

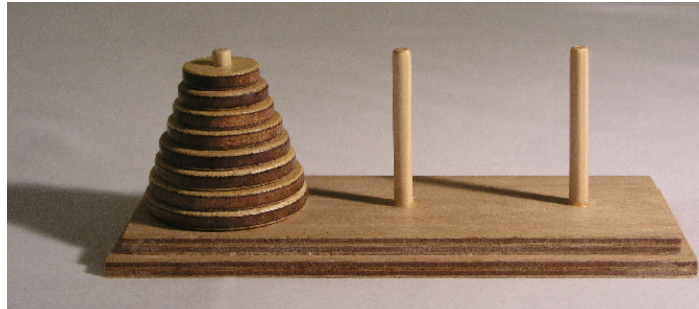
Resum del tema 6

- Recursió
 - Torres de Hanoi
- Dividir i vencer
 - Teorema Màster
 - MergeSort
 - merge
 - QuickSort
 - partition
 - Multiplicació de matrius (Strassen)
 - Càlcul de la mediana

La recursió

Per poder comprendre l'estratègia de dividir i vèncer ens cal aprofundir en el concepte de la recursió. I l'estudiarem a través d'un exemple clàssic: les torres de Hanoi

Exemple: el problema de les torres de Hanoi



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.

Exemple: el problema de les torres de Hanoi

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.

La idea bàsica per resoldre-ho amb un algorisme recursiu és que:

- 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.
- Llavors puc passar la peça que queda a A a C i oblidar-me d'ella, ja està ben col·locada!
- 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!

Exemple: el problema de les torres de Hanoi

```
def moveTower(height,fromPole, toPole, withPole): # height és el nombre de discs
    if height >= 1:                               # que hi ha al pal origen
        moveTower(height-1,fromPole,withPole,toPole)
        moveDisk(fromPole,toPole)
        moveTower(height-1,withPole,toPole,fromPole)
    return
def moveDisk(fp,tp):
    print("moving disk from",fp,"to",tp)

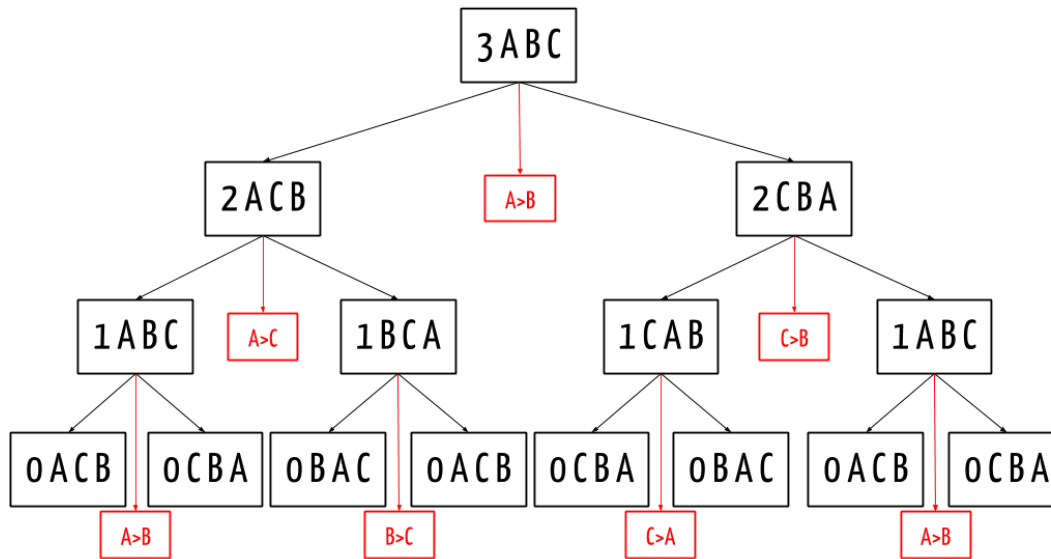
moveTower(3,"A","B","C")
```

```
moving disk from A to B
moving disk from A to C
moving disk from B to C
moving disk from A to B
moving disk from C to A
moving disk from C to B
moving disk from A to B
```

Després veurem que la complexitat és exponencial $O(2^n)$.

Exemple: el problema de les torres de Hanoi

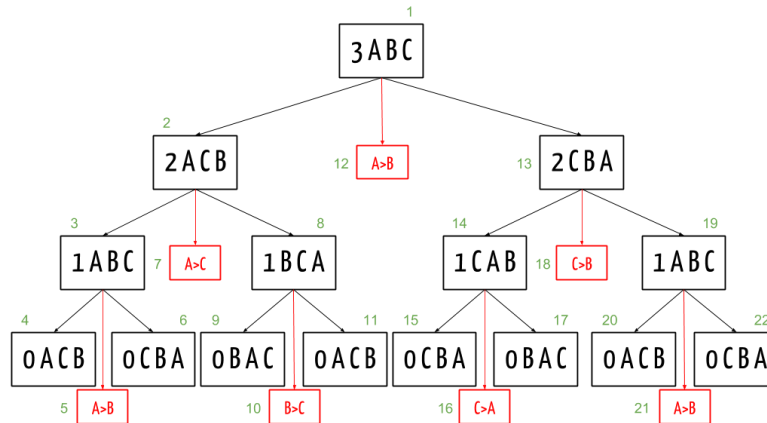
Les crides recursives generen aquest arbre (a cada node hi ha els paràmetres, que recordem eren: height, fromPole, toPole, withPole):



per exemple: 3ABC vol dir que hi ha tres discs a A, que es mouran a B amb l'ajut de C; 1CAB vol dir que hi ha 1 disc a C, que es mouran a A amb l'ajut de B.

Exemple: el problema de les torres de Hanoi

L'ordre de generació de la solució és:



```
def moveTower(height,fromPole, toPole, withPole):
    if height >= 1:
        moveTower(height-1,fromPole,withPole,toPole)
        moveDisk(fromPole,toPole)
        moveTower(height-1,withPole,toPole,fromPole)
    return
```

Exemple: el problema de les torres de Hanoi

- Què passaria amb l'arbre si enlloc de 3 peces en tenim 4?

Que tindria un nivell més amb el doble de fulles.

- Quantes vegades es crida recursivament la funció (quantes fulles i nodes te l'arbre) si tenim n peces?

$$1 + 2 + 4 + 8 + \dots + 2^n = \sum_{i=0}^n 2^i = 2^{n+1} - 1$$

- Quina és la complexitat de l'algorisme?

Si fa $(2^{(n+1)} - 1)$ crides i cada una de les crides com a màxim executa instruccions de cost $O(1)$, el cost és $O(2^{(n+1)})$. És a dir, ordre exponencial, $O(2^n)$.

Algorismes de dividir i vèncer

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

- Dividir un problema en subproblemes que són instàncies més petites (des del punt de vista de la mida de l'entrada) del mateix problema.
- Resoldre recursivament aquests subproblemes.
- Combinar adequadament les solucions dels subproblemes per trobar la solució del problema original.

Les **qüestions a resoldre** són tres:

- Com anem dividint el problema en subproblemes de forma recursiva?
- Com aturem la recursió i donem una solució al darrer subproblema?
- Com combinem les solucions recursives per assolir la solució del problema complet?

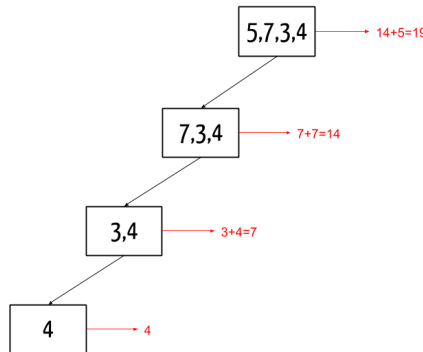
Exemple: Suma (recursiva) dels elements d'una llista.

```
def sum(l):  
    if len(l) == 1:  
        return l[0]  
    else:  
        return l[0] + sum(l[1:])
```

```
a =[5,7,3,8,10]
```

- Quina complexitat té aquest algorisme?

$O(n)$



Algorismes de dividir i vèncer: relacions de recurrència.

L'esquema general 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:

$$T(n) = aT(n/b) + O(n^d)$$

Aquest tipus de recurrència té una **solució tancada**, que està enunciada al **Teorema Master**.

Teorema Master

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

- **cas 1:** $T(n) = O(n^d)$ si $d > \log_b a$
 - **cas 2:** $T(n) = O(n^d \log n)$ si $d = \log_b a$
 - **cas 3:** $T(n) = O(n^{\log_b a})$ si $d < \log_b a$
-

Fixeu-vos que en el cas 1 el que mana és el cost de la combinació de solucions per què d és molt gran.

En el cas 3, en canvi, el que mana és $\log_b a$ que serà gran quan hi hagi molts subproblemes i no es redueixin gaire en cada divisió.

En el cas 2 àmbdues complexitats estan equilibrades.

Algorismes de dividir i vèncer: relacions de recurrència.

Exemple:

Si tenim un determinat problema que es pot dividir en 2 subproblemes ($a = 2$), cada un dels quals processa unes dades que són $1/2$ de les originals ($b = 2$), i la reconstrucció de la solució en costa $O(n^2)$...

... podem aplicar el cas (1) del teorema Master, perquè $2 > \log_2 1$...

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

Algorismes de dividir i vèncer per ordenar: *mergesort* i *quicksort*

A continuació coneixerem dos algorismes que ens ajudaran a ordenar una llista de forma més eficient, concretament amb una complexitat d' $O(n \log n)$.

Cadascun usa alguna funció auxiliar per preparar (*partition* al *quicksort*) o per recombinar (*merge* al *mergesort*) els subproblemes en que es basa la divisió.

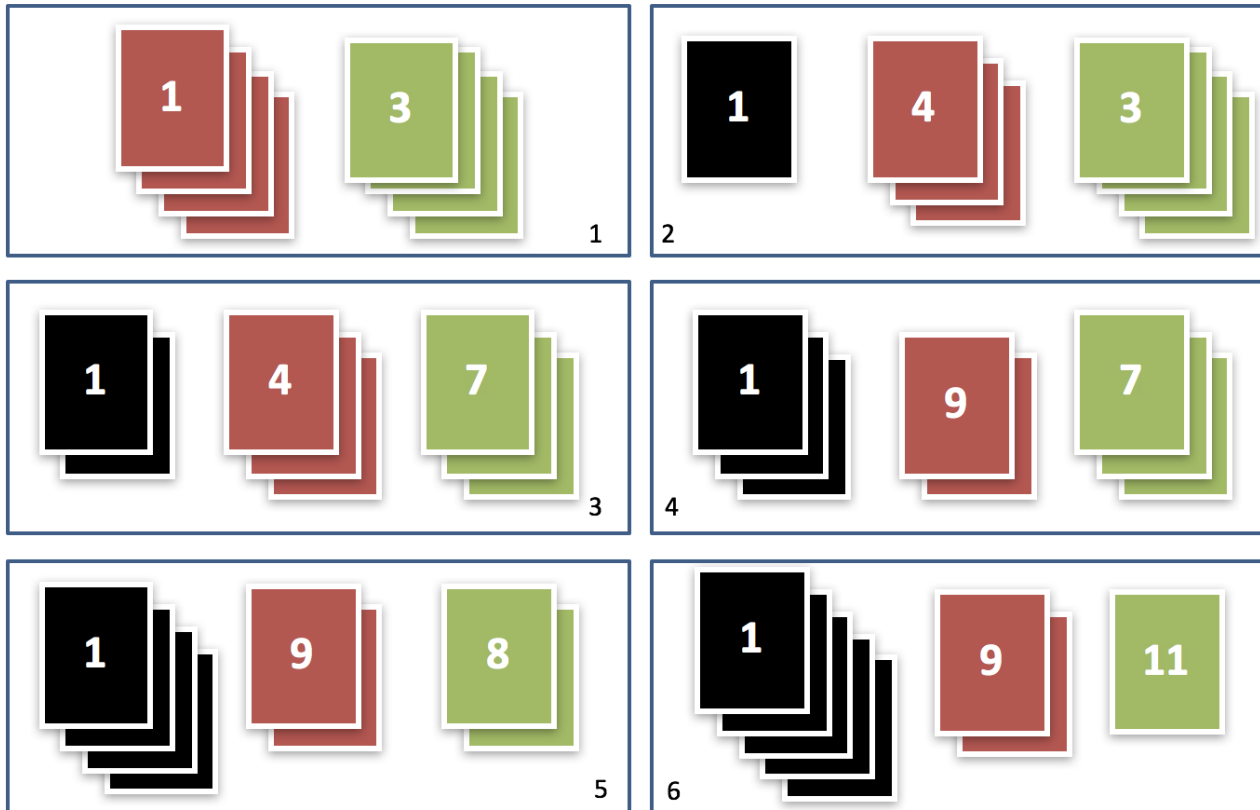
Algorismes de dividir-i-vèncer: **merge**

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). En aquest cas concret podem obtenir una llista ordenada barrejant les dues si seguim aquests passos:

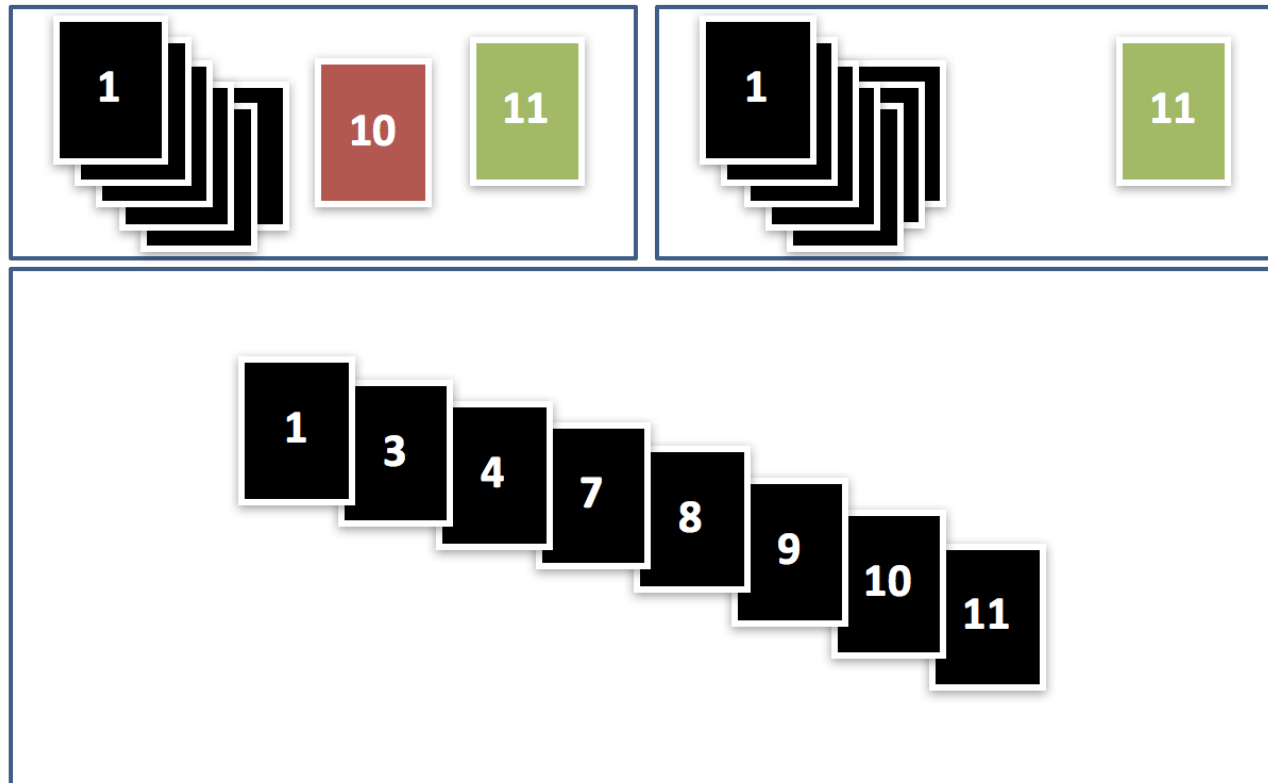
1. Comparem les dues cartes de sobre de tot de cada conjunt i escollim la més petita, que posem a un nou conjunt de cartes ordenades (pel darrera, si n'hi ha alguna).
2. Repetim 1 fins que un dels conjunts estigui buit, i el conjunt que encara té cartes l'afegim per darrera al conjunt de cartes ordenades.

El resultat és un conjunt de cartes ordenades!

Algorismes de dividir-i-vèncer: merge



Algorismes de dividir-i-vèncer: merge



Algorismes de dividir-i-vèncer: mergesort

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

`merge` és una funció auxiliar de l'algorisme general d'ordenació *mergesort*.

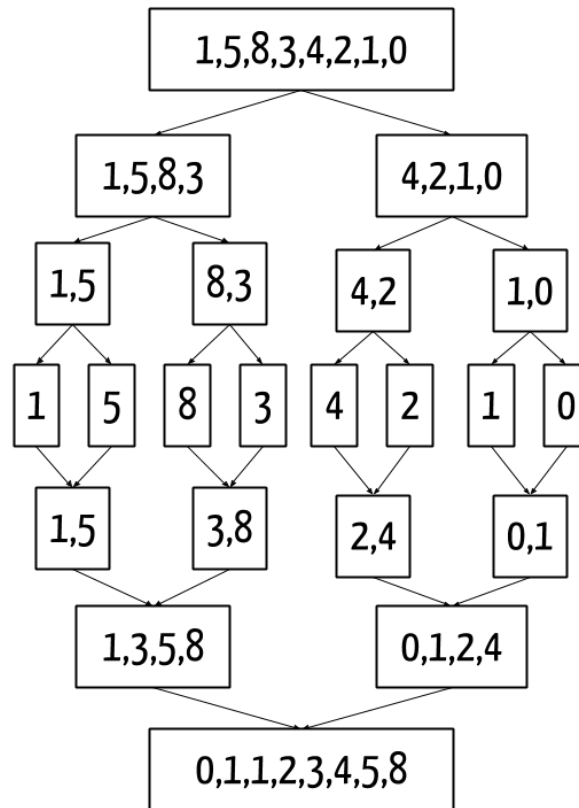
El que podem fer és descomposar el problema en subproblemes més petits fins que arribem als casos que sabem resoldre!

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**



Algorismes de dividir-i-vèncer: merge

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: merge

La versió no recursiva de merge seria aquesta:

```
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
```

Algorismes de dividir-i-vèncer: mergesort

Quina és la complexitat d'aquest algorisme?

La funció `merge` té una complexitat per cada crida recursiva $O(n)$ (en el pitjor dels casos). Com que aquesta és la funció que usem per combinar les solucions parcials, $d=1$.

Quan dividim creem 2 problemes, de mida $n/2$. Per tant, com que $a=2$ i $b=2$, $\log_b a = 1$, que és igual a d , podem aplicar el cas 2 del teorema Master: **mergesort té una complexitat $O(n \log n)$.**

La funció mergesort es pot implementar iterativament, de forma no recursiva, en la seva totalitat, però per fer-ho necessitem una estructura que no veurem en aquesta assignatura: 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 dels elements que conté: reordena els elements 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 tots els elements posteriors a la posició s siguin majors o iguals que $A[s]$:

$$A[0] \dots A[s-1] \leq A[s] \leq A[s+1] \dots A[n-1]$$

L'element $A[s]$ s'anomena **pivot**.

Per exemple:

3 56 34 2 \leq 99 \leq 134 345 111

és una partició de la llista [3 99 56 134 34 2 345 111]

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)    # quicksort_r té més paràmetres per  
                                      # facilitar les crides recursives  
  
def quick_sort_r(A , first, last):    # A és la llista, first i last els índexs  
                                      # entre els quals cal ordenar  
    if last > first:  
        pivot = partition(A, first, last) # dividim el problema  
        quick_sort_r(A, first, pivot - 1) # ordenem part esquerra  
        quick_sort_r(A, pivot + 1, last)  # ordenem part dreta
```

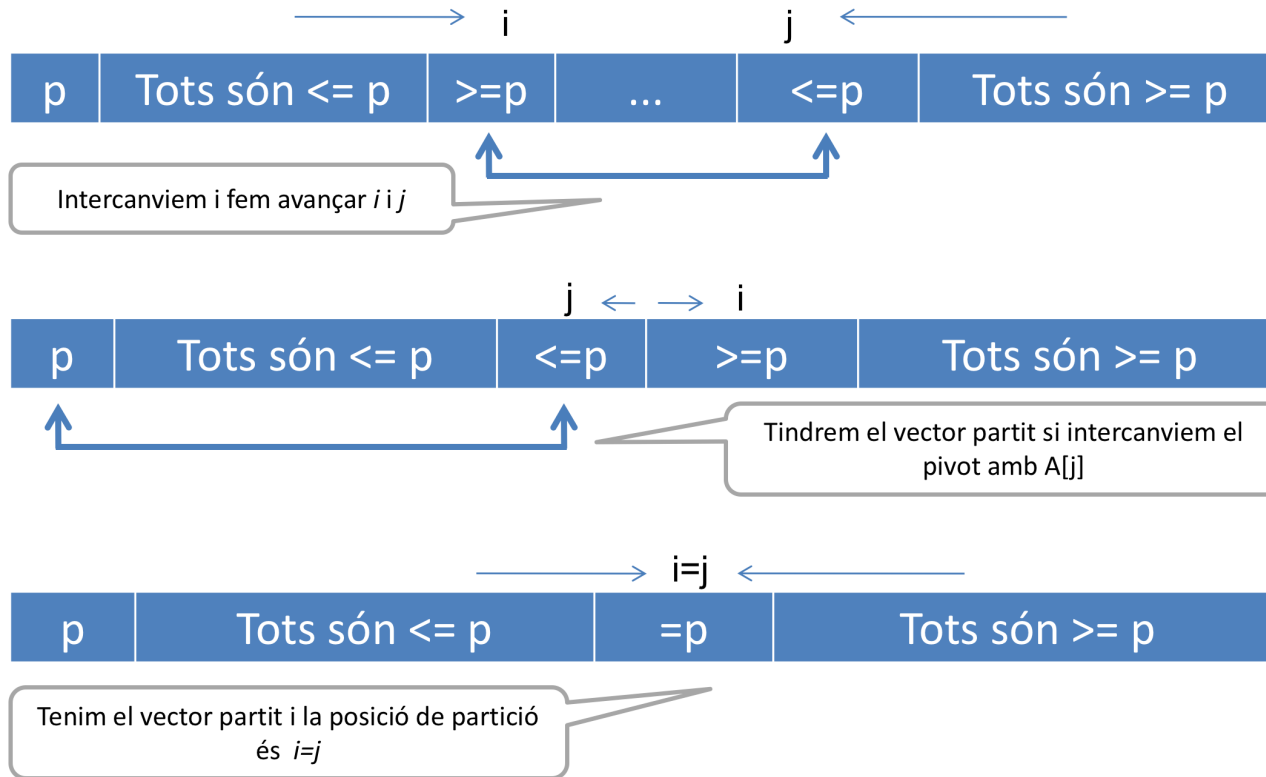

Algorismes de dividir-i-vèncer: **partition**

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.
 - 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.

Algorismes de dividir-i-vèncer: **partition**

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



Algorismes de dividir-i-vèncer: partition

```
def partition(A, first, last):
    # ordenem A[first], A[mid], A[last]
    mid = (first + last) // 2
    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]
    # inicialitzem pivot, i i j
    pivot = first
    i = first + 1
    j = last
    while True:
        # anem avançant
        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

quick_sort([3,7,2,4,1,80])
>>> [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 Linux/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:

$T(n) = 2 T(n/2) + n$, que segons el teorema Master és $O(n \log n)$.

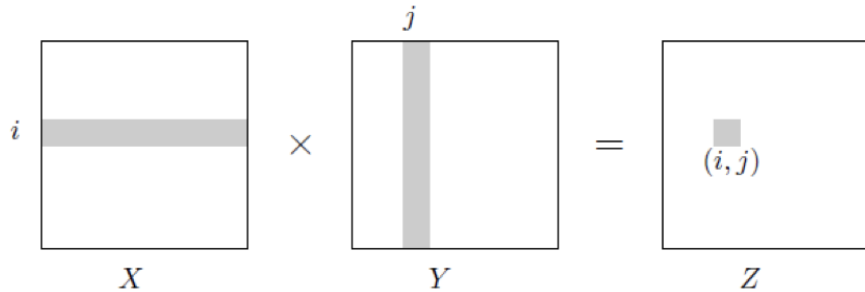
En **el pitjor cas** (p.e. $[4,3,2,1,0]$), totes les particions són als extrems (alguna de les subllistes estarà buida), llavors el nombre de comparacions serà $T(n) = O(n^2)$.

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

En el **cas promig** el nombre de comparacions serà $O(1,38 n \log_2 n)$, és a dir, un 38% més de comparacions que en el millor cas.

La complexitat se seguiria expressant com a **$O(n \log n)$** .

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 $O(n^3)$: 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 el 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 $T(n) = 8 T(n/2) + O(n^2)$.

Que resulta en una complexitat de $O(n^3)$... No hem guanyat res!

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

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

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

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

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

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

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

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

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

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 $O(n \log_2 7)$, que és aproximadament $O(n^{2.81})$.

```
1000000**3
>>> 1000000000000000000000000

1000000**2.81
>>> 72443596007499056
```

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

La mediana de $[45, 1, 10, 30, 25]$ és 25, perquè la **mediana** es defineix com l'element que queda al mig del vector si ordenem els seus elements. 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):

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

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

Per exemple, si $k=1$, la sortida hauria de ser el valor mínim de S .

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 fem una partició de la llista segons aquest nombre (per exemple, $v=5$) en tres parts, S_{left} , S_v i S_{right} :

$S :$

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_{right} atès que a S_{left} i S_v hi ha un total de 5 elements.

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, però encara no sabem com definir v .

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

Ja tenim l'algorisme recursiu definit, però encara no sabem com definir v .

L'ideal seria que v partís les llistes per la meitat. Aleshores la complexitat seria $T(n) = T(n/2) + O(n)$, que és una complexitat lineal!!

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

- En el pitjor cas farem: $n + (n-1) + (n-2) + \dots = O(n^2)$.
- En el millor cas farem $O(n)$.
- En el cas promig es pot demostrar que és $O(n)$.

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

```
import random
def kSelect(A,k,length): # escollim una posició r random( 1 - length(A))
    n = length-1
    r = random.randint(0, length-1)
    A1 = []
    A2 = []
    pivot = A[r]
    for i in range ( 0 , n+1): # construim la llista més petita i la més gran
        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
```

Possibles preguntes d'exàmen relacionades amb el tema 6

1. Quin cost té l'algorisme quicksort? És més ràpid que el mergesort?
2. Calcula les complexitats dels següents algorismes, segons el teorema Màster:
 - $T(n) = 3T(n/2) + n^2$
 - $T(n) = 4T(n/2) + n^2$
 - $T(n) = T(n/2) + 2^n$
 - $T(n) = 2^n T(n/2) + n^n$
 - $T(n) = 16T(n/4) + n$
 - $T(n) = 2T(n/2) + n \log n$
3. A cada iteració de partition quants elements addicionals tenim ordenats?