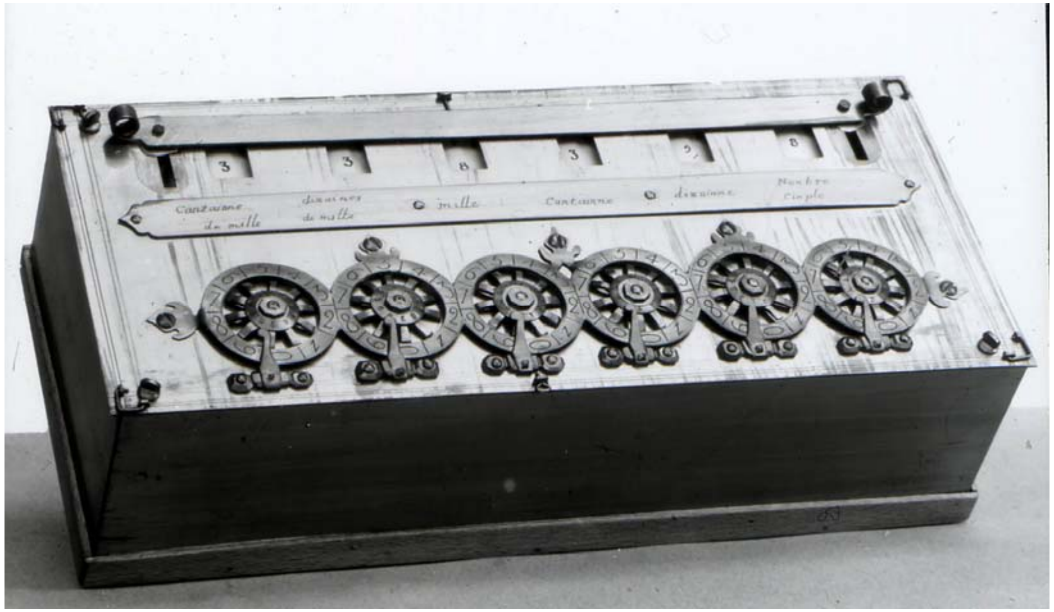
UNIVERSITAT DE BARCELONA

Grau d'Enginyeria Informàtica
Facultat de Matemàtiques i Informàtica



Algorísmica

Dividir i vèncer

Mireia Ribera i 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.

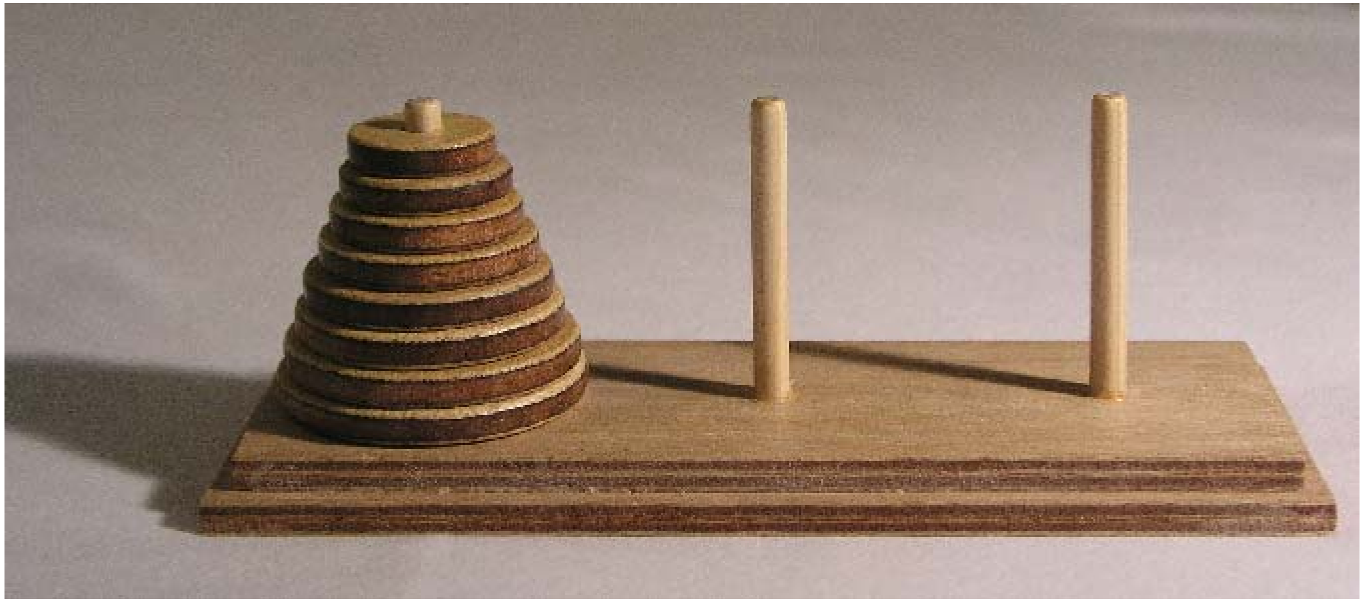
Algorismes de dividir i vèncer

Les qüestions a resoldre són tres:

- Com anem dividint el problema de forma recursiva?
- Quina és la solució al darrer pas de la recursió?
- Com combinem les solucions recursives per assolir la solució del problema complet?

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.

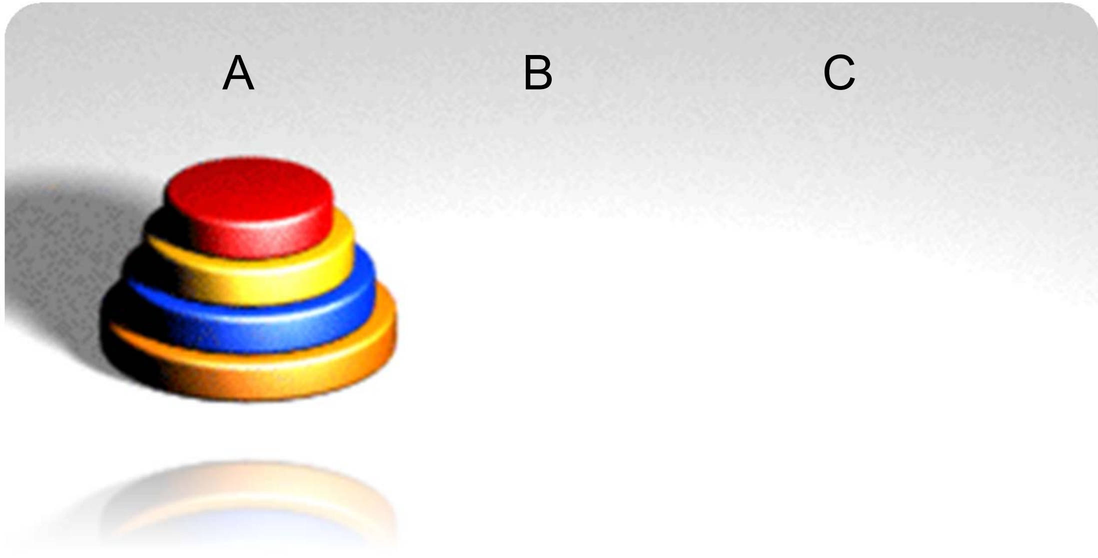
If the legend were true, and if the priests were able to move disks at a rate of one per second, using the smallest number of moves, it would take them $2^{64}-1$ seconds or roughly 585 billion years; it would take 18,446,744,073,709,551,615 turns to finish.



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

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

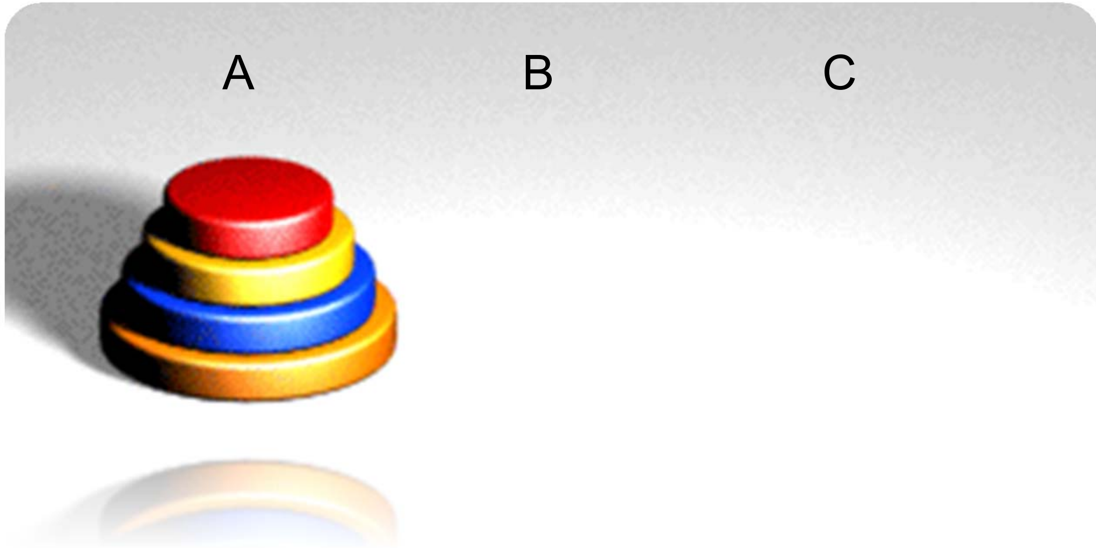
Mou els n-1 discs de dalt des de *a* a *b* fent servir *c* d'auxiliar.

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 $O(2^n)$

```
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)
```



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

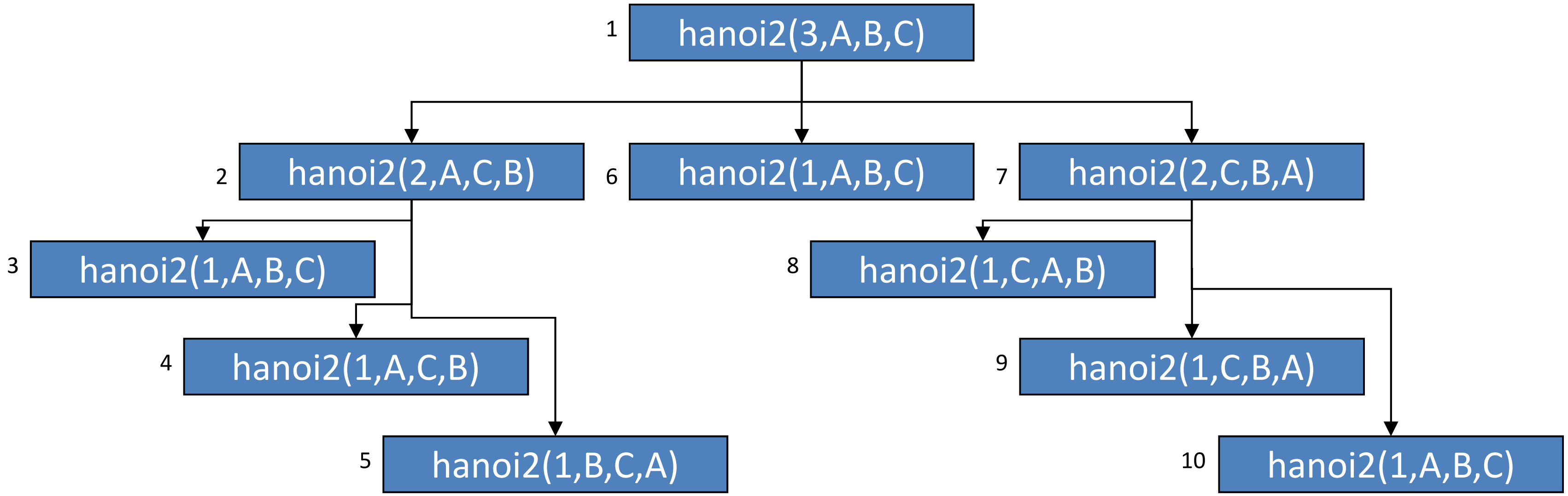
Recursivitat

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)
```

Recursivitat

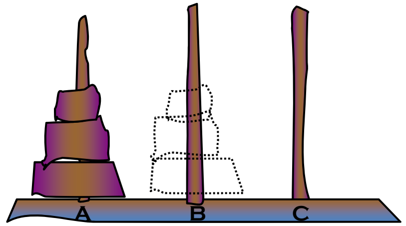
```
def hanoi2(n,source="A",spare="B",dest="C"):
    if n==1: print source+"->"+spare
    else:
        hanoi2(n-1, source,dest, spare)
        hanoi2(1, source, spare,dest)
        hanoi2(n-1,dest, spare, source)

hanoi2(3)
```



Recursivitat

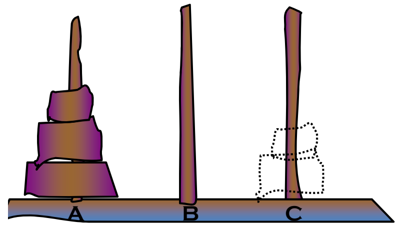
The initial call 1 is made, and **solveTowers** begins execution:

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



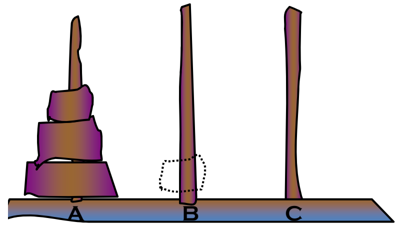
At point X, recursive call 2 is made, and the new invocation of the method begins execution:

```
count  = 3       X     count  = 2
source = A    ----->   source = A
dest   = B             dest   = C
spare  = C             spare  = B
```



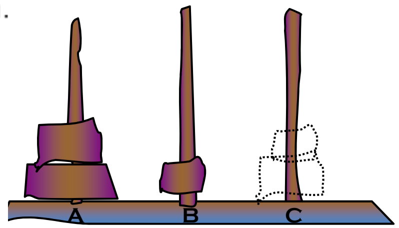
At point X, recursive call 3 is made, and the new invocation of the method begins execution:

```
count  = 3       X     count  = 2       X     count  = 1
source = A    ----->   source = A    ----->   source = A
dest   = B             dest   = C             dest   = B
spare  = C             spare  = B             spare  = C
```



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

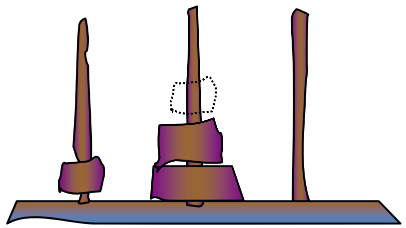
```
count  = 3       X     count  = 2             count  = 1
source = A    ----->   source = A             source = A
dest   = B             dest   = C             dest   = B
spare  = C             spare  = B             spare  = C
```



Recursivitat

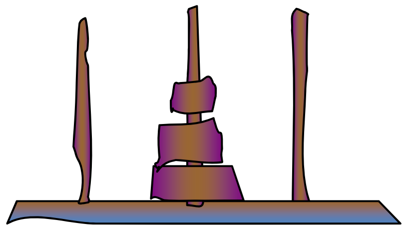
At point Z, recursive call 10 is made, and the new invocation of the method begins execution:

```
count  = 3         count  = 2         count  = 1
source = A    Z    source = C    Z    source = A
dest   = B  ---->  dest   = B  ---->  dest   = B
spare  = C         spare  = A         spare  = C
```



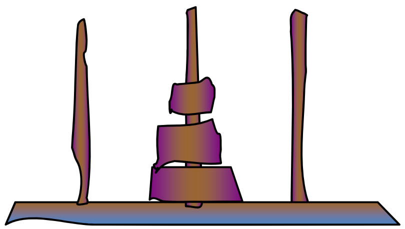
This is the base case, so a disk is moved, the return is made, and the method continues execution.

```
count  = 3         count  = 2         count  = 1
source = A    Z    source = C         source = A
dest   = B  ---->  dest   = B         dest   = B
spare  = C         spare  = A         spare  = C
```



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

```
count  = 3         spare  = 2         count  = 1
source = A         source = C         source = A
dest   = B         dest   = B         dest   = B
spare  = C         spare  = A         spare  = C
```



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 está 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>=0, llavors:

a = en quants problemes dividim

b = com dividim la n a cada subproblema

d = exponent ordre complexitat combinació respostes

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

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:

$$\boldsymbol{d > \log_b a, \qquad 2 > \log_2 2 = 1}$$

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

Algorismes de dividir-i-vèncer: mergesort

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

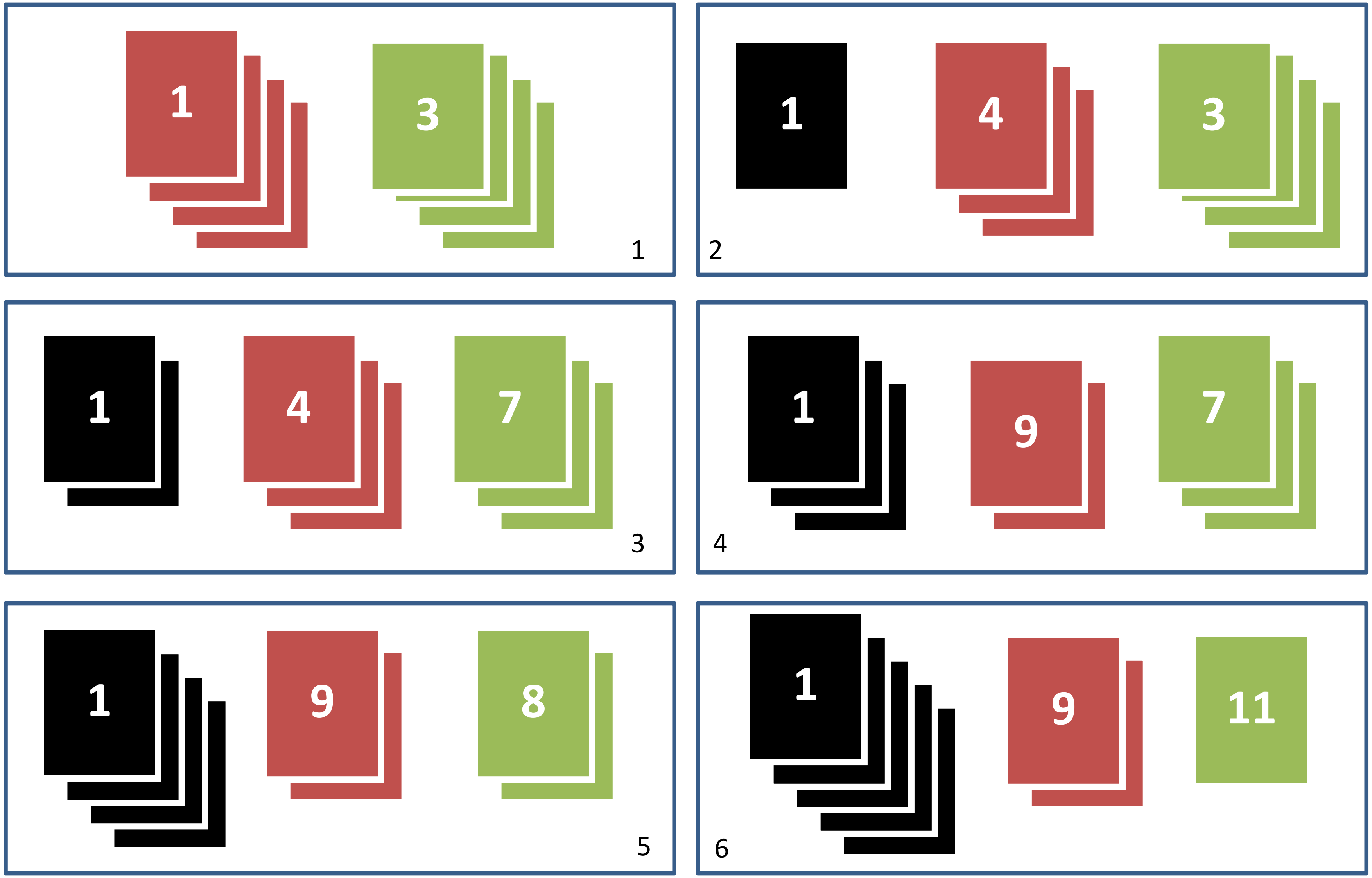
Idea de recursió!

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).

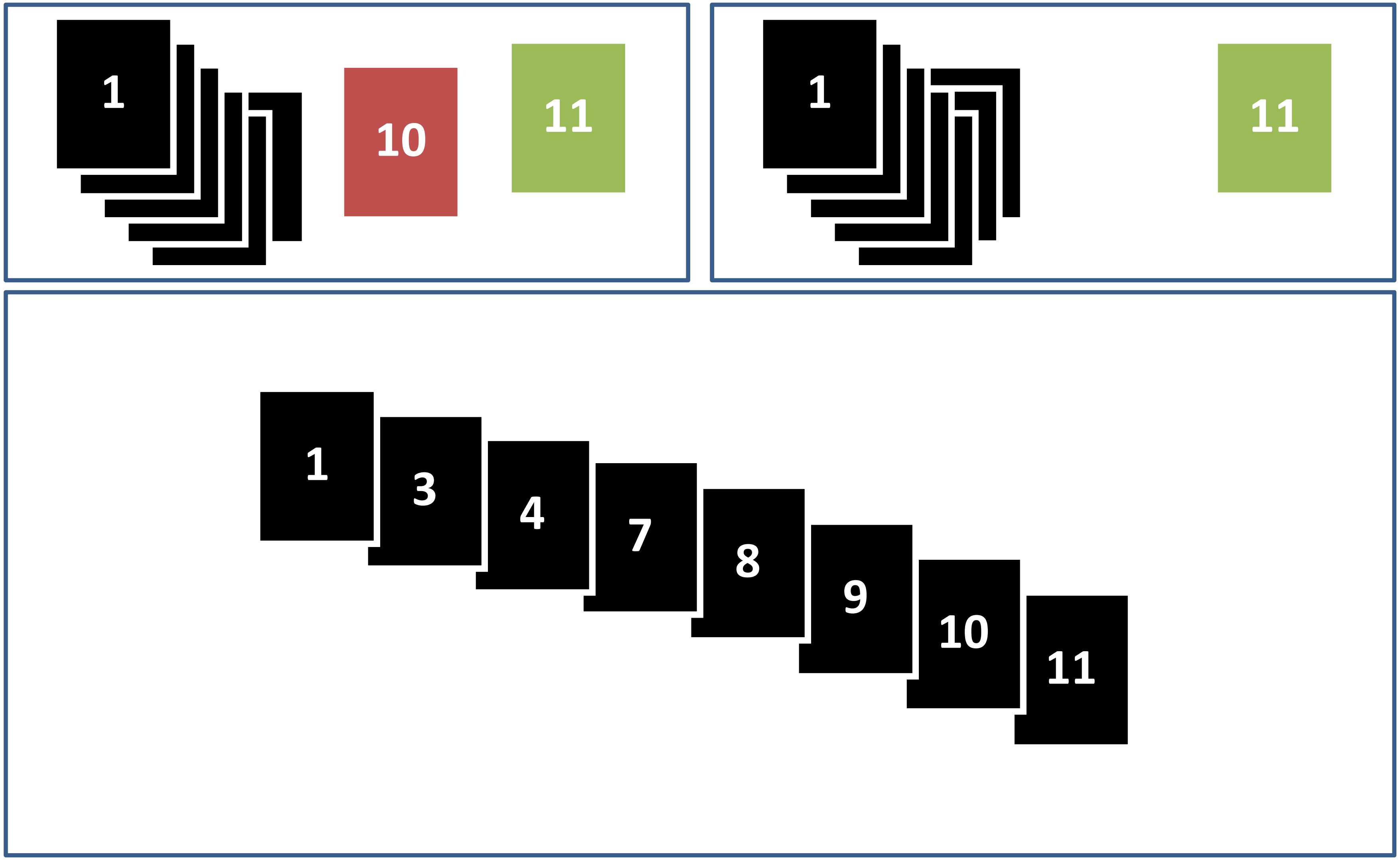
1. Comparem les dues cartes de sobre de tot de cada conjunt i escollim la més petita, que posem a un nou conjunt de cartes ordenades (pel darrera, si n'hi ha alguna).
2. Repetim 1 fins que un dels conjunts estigui buit, i l'altre l'afegim per darrera al conjunt de cartes ordenades.

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

Però en el cas general no partim de dos conjunts de cartes ordenades!

El que podem fer és descomposar el problema en subproblemes més petits fins que arribem als casos que sabem resoldre!

Algorismes de dividir-i-vèncer: mergesort

Això ho podem expressar recursivament:

```
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)
```

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**!

```
def merge(x,y):
    if len(x) < 1:
        return y
    if len(y) < 1:
        return x
    if x[0] <= y[0]:
        return [x[0]] + merge(x[1:],y)
    else:
        return [y[0]] + merge(x,y[1:])
```

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.

Algorismes de dividir-i-vèncer: mergesort

La seqüència de `mergesort` en un cas concret:

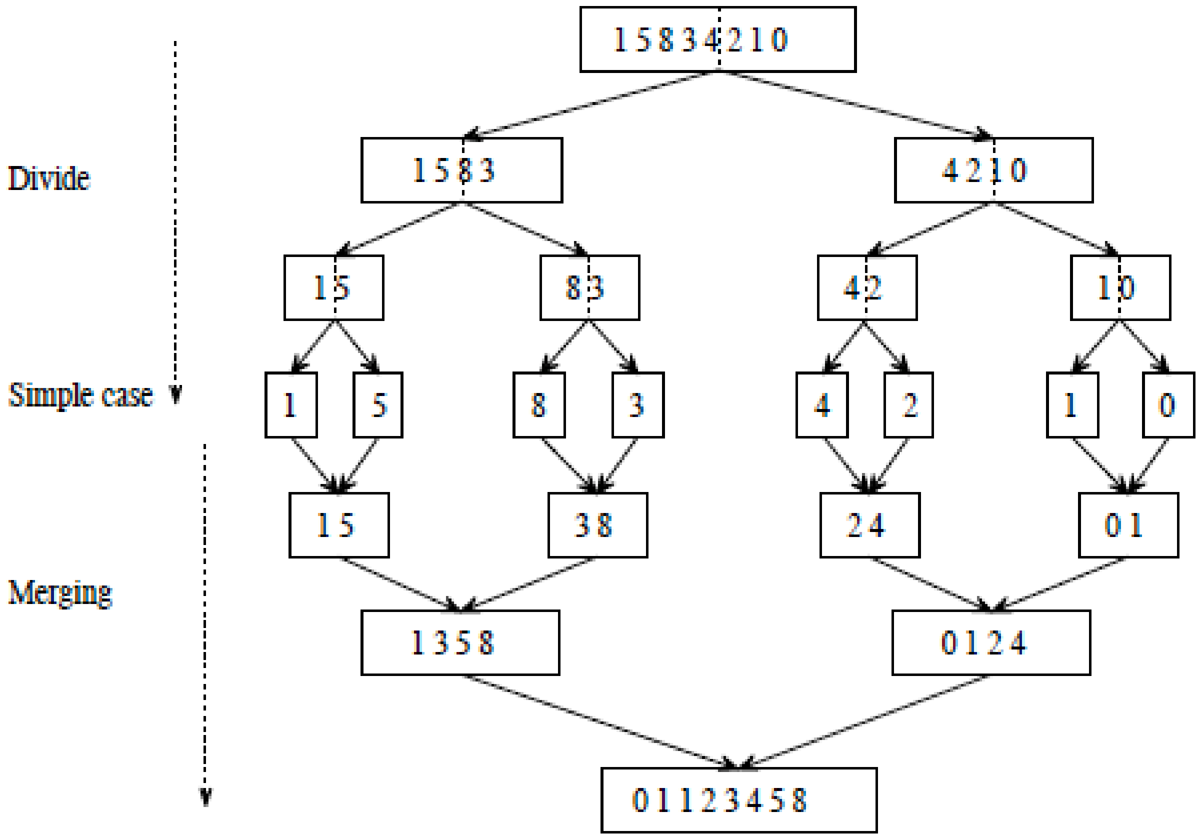


Figure 1: Mergesort

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

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

La funció `merge` té una complexitat per cada crida recursiva *O(n)* (en el pitjor dels casos). Per tant d=1.

Quan dividim creem 2 problemes, de mida n/2. Per tant a=2, b=2

`mergesort` té una complexitat:

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

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'estratègia de dividir i vèncer.

Aquest algorisme divideix el vector basant-se en els valors: reordena els valors de *A[0,...,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:

$$\underbrace{A[0] \ldots A[s-1]}_{\text{Tots són} \,\leq\, A[s]} \quad \underbrace{A[s]}_{\text{Pivot}} \quad \underbrace{A[s+1] \ldots A[n-1]}_{\text{Tots són} \,\geq\, A[s]}$$

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.

Re-escriptura de la crida a la funció per fer més simple la recursió.

```
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 `pivot=A[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.

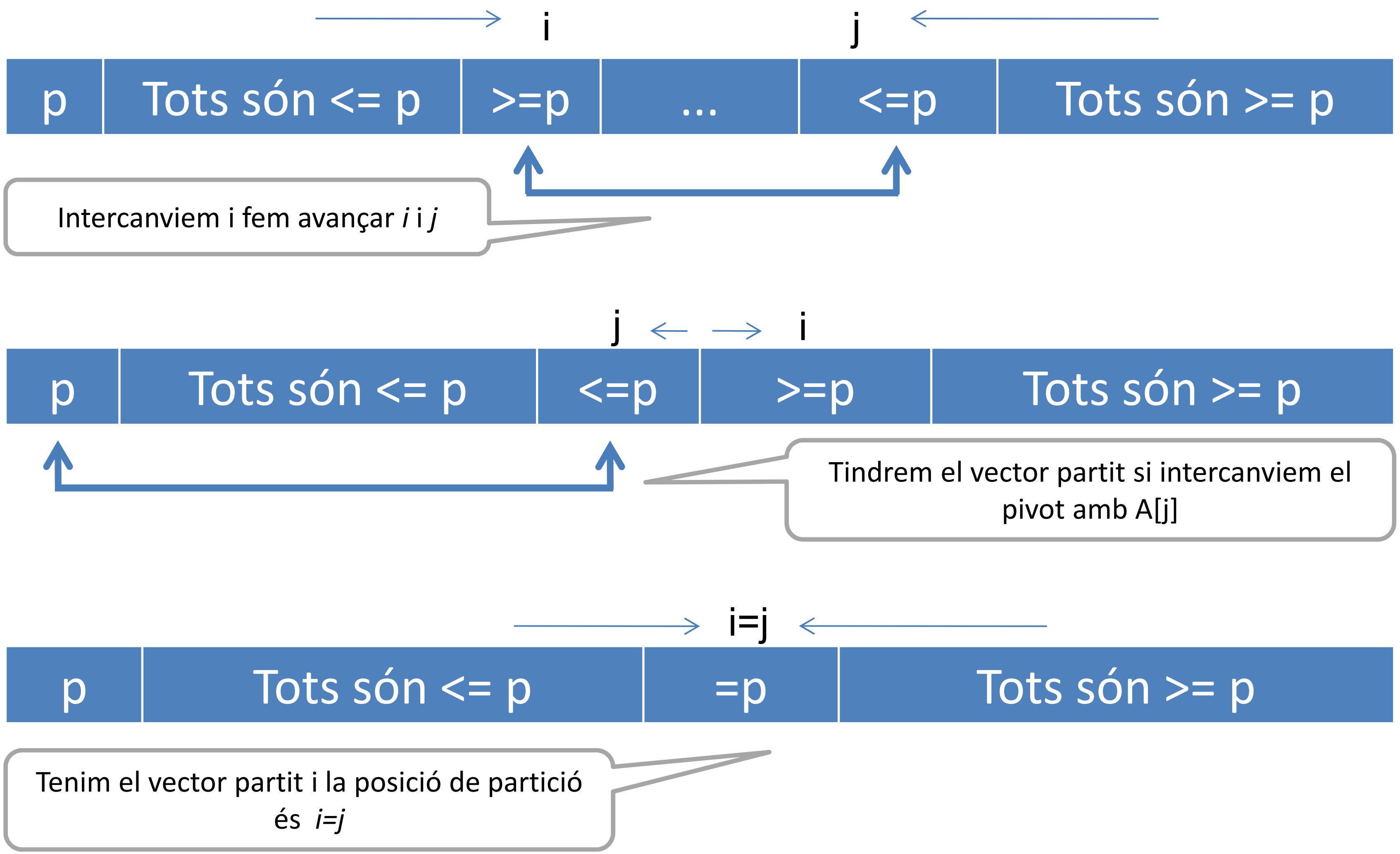
Algorismes de dividir-i-vèncer: quicksort

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):
    mid = (first + last)//2  #ordenem A[first],A[mid],A[last]
    if A[first] > A [mid]: A[first], A[mid] = A[mid], A[first]
    if A[first] > A [last]: A[first], A[last] = A[last], A[first]
    if A[mid] > A[last]:   A[mid], A[last] = A[last], A[mid]
    A[mid], A[first] = A[first], A[mid] #valor mig a l'inici
    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] #intercanviem, fem avançar i j
    A[j], A[pivot] = A[pivot], A[j] #vector partit, pivot=j
    return j

A = [3,7,2,4,1,80]
quick_sort(A)
A
>>> [1,2,3,4,7,80]
```

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

0	1	2	3	4	5	6	7
			4				

					8	9	7
						i	j

					8	7	9
						i	j

					8	7	9
						j	i

					7	**8**	**9**

Algorismes de dividir-i-vèncer: quicksort

0	1	2	3	4	5	6	7
					7		

							9

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

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:

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

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

Algorismes de dividir-i-vèncer: quicksort

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

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

Sempre podem desordenar la llista al principi i aquest cas no existirà!

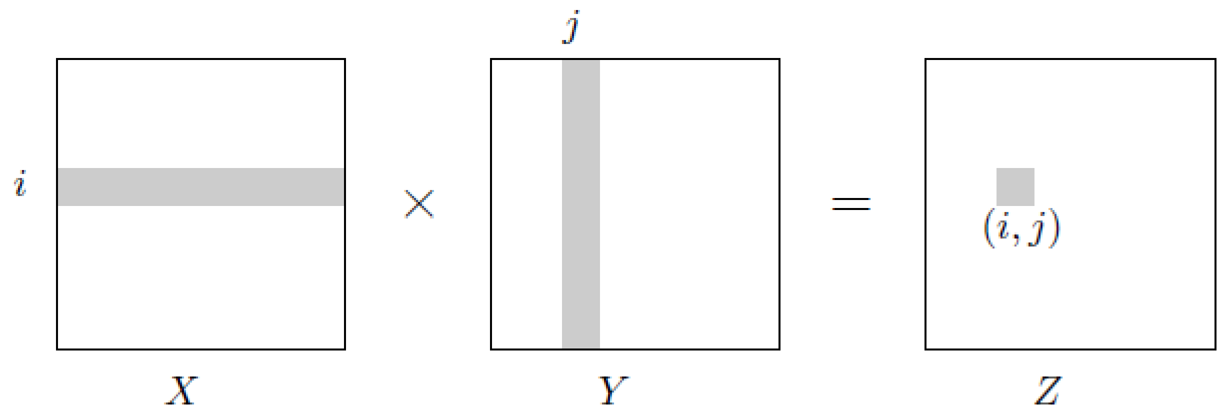
En el **cas promig**, i el nombre de comparacions serà:

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

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

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

Algorismes de dividir-i-vèncer: multiplicació de matrius



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

És evident que la implementació directa de la multiplicació de matrius és ***O(n³)***: 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 x n)* es pot calcular a partir de la seva descomposició en blocs (*n/2* x *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 x n*) a 8 matrius (*n/2 x n/2*), i per tant la seva complexitat és:

Aquesta és la part corresponent a les sumes

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

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

$$\begin{aligned} P_1 &= A(F - H) \\ P_2 &= (A + B)H \\ P_3 &= (C + D)E \\ P_4 &= D(G - E) \end{aligned} \qquad \begin{aligned} P_5 &= (A + D)(E + H) \\ P_6 &= (B - D)(G + H) \\ P_7 &= (A - C)(E + F) \end{aligned}$$

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
1000000000000000000
>>> 1000000**2.81
7244359600749856
```

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):

$$S: \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|}\hline 2 & 36 & 5 & 21 & 8 & 13 & 11 & 20 & 5 & 4 & 1 \\ \hline\end{array}$$

$$S_L: \begin{array}{|c|c|c|}\hline 2 & 4 & 1 \\ \hline\end{array} \qquad S_v: \begin{array}{|c|c|}\hline 5 & 5 \\ \hline\end{array} \qquad S_R: \begin{array}{|c|c|c|c|c|c|}\hline 36 & 21 & 8 & 13 & 11 & 20 \\ \hline\end{array}$$

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: $n + (n-1) + (n-2) + \cdots + \frac{n}{2} = O(n^2)$

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 aleatòria
        # entre 1 i length(A)
        n = length-1
        r = random.randint(0, length-1)
        A1 = []
        A2 = []
        pivot = A[r]
        # construim la llista més petita i la més 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
```