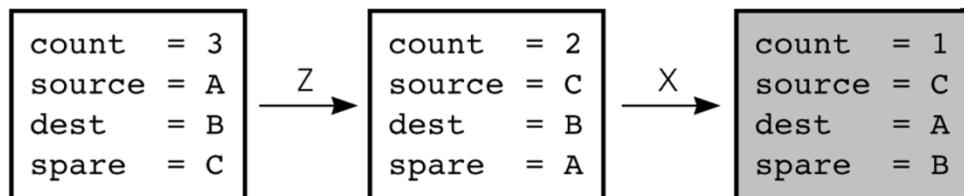
```
count = 3  
source = A  
dest = B  
spare = C
```

```
count = 2  
source = C  
dest = B  
spare = A
```

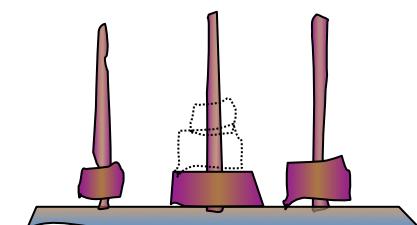
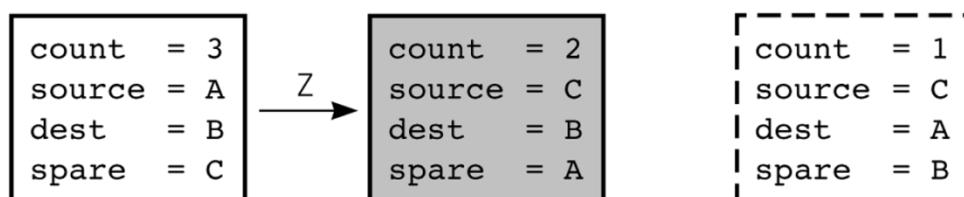
Z



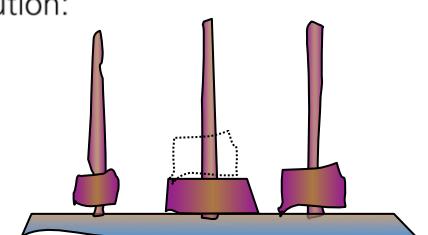
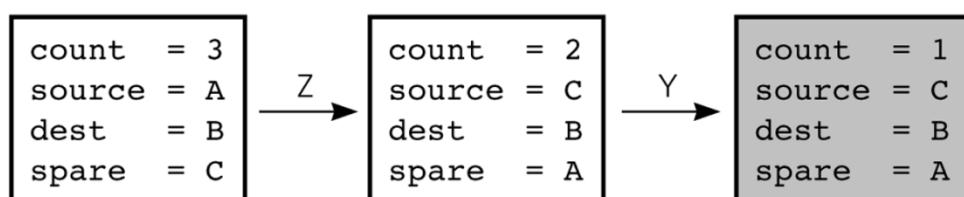
At point X, recursive call 8 is made, and the new invocation of the method begins execution:



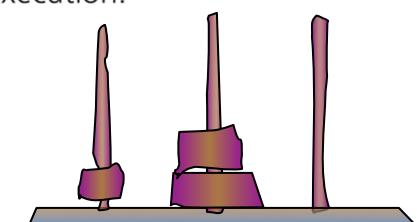
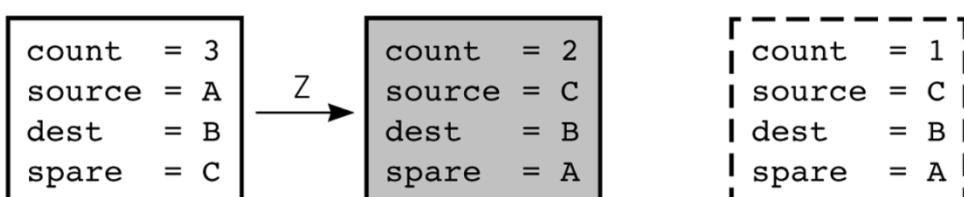
This is the base case, so a disk is moved, the return is made, and the method continues execution.



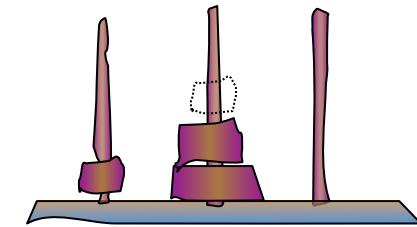
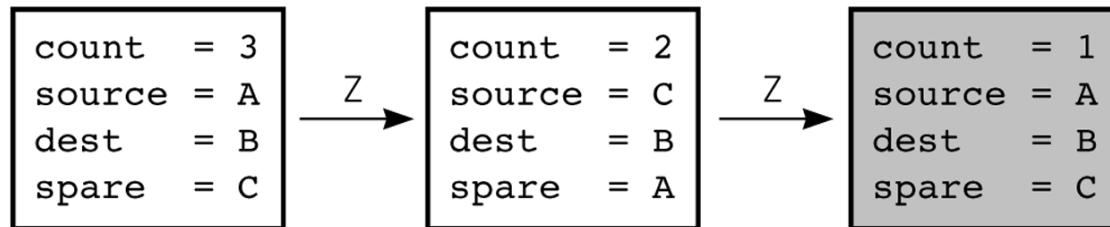
At point Y, recursive call 9 is made, and the new invocation of the method begins execution:



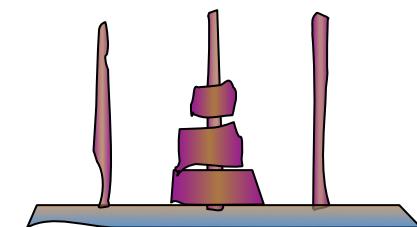
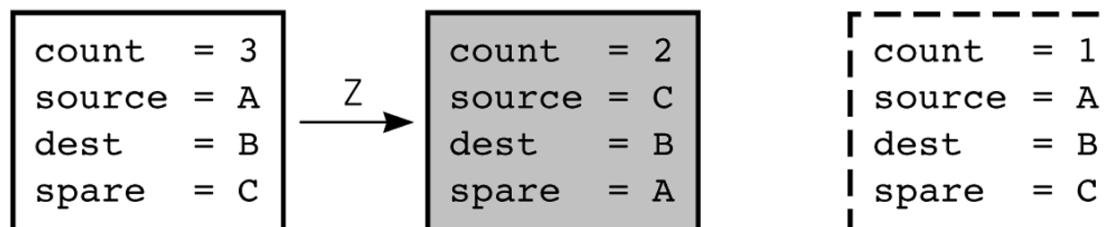
This is the base case, so a disk is moved, the return is made, and the method continues execution.



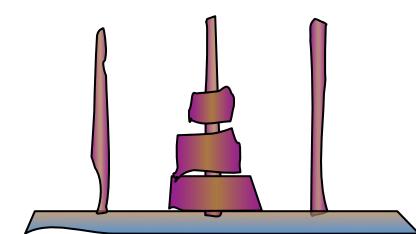
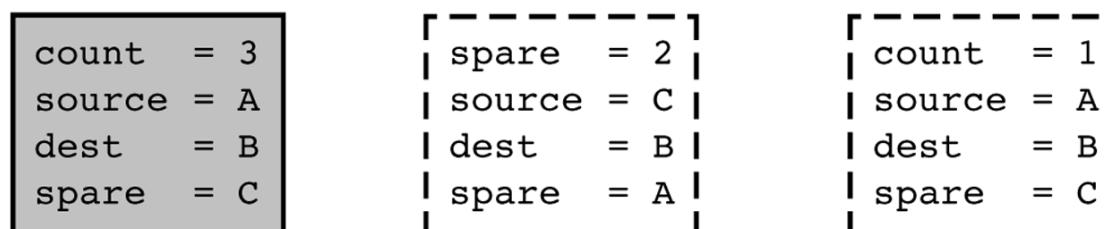
At point Z, recursive call 10 is made, and the new invocation of the method begins execution:



This is the base case, so a disk is moved, the return is made, and the method continues execution.



This invocation completes, the return is made, and the method continues execution.



## Algorismes de dividir i vèncer: relacions de recurrència.

L'esquema general (no per tots els casos!) d'aquests algorismes és: tenim un problema de mida  $n$ , que reformulem mitjançant la solució d' $a$  problemes de mida  $n/b$  i llavors combinem les respostes en un temps  $O(n^d)$ .

La seva complexitat serà per tant  $T(n)=aT(n/b)+O(n^d)$

Aquesta recurrència té una solució tancada, que està enunciada al Teorema Master.

## Algorismes de dividir i vèncer: relacions de recurrència.

### Teorema Master:

Si  $T(n)=aT(n/b)+O(n^d)$  per algunes constants  $a>0$ ,  $b>1$ ,  
i  $d\geq 0$ , llavors:

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a . \end{cases}$$

## Algorismes de dividir i vèncer: relacions de recurrència.

**Exemple:**  $T(n) = 2T\left(\frac{n}{2}\right) + n^2$

Les variables són:  $a=2$ ,  $b=2$ ,  $d=2$ ,  $\log_b a = \log_2 2 = 1$ .

Per tant s'aplica el cas (1) del teorema Master:

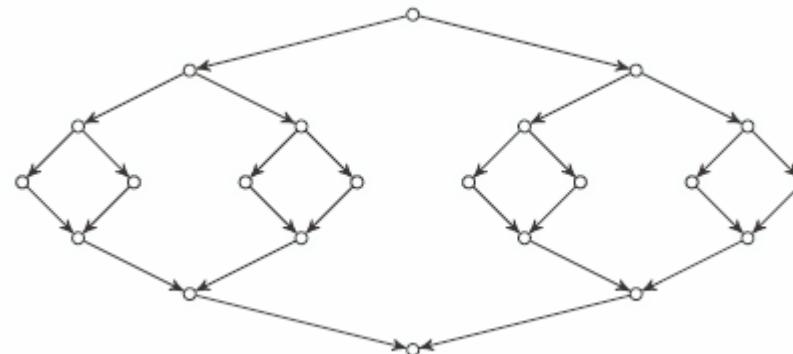
$$d > \log_b a, \quad 2 > \log_2 2 = 1$$

i per tant tenim  $O(n^2)$ .

## Algorismes de dividir i vèncer: esquema general.

*Listing 6-1. A General Implementation of the Divide and Conquer Scheme*

```
def divide_and_conquer(S, divide, combine):
    if len(S) == 1: return S
    L, R = divide(S)
    A = divide_and_conquer(L, divide, combine)
    B = divide_and_conquer(R, divide, combine)
    return combine(A, B)
```



## Algorismes de dividir-i-vèncer: mergesort

L'ordenació d'una llista es pot plantejar d'aquesta manera!

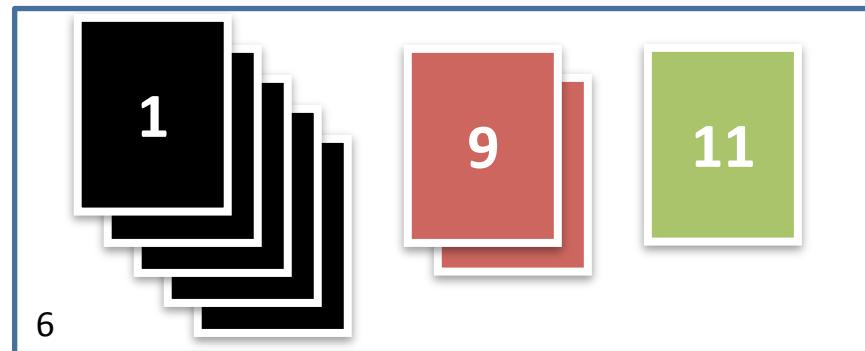
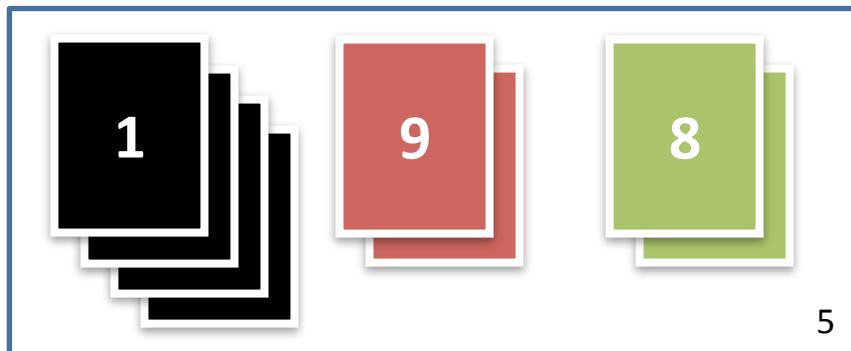
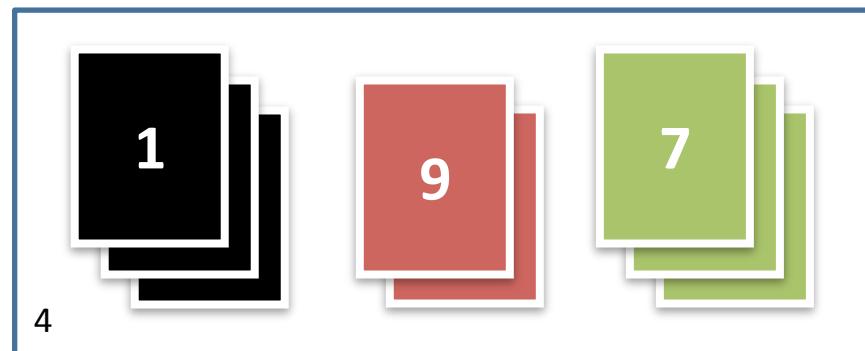
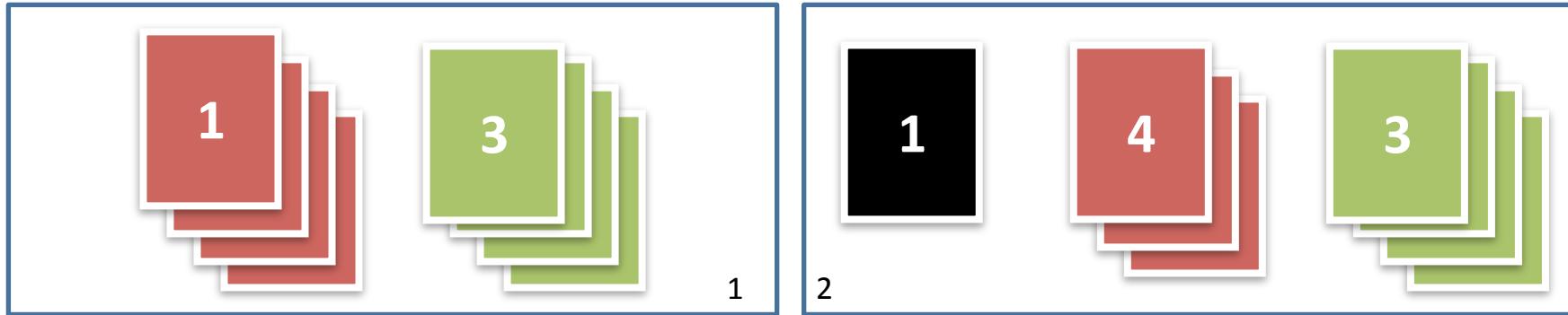
Idea de recursió!

Suposem que tenim dos conjunts de cartes, amb  $n$  elements cada un, que estan ordenats de menor a major (quan les posem de cara la més petita està al davant).

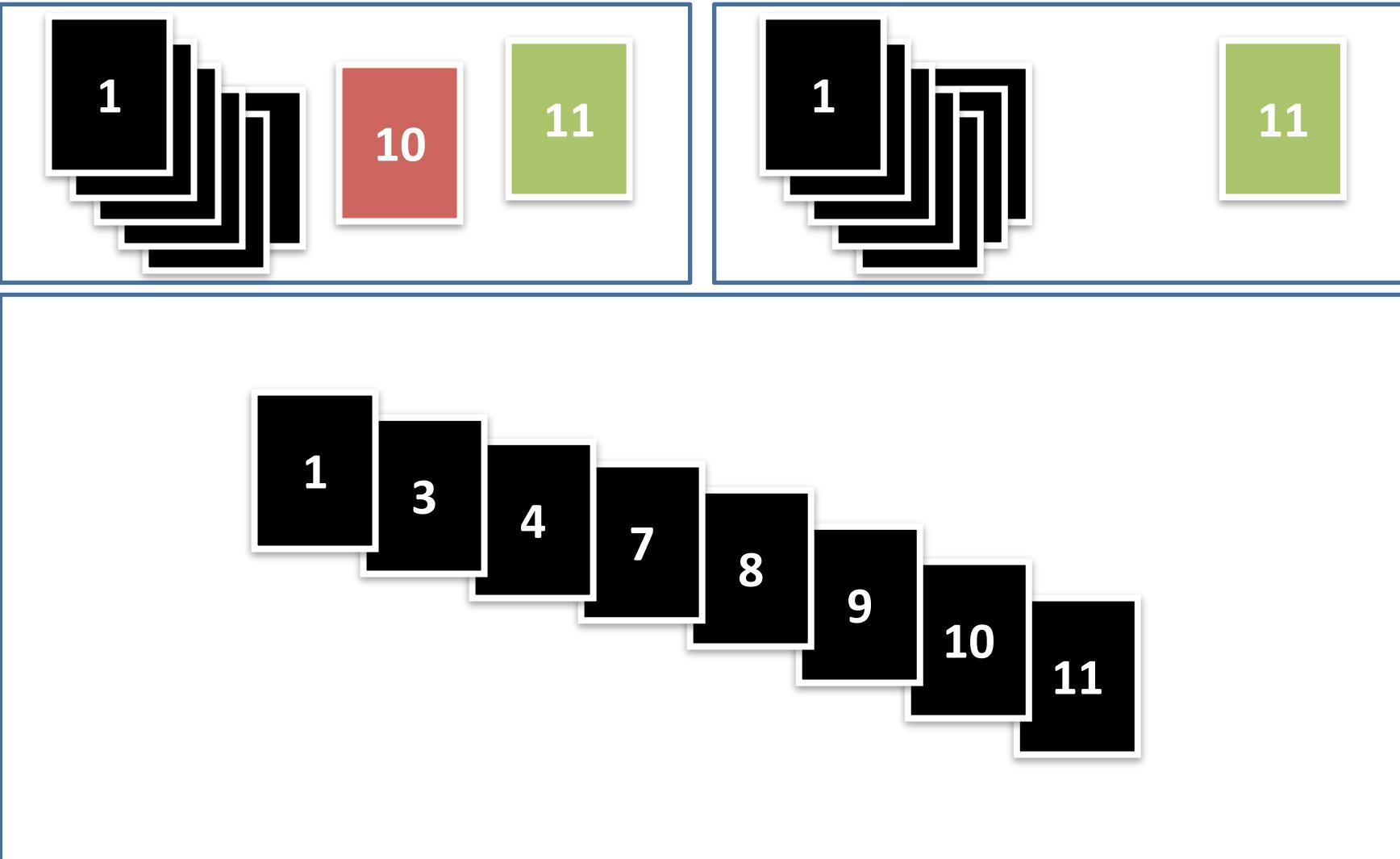
1. Comparem les dues cartes de sobre de tot de cada conjunt i escollim la més petita, que posem a un nou conjunt de cartes ordenades (pel darrera, si n'hi ha alguna).
2. Repetim 1 fins que un dels conjunts estigui buit, i l'altre l'afegim per darrera al conjunt de cartes ordenades.

El resultat és un conjunt de cartes ordenades!

## Algorismes de dividir-i-vèncer: mergesort



## Algorismes de dividir-i-vèncer: mergesort



## Algorismes de dividir-i-vèncer: mergesort

Això ho podem expressar recursivament:

```
function mergesort(a[1...n])
Input: An array of numbers a[1...n]
Output: A sorted version of this array

if n > 1:
    return merge(mergesort(a[1...[n/2]]), mergesort(a[[n/2] + 1...n]))
else:
    return a
```

La correcció d'aquest algorisme és evident, sempre i quan definim bé la funció merge.

## Algorismes de dividir-i-vèncer: mergesort

La funció merge també la podem definir recursivament!

```
function merge(x[1...k], y[1...l])
if k = 0: return y[1...l]
if l = 0: return x[1...k]
if x[1] ≤ y[1]:
    return x[1] o merge(x[2...k], y[1...l])
else:
    return y[1] o merge(x[1...k], y[2...l])
```

Aquesta funció es pot definir també de forma NO recursiva, i és simplement anar comparant i copiant de forma ordenada els dos vectors en un nou vector.

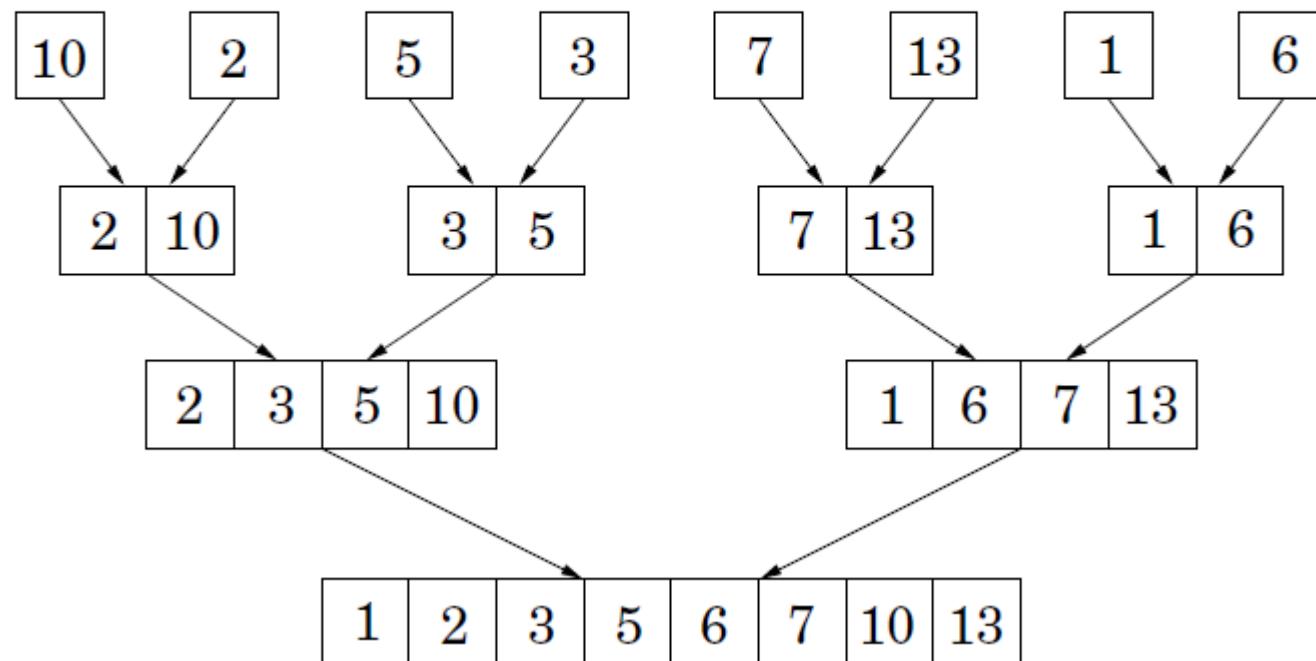
L'operador o significa concatenació.

## Algorismes de dividir-i-vèncer: mergesort

La seqüència de merge en un cas concret:

Input: 

10	2	5	3	7	13	1	6
----	---	---	---	---	----	---	---



## Algorismes de dividir-i-vèncer: mergesort

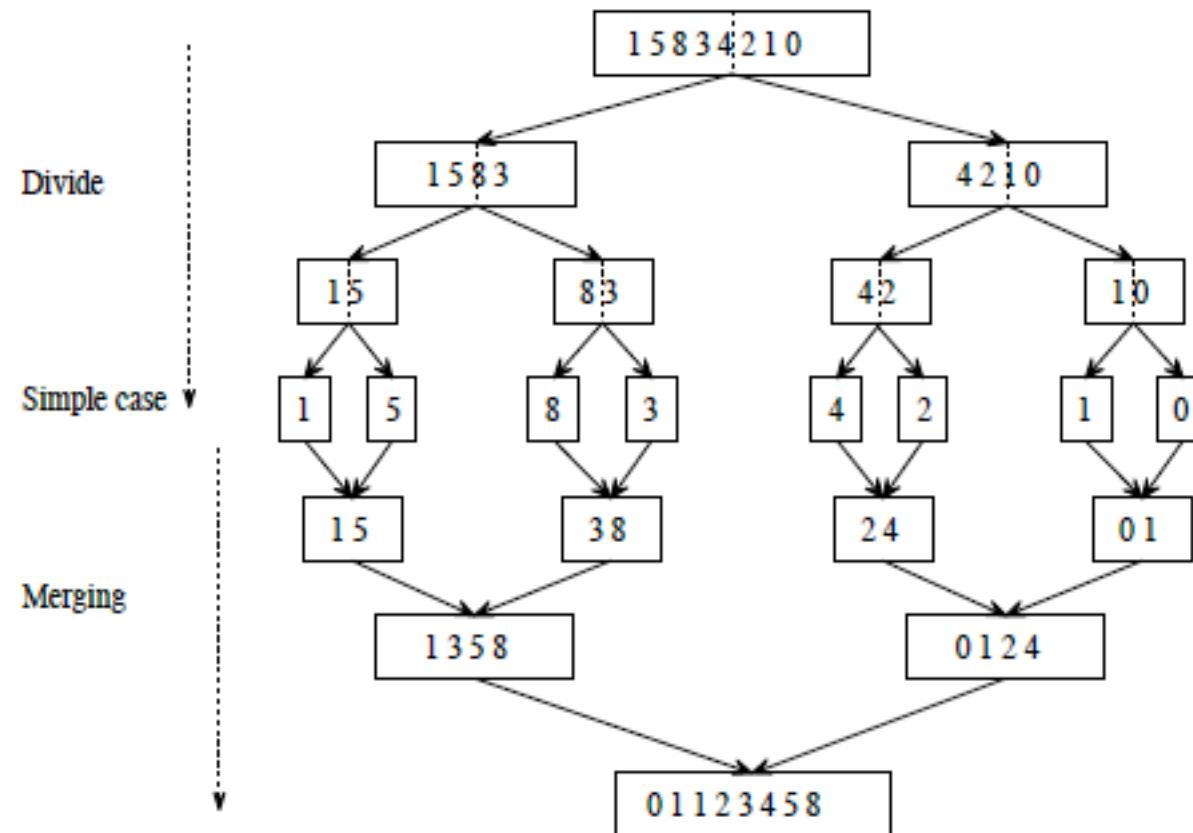
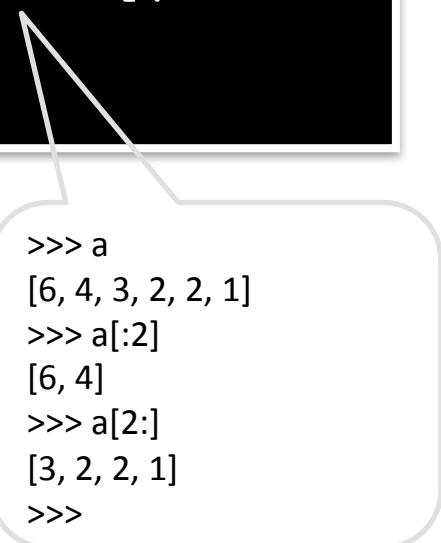


Figure 1: Mergesort

## Algorismes de dividir-i-vèncer: mergesort

```
def mergesort(list):
    if len(list) < 2:
        return list
    else:
        middle = len(list) / 2
        left = mergesort(list[:middle])
        right = mergesort(list[middle:])
        return merge(left, right)
```



```
>>> a
[6, 4, 3, 2, 2, 1]
>>> a[:2]
[6, 4]
>>> a[2:]
[3, 2, 2, 1]
>>>
```

## Algorismes de dividir-i-vèncer: mergesort

```
def merge(left, right):
    result = []
    i ,j = 0, 0
    while(i < len(left) and j < len(right)):
        if (left[i] <= right[j]):
            result.append(left[i])
            i = i + 1
        else:
            result.append(right[j])
            j = j + 1

    result += left[i:]
    result += right[j:]
    return result
```

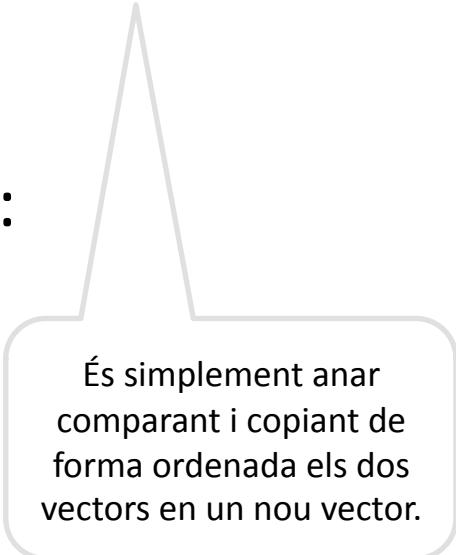
Versió no recursiva

## Algorismes de dividir-i-vèncer: mergesort

La funció merge té una complexitat per cada crida recursiva  $O(n)$  (en el pitjor dels casos).

Per tant, mergesort té una complexitat:

$$T(n)=2T(n/2)+O(n).$$



És simplement anar comparant i copiant de forma ordenada els dos vectors en un nou vector.

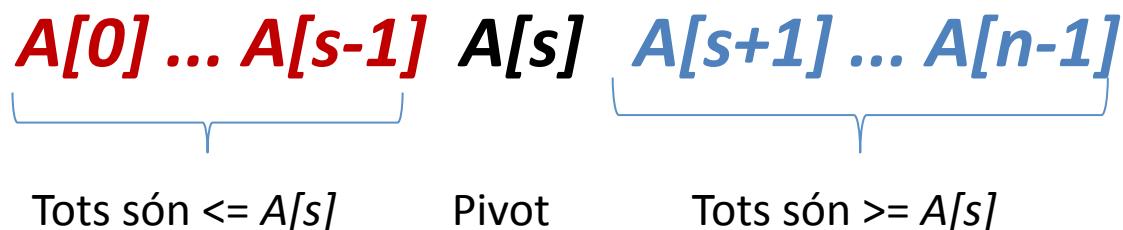
O el que és el mateix,  **$O(n \log n)$** .

(La funció mergesort es pot implementar iterativament, però per fer-ho necessitem una estructura que encara no coneixem: la cua).

## Algorismes de dividir-i-vèncer: **quicksort**

**Quicksort** és un altre algorisme d'ordenació basat en l'estrategia de dividir i vèncer.

Aquest algorisme divideix el vector basant-se en els valors: reordena els valors de  $A[0, \dots, n-1]$  per aconseguir una **partició**, una situació en la que tots els elements anteriors a una posició  $s$  siguin menors o iguals que  $A[s]$  i els de després més grans o iguals:



Idea de recursió!

## Algorismes de dividir-i-vèncer: **quicksort**

Òbviament, si tenim aquesta situació  $A[s]$  ja està al seu lloc i no s'haurà de moure, i podem passar a ordenar el que hi ha a ambdues bandes.

```
def quick_sort(A):
    quick_sort_r(A, 0, len(A) - 1)

def quick_sort_r(A , first, last):
    if last > first:
        pivot = partition(A, first, last)
        quick_sort_r(A, first, pivot - 1)
        quick_sort_r(A, pivot + 1, last)
```

## Algorismes de dividir-i-vèncer: quicksort

Com **calculem la partició** d'una llista  $A$ ?

Primer seleccionem un element, respecte del qual dividirem la subllista, que anomenarem el **pivot**. Per exemple, escollim  $\text{pivot} = A[\text{first}]$ .

Després hem de reordenar per aconseguir una partició. Això ho podem fer amb dues passades (d'esquerra a dreta i de dreta a esquerra) de la llista.

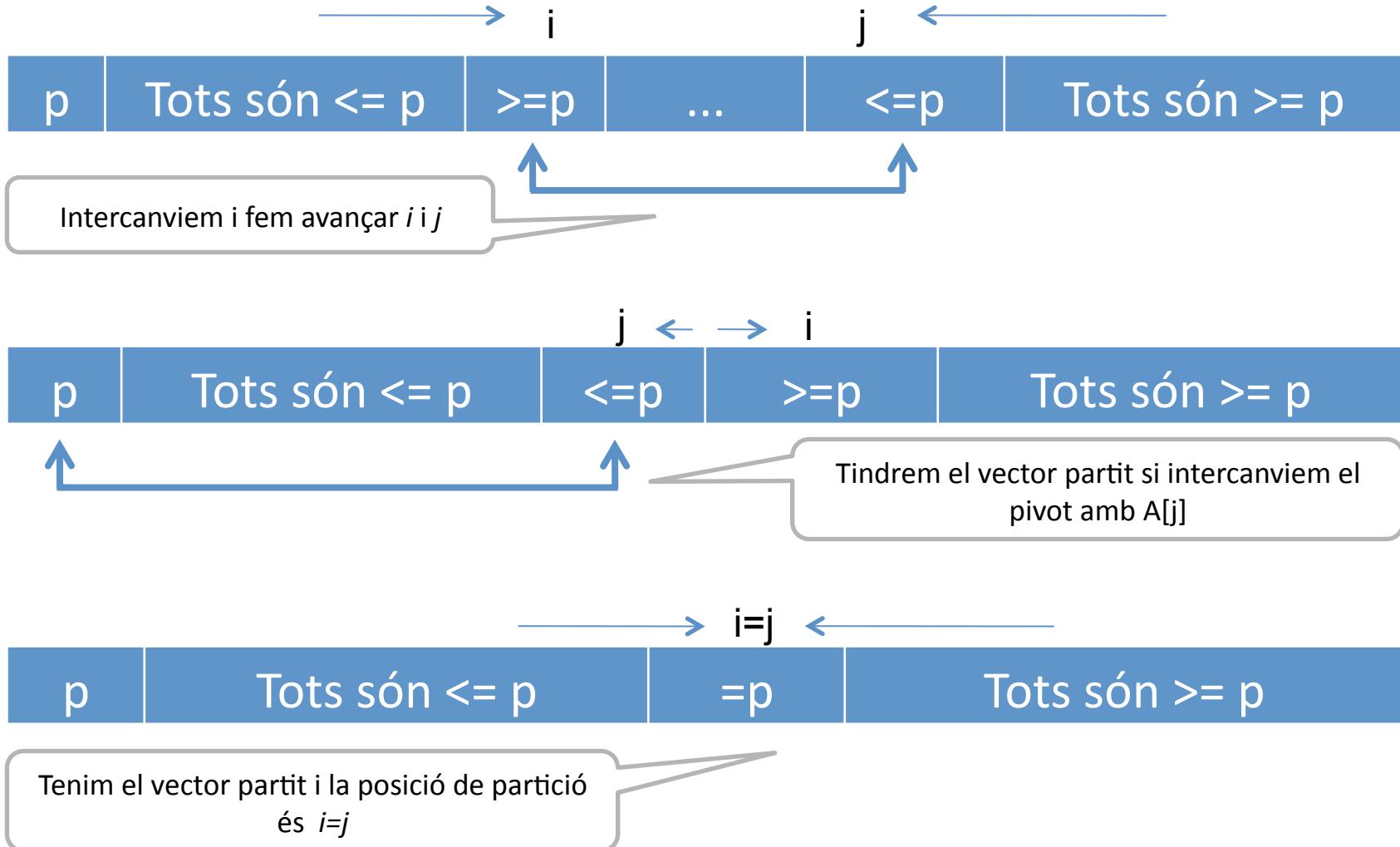
## Algorismes de dividir-i-vèncer: **quicksort**

La passada d'esquerra a dreta ( $i$ ) comença pel segon element i no s'atura fins trobar un element més gran o igual que el pivot ( $p$ ).

La passada de dreta a esquerra ( $j$ ) comença per l'últim element i s'atura quan troba un element més petit o igual que el pivot.

Quan les dues passades s'aturen, ens podem trobar en tres situacions:

## Algorismes de dividir-i-vèncer: quicksort



## Algorismes de dividir-i-vèncer: quicksort

```
def partition(A, first, last):
    sred = (first + last)/2
    if A[first] > A[sred]: A[first], A[sred] = A[sred], A[first]
    if A[first] > A[last]: A[first], A[last] = A[last], A[first]
    if A[sred] > A[last]: A[sred], A[last] = A[last], A[sred]
    A[sred], A[first] = A[first], A[sred]
    pivot = first
    i = first + 1
    j = last

    while True:
        while i <= last and A[i] <= A[pivot]: i += 1
        while j >= first and A[j] > A[pivot]: j -= 1
        if i >= j: break
        else:
            A[i], A[j] = A[j], A[i]
    A[j], A[pivot] = A[pivot], A[j]
    return j
```

## Algorismes de dividir-i-vèncer: quicksort

0	1	2	3	4	5	6	7
---	---	---	---	---	---	---	---

5	3	1	9	8	2	4	7
	i						j

5	3	1	9	8	2	4	7
			i			j	

5	3	1	4	8	2	9	7
			i			j	

5	3	1	4	8	2	9	7
				i	j		

5	3	1	4	2	8	9	7
				i	j		

## Algorismes de dividir-i-vèncer: quicksort

0	1	2	3	4	5	6	7
---	---	---	---	---	---	---	---

5	3	1	4	2	8	9	7
				j	i		

2	3	1	4	5	8	9	7

2	3	1	4				
	i		j				

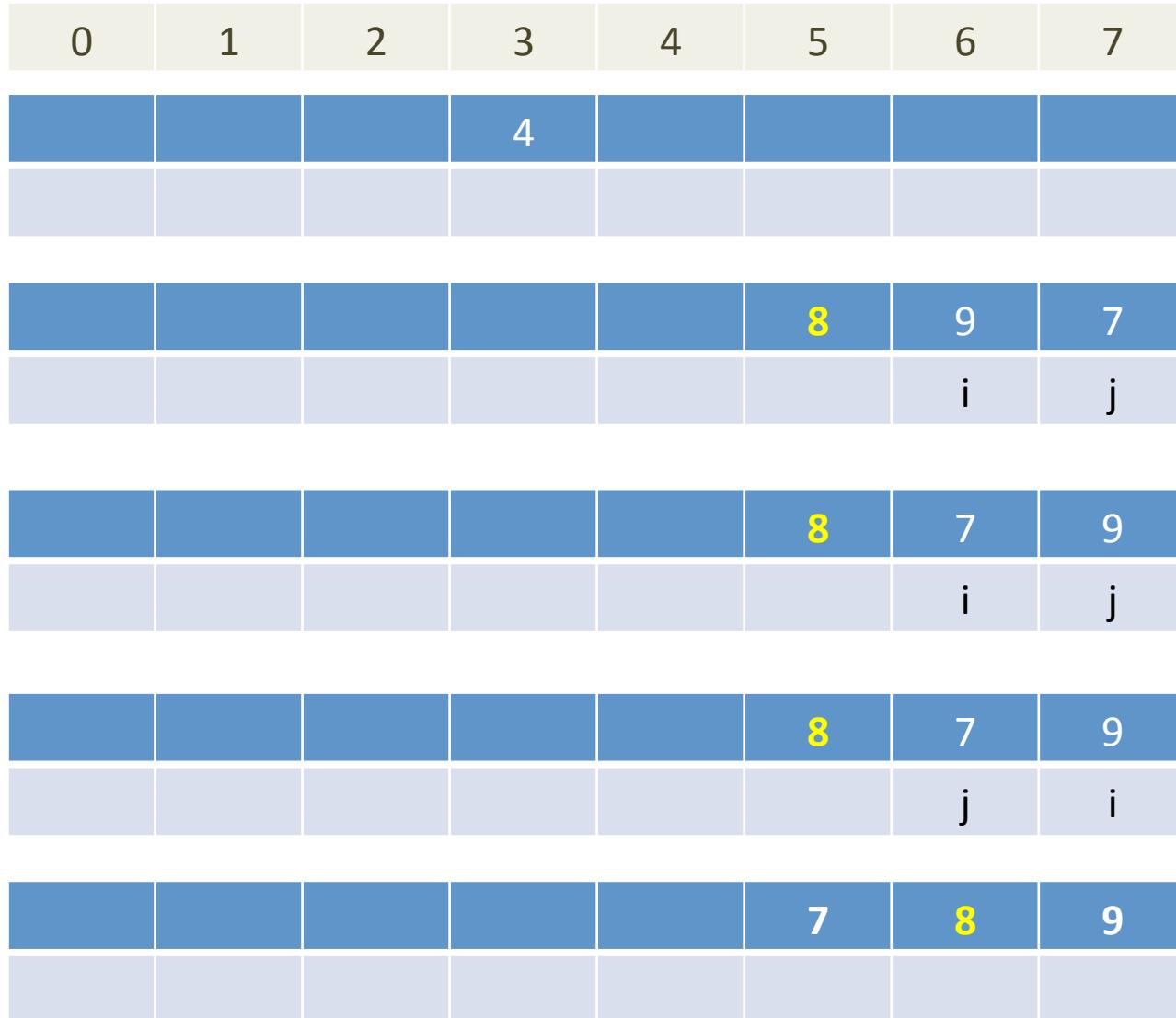
2	3	1	4				
	i	j					

2	1	3	4				
	i	j					

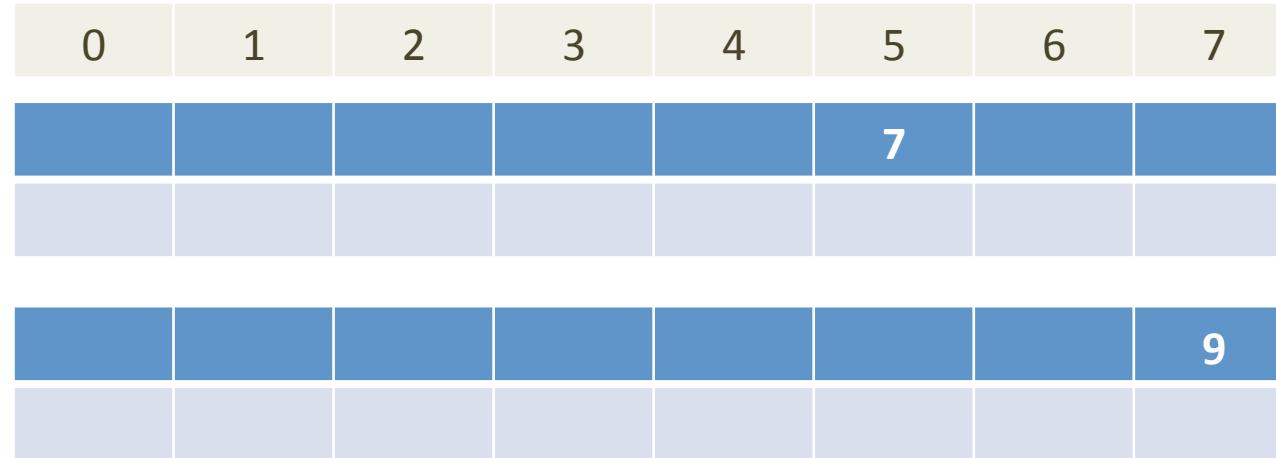
## Algorismes de dividir-i-vèncer: quicksort

0	1	2	3	4	5	6	7
2	1	3	4				
	j	i					
1	2	3	4				
1							
		3	4				
			ij				
		3	4				
			j	i			

## Algorismes de dividir-i-vèncer: quicksort



## Algorismes de dividir-i-vèncer: quicksort



## Algorismes de dividir-i-vèncer: **quicksort**

Quina és l'eficiència del *quicksort*?

Observació: el nombre de comparacions que fa abans d'una partició són  $n+1$  si els índexs es creuen i  $n$  si coincideixen.

Si totes les particions passen al mig del vector tenim el **millor cas**, i el nombre de comparacions serà:

$$C_{\text{millor}}(n) = 2C_{\text{millor}}(n/2) + n$$

I segons el teorema Master això és  **$O(n \log n)$** .

## Algorismes de dividir-i-vèncer: quicksort

En el **pitjor cas** (p.e. [0,1,2,3,4]), totes les particions són als extrems (alguna de les subllistes estarà buida), llavors el nombre de comparacions serà:

$$C_{pitjor}(n) = O(n^2)$$

En el **cas promig**, i el nombre de comparacions serà:

$$C_{promig}(n) = \frac{1}{n} \sum_{s=0}^{n-1} [(n+1) + C_{promig}(s) + C_{promig}(n-1-s)]$$

Que si resolem resulta en:  $C_{promig}(n) \approx 2n \ln n \approx 1.38n \log_2 n$

**És a dir, en el cas promig fa només un 38% més de comparacions que en el millor cas!**

## Algorismes de dividir-i-vèncer: **quicksort**

*Quicksort* és l'algorisme que fa servir Unix per ordenar amb la seva instrucció `sort`.

```
$ cat phonebook
Smith, Brett 555-4321
Doe, John 555-1234
Doe, Jane 555-3214
Avery, Cory 555-4321
Fogarty, Suzie 555-2314
```

```
$ sort phonebook
Avery, Cory 555-4321
Doe, Jane 555-3214
Doe, John 555-1234
Fogarty, Suzie 555-2314
Smith, Brett 555-4321
```

## Algorismes de dividir-i-vèncer: multiplicació de matrius

$$X \times Y = Z$$
$$Z_{ij} = \sum_{k=1}^n X_{ik} Y_{kj}$$

És evident que la implementació directa de la multiplicació de matrius és  $O(n^3)$ : s'han de calcular  $n^2$  elements, i cada càlcul és  $O(n)$ .

Fins a 1969 es pensava que no es podia fer d'una altra manera!

## Algorismes de dividir-i-vèncer: multiplicació de matrius

Però a 1969, el Dr. Volker Strassen va trobar una manera més òptima:

Es va basar en que el producte de dues matrius ( $n \times n$ ) es pot calcular a partir de la seva descomposició en blocs ( $n/2 \times n/2$ ):

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}, \quad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}.$$

$$XY = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$$

## Algorismes de dividir-i-vèncer: multiplicació de matrius

I que aquesta descomposició es pot expressar recursivament. La recurrència consisteix en passar d'una matriu  $(n \times n)$  a 8 matrius  $(n/2 \times n/2)$ , i per tant la seva complexitat és:

$$T(n) = 8 T(n/2) + O(n^2)$$

Aquesta és la part  
correspondent a les  
sumes

Que resulta en una complexitat de  $O(n^3)$ ...

- Però.....

## Algorismes de dividir-i-vèncer: multiplicació de matrius

- Però Strassen es va adonar que aquestes operacions es podien agrupar així:

$$XY = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$$

on

$$P_1 = A(F - H)$$

$$P_2 = (A + B)H$$

$$P_3 = (C + D)E$$

$$P_4 = D(G - E)$$

$$P_5 = (A + D)(E + H)$$

$$P_6 = (B - D)(G + H)$$

$$P_7 = (A - C)(E + F)$$

## Algorismes de dividir-i-vèncer: multiplicació de matrius

- La complexitat ara és:

$$T(n) = 7 T(n/2) + O(n^2)$$

Que resulta en una complexitat de  $O(n^{\log_2 7}) \approx O(n^{2.81})$ .

```
>>> 1000000**3  
10000000000000000000  
>>> 100000**2.81  
72443596007499056
```

## Algorismes de dividir-i-vèncer: càlcul de la mediana

La **mediana** de  $[45,1,10,30,25]$  és 25, perquè és l'element que queda al mig si els ordenem. Per tant, la seva implementació directa és  **$O(n \log n)$** .

- **Ho podem fer lineal?**
- Considerem el següent problema (que subsumeix el problema de la mediana):

### Selecció

Entrada: Una llista de nombres  $S$ ; un enter  $k$ .

Sortida: El  $k$ -èssim element més petit de  $S$ .

*Per exemple, si  $k=1$ , és el valor mínim de  $S$ , i si és  $\lfloor |S|/2 \rfloor$  és la mediana.*

## Algorismes de dividir-i-vèncer: càlcul de la mediana

Anem a plantejar una solució de **dividir i vèncer**.

Suposem un nombre qualsevol  $v$  i que dividim la llista segons aquest nombre (per exemple,  $v=5$ ):

$S :$ 

2	36	5	21	8	13	11	20	5	4	1
---	----	---	----	---	----	----	----	---	---	---

$S_L :$ 

2	4	1
---	---	---

$S_v :$ 

5	5
---	---

$S_R :$ 

36	21	8	13	11	20
----	----	---	----	----	----

Elements menors que  $v$

Elements iguals que  $v$

Elements més grans que  $v$

## Algorismes de dividir-i-vèncer: càlcul de la mediana

Ara la cerca es podria limitar a una de les tres subllistes: si busquéssim el 8è element, ha de ser el tercer element més petit de  $S_R$  atès que els elements de  $S_L$  i  $S_V$  són 5.

En general:

$$\text{selection}(S, k) = \begin{cases} \text{selection}(S_L, k) & \text{if } k \leq |S_L| \\ v & \text{if } |S_L| < k \leq |S_L| + |S_v| \\ \text{selection}(S_R, k - |S_L| - |S_v|) & \text{if } k > |S_L| + |S_v|. \end{cases}$$

Aquestes subllistes es podem calcular en temps lineal!

## Algorismes de dividir-i-vèncer: càlcul de la mediana

Ja tenim l'algorisme recursiu definit, excepte com definir  $v$ .

L'ideal seria que  $v$  partís les llistes per la meitat:

$$|S_L|, |S_R| \approx \frac{1}{2}|S|.$$

Aleshores la complexitat seria:

$$T(n) = T(n/2) + O(n)$$

que és una complexitat lineal!!

## Algorismes de dividir-i-vèncer: càlcul de la mediana

La solució és triar-lo de forma aleatòria cada vegada!

En el **pitjor cas** farem:  $n + (n - 1) + (n - 2) + \dots + \frac{n}{2} = O(n^2)$

En el **millor cas** farem  $O(n)$

En el **cas promig** es pot demostrar que és  $O(n)$ .

```

import random
def kSelect(A,k,length):
    # escollim una posició r de forma aleatoria
    # entre 1 i length(A)
    n = length-1
    r = random.randint(0, length-1)
    A1 = []
    A2 = []
    pivot = A[r]
    # construim lallista mes petita i la mes gran
    for i in range ( 0 , n+1):
        if A[i] < pivot : A1.append(A[i])
        if A[i] > pivot : A2.append(A[i])
    if k <= len(A1):
        # cerquem a la llista dels elements mes petits
        return kSelect(A1, k ,len(A1))
    if k > len(A) - len(A2):
        # cerquem a la llista dels elements mes grans
        return kSelect(A2, k-(len(A)-len(A2)),len(A2))
    else : return pivot

A = range(1,10001)
random.shuffle(A)
length = len(A)
value = kSelect(A,length/2,length)
print value
value = kSelect(A,1,length)
print value

```