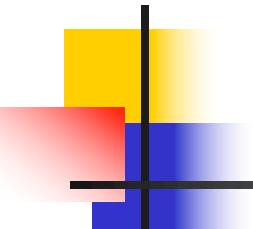


## Chapitre 4

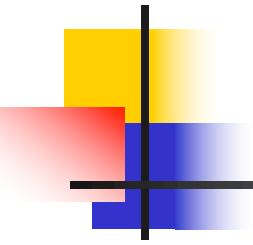
# Diviser pour régner



# Gabarit général de l'approche diviser pour régner

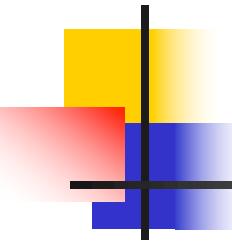
- Cette célèbre technique de conception a permis d'obtenir des algorithmes très efficaces pour plusieurs problèmes différents
- Décrivons le gabarit général utilisé pour ce type d'algorithme
- Cette approche nécessite l'emploi d'un sous algorithme, que nous nommerons *adhoc*, pour solutionner tout instance dont la taille est suffisamment petite.
  - (plus petite qu'un seuil préalablement défini)
- Désignons par  $|x|$  la taille de l'instance  $x$
- Le gabarit général est le suivant:

**ALGORITHME** DiviserPourRégner( $x$ )  
    **if**  $|x| <$  seuil **return** *adhoc*( $x$ )  
    **diviser**  $x$  en plus petites instances  $x_1, x_2, \dots, x_r$   
    **for**  $i \leftarrow 1$  **to**  $r$  **do**  $y_i \leftarrow$  DiviserPourRégner( $x_i$ )  
    **recombiner** les  $y_i$  pour obtenir la solution  $y$  de  $x$   
    **return**  $y$



## Gabarit général de l'approche diviser pour régner (suite)

- La valeur du seuil est une constante (préalablement choisie) qui ne doit pas dépendre de la taille de l'instance
  - Sinon, ce ne serait plus un algorithme diviser pour régner
- La taille de chaque petite instance  $x_i$  devrait être la plus semblable possible
  - Si une des petites instances est substantiellement plus grosse que les autres, on ne divise plus et l'algorithme devient « diminuer pour régner » (prochain chapitre).
- Soit  $|x| = n$  (la taille de l'instance initiale)
- L'algorithme divise  $x$  en  $r$  petites instances
- Supposons que la taille de chaque petite instance est  $n/b$ 
  - Souvent nous avons  $b = r$  (le cas le plus fréquent est  $b = r = 2$ ).
  - Mais il arrive parfois que  $r \neq b$ . Par exemple  $b = 2$  et  $r = 3$  ou  $b = 2$  et  $r = 1$  (voir exemple plus loin)

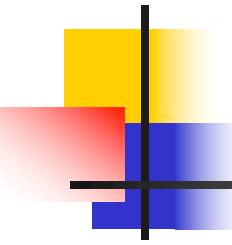


## La récurrence diviser pour régner

- Supposons que  $f(n)$  est le temps requis pour diviser  $x$  en  $r$  petites instances et recombiner les  $r$  solutions  $y_i$  en une solution  $y$  pour  $x$
- Le temps d'exécution  $T(n)$  de l'algorithme est alors donné par la **récurrence diviser pour régner**:

$$T(n) = r T(n/b) + f(n)$$

- On suppose alors que chacune des  $r$  petites instances possède exactement la même taille et ce pour tout appel récursif. Cela n'est possible que lorsque  $n$  est une puissance de  $b$ :  $n = b^k$
- Cependant, lorsque  $f(n)$  possède un ordre de croissance polynomial, le théorème suivant nous informera que l'ordre de croissance de la solution  $T(n)$  de cette récurrence sera donnée par une fonction harmonieuse. Alors, en vertu de la règle de l'harmonie du chapitre 2, nous pourrons en conclure que l'ordre de croissance obtenu pour  $T(n)$  sera valide pour tout  $n$  (et non uniquement pour  $n = b^k$ ).



## Solution de la récurrence diviser pour régner

- La solution (lorsque  $n \rightarrow \infty$ ) de la récurrence diviser pour régner est donnée par le **théorème général**:
  - Si  $T(n)$  obéit à la récurrence  $T(n) = r T(n/b) + f(n)$  pour  $n = b^k$  (avec  $k = 1, 2, \dots$ ), si  $r > 0$ , si  $b > 1$  et si  $f(n) \in \Theta(n^d)$  avec  $d \geq 0$ , alors:

$$T(n) \in \begin{cases} \Theta(n^d) & \text{si } r < b^d \\ \Theta(n^d \log n) & \text{si } r = b^d \\ \Theta(n^{\log_b r}) & \text{si } r > b^d \end{cases}$$

- Si la récurrence est donnée par l'inégalité  $T(n) \leq r T(n/b) + f(n)$ , alors ce théorème s'applique en remplaçant  $\Theta()$  par  $O()$
- Si la récurrence est donnée par l'inégalité  $T(n) \geq r T(n/b) + f(n)$ , alors ce théorème s'applique en remplaçant  $\Theta()$  par  $\Omega()$
- **Preuve:** voir annexe B du manuel (la preuve est facultative).

# Solution de la récurrence diviser pour régner

- Le théorème général s'applique aussi à une forme plus générale de la récurrence

$$T(n) = \sum_{i=1}^r T\left(\frac{n}{b} + c_i\right) + f(n)$$

où  $c_i$  est une constante.

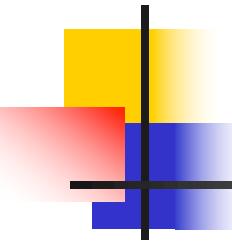
- On remarque que le calcul de  $T(n)$  nécessite  $r$  calculs de la fonction  $T$  et que la variable  $n$  est toujours divisée par  $b$ .
- Exemple:**

$$T(n) = T\left(\frac{n}{3} + 1\right) + T\left(\frac{n}{3} - 1\right) + 2n$$

- Nous avons  $r = 2$ ,  $b = 3$ ,  $d = 1$  donc le cas où  $r < b^d$  s'applique.

$$T(n) \in \Theta(n)$$

- À titre indicatif, le [théorème d'Akra-Bazzi](#) permet de déterminer l'ordre de croissance de relations encore plus complexes.



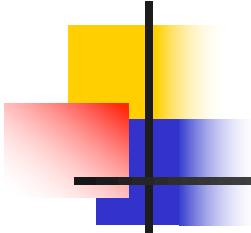
## Observations sur la solution de la récurrence

- Seul le comportement asymptotique ( $n \rightarrow \infty$ ) est spécifié.
- Cette solution asymptotique ne dépend pas de la valeur du seuil.
- La valeur du seuil d'un algorithme diviser pour régner n'affecte donc pas l'ordre de croissance de son temps d'exécution.
- L'ordre de croissance de  $T(n)$  dépend uniquement de  $d$  lorsque  $r \leq b^d$ .
- Si  $r > b^d$ , c'est  $r$  et  $b$  qui affectent l'ordre de croissance de  $T(n)$ .
- Pour utiliser le théorème général, il suffit de vérifier que  $T(n)$  obéi à la récurrence diviser pour régner pour  $n = b^k$  (seulement)
- Nous obtenons une **borne asymptotique exacte** pour  $T(n)$  lorsque  $T(n) = r T(n/b) + f(n)$ . Cette borne dépend de  $\Theta(f(n))$ .
- Nous obtenons une **borne asymptotique supérieure** pour  $T(n)$  lorsque  $T(n) \leq r T(n/b) + f(n)$ . Cette borne dépend de  $O(f(n))$ .
- Nous obtenons une **borne asymptotique inférieure** pour  $T(n)$  lorsque  $T(n) \geq r T(n/b) + f(n)$ . Cette borne dépend de  $\Omega(f(n))$ .

# Le tri fusion

- C'est un exemple classique d'algorithme diviser pour régner
- Pour trier  $A[0..n-1]$ :
  - On divise d'abord  $A$  en deux moitiés  $A[0..\lfloor n/2 \rfloor - 1]$  et  $A[\lfloor n/2 \rfloor .. n-1]$
  - On tri récursivement chacune de ces deux moitiés
  - On fusionne ces deux moitiés triées en un seul tableau trié
  - Le seuil est fixé à  $n = 1$  (un tableau avec un seul élément est toujours trié; il n'y a donc rien à faire dans ce cas)

```
ALGORITHME MergeSort(A[0..n-1])
//Entrée: le tableau A
//Sortie: le tableau A trié
if n > 1
    for i = 0.. $\lfloor n/2 \rfloor - 1$  do B[i]  $\leftarrow$  A[i]
    for i =  $\lfloor n/2 \rfloor .. n-1$  do C[i -  $\lfloor n/2 \rfloor$ ]  $\leftarrow$  A[i]
    MergeSort(B[0.. $\lfloor n/2 \rfloor - 1$ ])
    MergeSort(C[0.. $\lfloor n/2 \rfloor - 1$ ])
    Merge(B, C, A)
    (Delete B; Delete C)
```



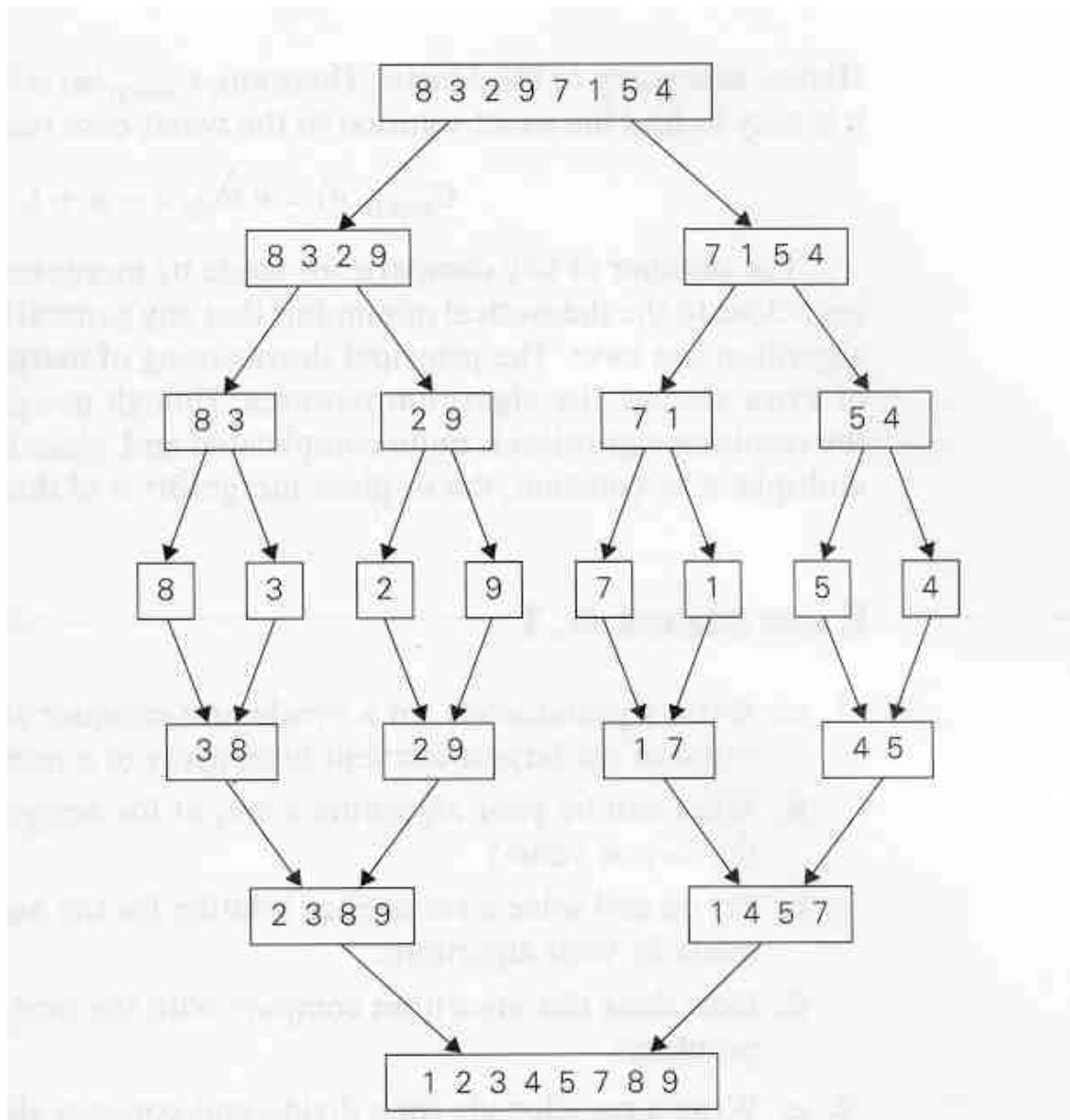
## La procédure de fusion

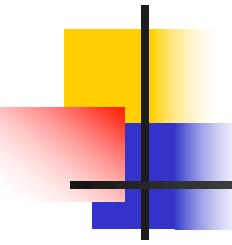
- Deux index pointent initialement sur le premier élément de chaque tableau
- Le plus petit de ces deux éléments pointés est copié à la première place de disponible dans le nouveau tableau
- L'index du plus petit élément est incrémenté de 1
- Nous continuons jusqu'à ce que tous les éléments ont été copiés.

**ALGORITHME** Merge( $B[0..p-1]$ ,  $C[0..q-1]$ ,  $A[0..p+q-1]$ )

```
i ← 0; j ← 0;  
for k = 0..p+q-1 do  
    if (i < p) and (j = q or B[i] < C[j]) then  
        A[k] ← B[i];  
        i ← i + 1;  
    else  
        A[k] ← C[j];  
        j ← j + 1;
```

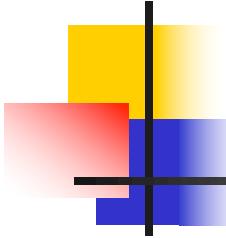
# Exemple illustrant l'algorithme du tri fusion





## Analyse de l'algorithme du tri fusion

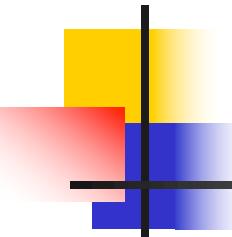
- Peu importe notre choix pour l'opération de base, le nombre d'opérations de base  $C(n)$  effectuées par  $\text{MergeSort}(A[0..n-1])$  sera, dans le pire cas et le meilleur cas, donné par:
  - $C(n) = 2 C(n/2) + C_{\text{merge}}(n) + C_{b-c}(n)$  lorsque  $n = 2^k$
  - où  $C_{\text{merge}}(n)$  est le nombre d'opérations de base effectuées par  $\text{Merge}(B[0..(n/2)-1], C[0..(n/2)-1], A[0..n-1])$
  - et  $C_{b-c}(n)$  est le temps requis pour construire les tableaux additionnels B et C et les détruire
- Il s'agit donc d'une récurrence diviser pour régner avec  $r = b = 2$
- Pour appliquer le théorème général sur  $C(n)$ , nous devons déterminer l'ordre de croissance de  $C_{\text{merge}}(n)$  et de  $C_{b-c}(n)$ .
- Pour  $C_{b-c}(n)$ , nous pouvons choisir l'affectation (le copiage d'un élément à l'autre) comme opération de base.
  - Puisqu'il y a  $n$  affectations, on a  $C_{b-c}(n) = n$ .



## Analyse de l'algorithme du tri fusion (suite)

---

- Pour  $C_{\text{merge}}(n)$  nous pouvons prendre la condition du *if* comme opération de base.
- Elle est exécutée exactement  $n$  fois puisque  $p + q = n$ .
- Alors  $\mathbf{C_{\text{merge}}(n) = n}$ .

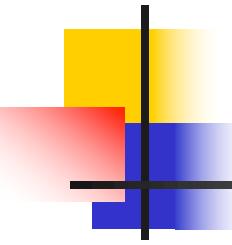


## Analyse de l'algorithme du tri fusion (suite)

- La relation de récurrence pour  $C(n)$  est alors donnée par
  - $C(n) = 2 C(n/2) + f(n)$  avec  $f(n) \in \Theta(n)$  (lorsque  $n = 2^k$ )
- C'est une récurrence diviser pour régner avec  $r = b = 2$  et  $f(n) \in \Theta(n)$ . Donc  $d = 1$  et  $r = b^d$ . Le théorème général nous donne alors:

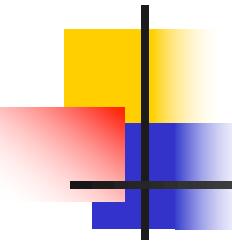
$$C(n) \in \Theta(n \log n)$$

- C'est donc mieux que le  $\Theta(n^2)$  de l'algorithme force brute SelectSort()
- Il est possible de démontrer (voir chap 10) que tout algorithme de tri par comparaison doit nécessairement avoir  $C_{\text{worst}}(n) \in \Omega(n \log n)$
- Le tri fusion est donc **asymptotiquement optimal** en pire cas!
- Cependant, le tri fusion ne tri pas sur place. Il nécessite le stockage de  $\Theta(n)$  éléments additionnels pour les tableaux B et C lorsque l'on détruit B et C après chaque utilisation. (Si l'on ne détruit jamais B et C, ce sera  $\Theta(n \log n)$ : pourquoi?)



## Le tri rapide

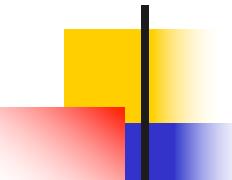
- Le tri rapide est peut-être l'algorithme de tri le plus fréquemment utilisé
  - C'est également un algorithme divisor pour régner
- La première étape de l'algorithme est de **partitionner** un tableau  $A[0..n-1]$  autour d'un élément  $A[s]$  appelé le **pivot**.
- Le tableau  $A[0..n-1]$  est partitionné autour du pivot  $A[s]$  ssi les éléments  $A[0]$  jusqu'à  $A[s-1]$  sont tous  $\leq A[s]$  et les éléments  $A[s+1]$  jusqu'à  $A[n-1]$  sont tous  $\geq A[s]$ .
  - Lorsque cette **partition** est effectuée, l'élément  $A[s]$  occupe sa position finale dans le tableau trié.
- Pour trier le tableau, il suffit alors de répéter (récursivement) cette procédure pour le sous tableau  $A[0..s-1]$  et le sous tableau  $A[s+1..n-1]$ 
  - Pour chaque appel, on partitionne le sous tableau autour d'un élément pivot choisi dans le sous tableau



## Le tri rapide (suite)

```
ALGORITHME QuickSort(A[l..r])
//Entrée: le sous tableau A[l..r] de A[0..n-1]
//Sortie: le sous tableau A[l..r] trié
if l < r
    s ← Partition(A[l..r])
    QuickSort(A[l..s-1])
    QuickSort(A[s+1..r])
```

- La procédure Partition(A[l..r]) retourne la position s du pivot A[s] autour duquel A[l..r] est partitionné
  - Le pivot choisi par Partition(A[l..r]) est le premier élément A[l] = p, ensuite le tableau A[l..r] est partitionné autour de cet élément p
- La procédure parcourt le vecteur A en utilisant deux index i et j tel que  $l \leq j \leq i \leq r$
- Les éléments dans  $A[l..j]$  sont plus petits que le pivot p.
- Les éléments dans  $A[j+1..i]$  sont plus grands que le pivot p.
- Les éléments dans  $A[i+1..r]$  ne sont pas encore traités.



## Le procédure Partition( $A[l..r]$ )

---

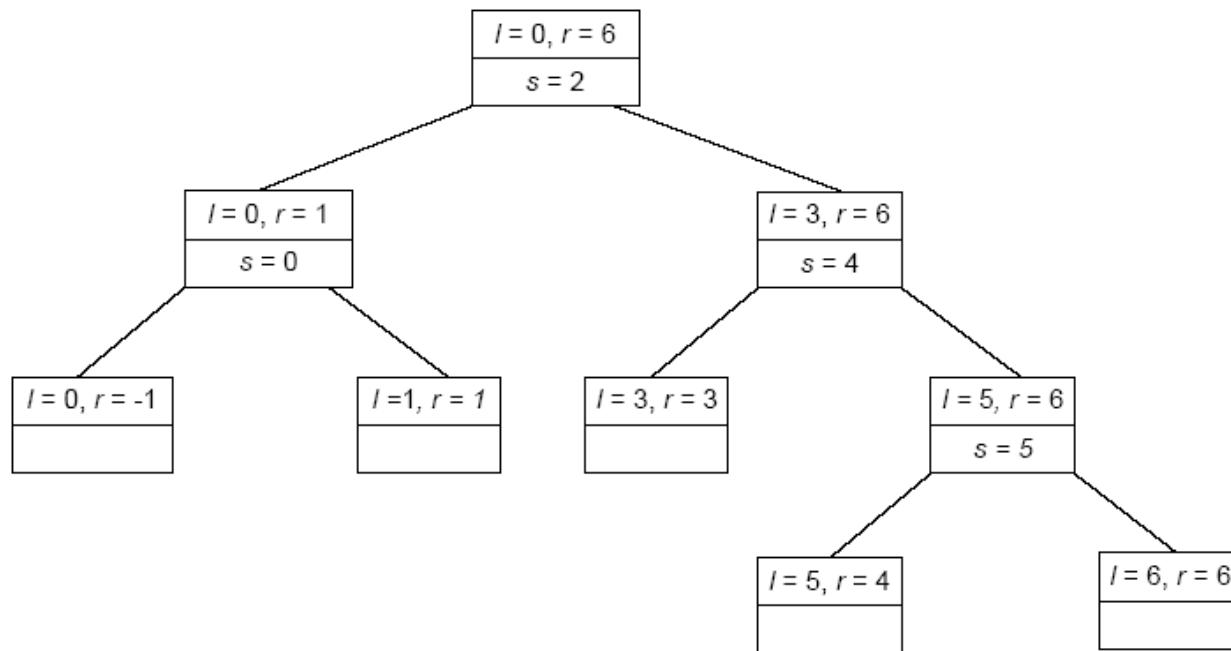
### Algorithme 1 : Partition( $A[l..r]$ )

---

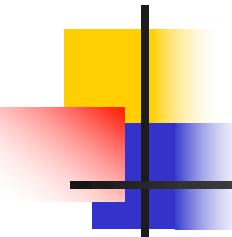
```
// Partitionne le tableau en prenant le premier élément comme pivot
// Entrée : Un sous-tableau de  $A[0..n - 1]$  défini par les bornes  $l$  et  $r$ 
// Sortie : Le sous-tableau partitionné et la position du pivot
 $p \leftarrow A[l]$ 
 $j \leftarrow l$ 
// Invariant :  $A[k] \leq p$  pour  $k$  tel que  $l \leq k \leq j$ 
Pour  $i = l + 1..r$  Faire
    Si  $A[i] < p$  alors
         $j \leftarrow j + 1$ 
        interchange  $A[i]$  et  $A[j]$ 
    // Invariant :  $A[k] > p$  pour  $k$  tel que  $j < k \leq i$ 
interchange  $A[j]$  et  $A[l]$ 
retourner  $j$ 
```

---

# Exemple d'exécution du tri rapide



	0	1	2	3	4	5	6
E	$X^i$	A	M	P	L	$E^j$	
E	E	$A^j$	$M^i$	P	L	X	
A	E	E	M	P	L	X	
A	$E^{ij}$						
A	$E^j$	$E^i$					
A	E						
M		$P^i$	L	$X^j$			
M		$P^i$	$L^j$	X			
M		$L^i$	$P^j$	X			
M		$L^j$	$P^i$	X			
L		M	P	X			
L							
P		$X^{ij}$					
P		$X^j$	$X^i$				
P		X					
							X



## Analyse du tri rapide

- Utilisons la comparaison entre 2 éléments pour l'opération de base.
- Le nombre de comparaisons effectuées par Partition( $A[0..n-1]$ ) est de  $n-1$ .
  - Alors  $C_{\text{partition}}(n) \in \Theta(n)$
- Les meilleurs cas pour le tri rapide sont ceux pour lesquels le pivot est toujours l'élément médian. Donc, pour  $n = 2^k$ , nous avons:
  - $C_{\text{best}}(n) = 2C_{\text{best}}(n/2) + C_{\text{partition}}(n)$
  - Alors:  $C_{\text{best}}(n) \in \Theta(n \log n)$  selon le théorème général ( $b=r=2$ ,  $d=1$ )
- Les pires cas sont, entre autre, ceux pour lesquels le pivot est toujours le plus petit élément du sous tableau.
  - Cela se produit lorsque  $A[0..n-1]$  est déjà trié en ordre croissant.
  - Dans ces pires cas, Quicksort fait un appel récursif sur un tableau vide et un autre sur  $A[1..n-1]$ . Nous avons alors:
    - $C_{\text{worst}}(n) = C_{\text{worst}}(n-1) + C_{\text{partition}}(n)$

## Analyse du tri rapide (suite)

- La méthode des substitutions à rebours donne:
  - $$\begin{aligned} C_{\text{worst}}(n) &= C_{\text{worst}}(n-1) + C_{\text{partition}}(n) \\ &= C_{\text{worst}}(n-2) + C_{\text{partition}}(n-1) + C_{\text{partition}}(n) \\ &\dots \\ &= C_{\text{worst}}(n-i) + C_{\text{partition}}(n-i+1) \dots + C_{\text{partition}}(n) \\ &= C_{\text{worst}}(1) + C_{\text{partition}}(2) \dots + C_{\text{partition}}(n) \quad (\text{pour } i = n-1) \end{aligned}$$
- Puisque  $C_{\text{worst}}(1) = 0$  et que  $C_{\text{partition}}(i) = i - 1$  en pire cas. Nous avons:
$$C_{\text{worst}}(n) = \sum_{i=2}^n (i - 1) = \sum_{i=1}^{n-1} i = \frac{n(n - 1)}{2} \in \Theta(n^2)$$
- Conclusion: le temps d'exécution du tri rapide est  $\Theta(n^2)$  en pire cas et  $\Theta(n \log n)$  en meilleur cas.

## Analyse du cas moyen du tri rapide

- Quel est alors son temps d'exécution en moyenne?
- Supposons, pour simplifier l'analyse, que tous les éléments à trier sont **distincts** et que toutes les  $n!$  configurations possibles des éléments dans  $A[0..n-1]$  sont équiprobables.
- Dans ces circonstances, la partition du tableau  $A[0..n-1]$  peut survenir à chaque position  $s \in \{0..n-1\}$  avec une même probabilité de  $1/n$ .
- Le temps d'exécution moyen sera alors donné par:

$$\begin{aligned} C_{avg}(n) &= \frac{1}{n} \sum_{s=0}^{n-1} [C_{avg}(s) + C_{avg}(n - (s + 1)) + C_{partition}(n)] \\ &= \frac{1}{n} \sum_{s=0}^{n-1} [C_{avg}(s) + C_{avg}(n - (s + 1)) + (n + 1)] \\ &= (n + 1) + \frac{2}{n} \sum_{s=0}^{n-1} C_{avg}(s) \quad \text{car : } \sum_{s=0}^{n-1} C(s) = \sum_{s=0}^{n-1} C(n - 1 - s) \end{aligned}$$

## Analyse du cas moyen du tri rapide (suite)

Alors :

$$nC_{avg}(n) = n(n + 1) + 2 \sum_{s=0}^{n-1} C_{avg}(s)$$

En substituant  $n - 1$  pour  $n$ , on trouve :

$$(n - 1)C_{avg}(n_1) = (n - 1)n + 2 \sum_{s=0}^{n-2} C_{avg}(s)$$

La différence entre ces deux équations donne :

$$nC_{avg}(n) = (n + 1)C_{avg}(n - 1) + 2n$$

En divisant par  $n(n + 1)$ , on trouve :

$$\frac{C_{avg}(n)}{n + 1} = \frac{C_{avg}(n - 1)}{n} + \frac{2}{n + 1}$$

## Analyse du cas moyen du tri rapide (suite)

En substituant  $B(n) = C_{avg}(n)/(n + 1)$ , nous obtenons la récurrence suivante :

$$B(n) = B(n-1) + \frac{2}{n+1} \quad \text{avec : } B(1) = B(0) = 0$$

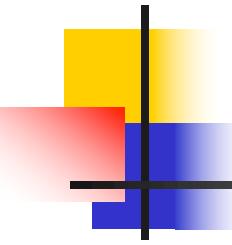
La méthode des substitutions à rebours donne :

$$\begin{aligned} B(n) &= B(n-2) + \frac{2}{n} + \frac{2}{n+1} \\ &\dots \\ &= B(n-i) + \frac{2}{n+2-i} + \dots + \frac{2}{n} + \frac{2}{n+1} \\ &= B(1) + \frac{2}{3} + \frac{2}{4} + \dots + \frac{2}{n+1} \\ &= 0 + 2 \sum_{k=3}^{n+1} \frac{1}{k} \end{aligned}$$

## Analyse du cas moyen du tri rapide (suite)

Or puisque  $1/x$  est non croissant, on a :

$$\begin{aligned} \int_3^{n+2} \frac{1}{x} dx &\leq \sum_{k=3}^{n+1} \frac{1}{k} \leq \int_2^{n+1} \frac{1}{x} dx \\ \Rightarrow |\ln x|_3^{n+2} &\leq \sum_{k=3}^{n+1} \frac{1}{k} \leq |\ln x|_2^{n+1} \\ \Rightarrow \ln(n+2) - \ln 3 &\leq \sum_{k=3}^{n+1} \frac{1}{k} \leq \ln(n+1) - \ln 2 \\ \Rightarrow \ln(n+1) - \ln 3 &\leq \sum_{k=3}^{n+1} \frac{1}{k} \leq \ln(n+1) - \ln 2 \\ \Rightarrow 2(\ln(n+1) - \ln 3) &\leq B(n) \leq 2(\ln(n+1) - \ln 2) \end{aligned}$$

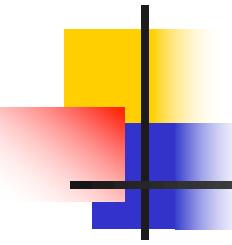


## Analyse du cas moyen du tri rapide (suite)

- Puisque  $B(n) = C_{avg}(n)/(n+1)$ , nous obtenons finalement:

$$\begin{aligned} 2(n + 1)(\ln(n + 1) - \ln 3) &\leq C_{avg}(n) \leq 2(n + 1)(\ln(n + 1) - \ln 2) \\ \Rightarrow C_{avg}(n) &\in \Theta(n \ln n) \end{aligned}$$

- Pour les grandes valeurs de  $n$ , on a :  $C_{avg}(n) \approx 2n \ln(n) \approx 1.38 n \lg(n)$
- La faible valeur de cette constante (1.38) donne un temps d'exécution moyen inférieur au tri par tas (voir + loin) ayant  $C_{worst}(n) \in \Theta(n \log n)$
- Le tri fusion donne  $C_{avg}(n) \approx n \lg(n)$  mais, contrairement au tri rapide, il ne tri pas sur place.
- L'excellente efficacité en moyenne du tri rapide en fait un candidat de choix lorsque le tableau à trier n'est pas déjà quasiment trié



## Multiplication de matrices

- Nous voulons calculer  $C = AB$  où  $A$  et  $B$  sont deux matrices de dimensions  $n \times n$ .
- L'algorithme standard consiste à trouver la valeur  $c_{ij}$  en calculant le produit scalaire entre la  $i^{\text{ème}}$  ligne de  $A$  et la  $j^{\text{ème}}$  colonne de  $B$ .
  - L'opération de base dans le calcul du produit scalaire est la multiplication.
  - Il faut  $n$  multiplications pour calculer le produit scalaire entre deux vecteurs de taille  $n$ .
  - Il y a  $n^2$  produits scalaires à calculer pour un temps de calcul de  $C(n) \in \Theta(n^3)$  dans tous les cas.
- Ainsi, multiplier deux matrices de dimensions  $2 \times 2$  requiert 8 multiplications.
- Il est possible d'appliquer la technique diviser pour régner afin d'obtenir un algorithme plus efficace.

# Algorithme de Strassen

- La multiplication d'une matrice  $2 \times 2$  peut être représentée de la façon suivante.

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix} = \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

- pour

$$m_1 = (a_{00} + a_{11}) \times (b_{00} + b_{11})$$

$$m_2 = (a_{10} + a_{11}) \times b_{00}$$

$$m_3 = a_{00} \times (b_{01} - b_{11})$$

$$m_4 = a_{11} \times (b_{10} - b_{00})$$

$$m_5 = (a_{00} + a_{01}) \times b_{11}$$

$$m_6 = (a_{10} - a_{00}) \times (b_{00} + b_{01})$$

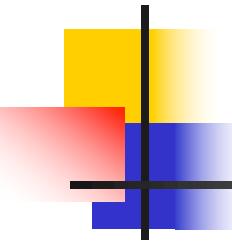
$$m_7 = (a_{01} - a_{11}) \times (b_{10} + b_{11})$$

## Algorithme de Strassen

- La représentation de Strassen nécessite 7 multiplications donnant les 7 nombres  $m_1, m_2, \dots, m_7$ . C'est une multiplication de moins que l'algorithme conventionnel.
- La même technique peut être utilisée pour multiplier des matrices carrés de dimensions arbitraires.
- En effet, une matrice de dimensions  $2n \times 2n$  peut être décomposée en 4 sous-matrices de dimensions  $n \times n$ .

$$\left[ \begin{array}{c|c} C_{00} & C_{01} \\ \hline C_{10} & C_{11} \end{array} \right] = \left[ \begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{10} & A_{11} \end{array} \right] \left[ \begin{array}{c|c} B_{00} & B_{01} \\ \hline B_{10} & B_{11} \end{array} \right]$$

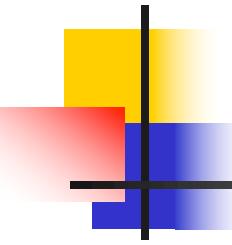
- La relation de Strassen permet donc de multiplier deux matrices de dimensions  $2n \times 2n$  en réduisant le problème à la multiplication de 7 matrices de dimensions  $n \times n$ .
- Si  $n$  n'est pas pair, il est toujours possible d'ajouter une ligne et une colonne de zéros. Cette ligne et cette colonne doit être retranchée du résultat final.



# Analyse de l'algorithme de Strassen

- Prenons les additions et soustractions comme opérations de base.
- Pour diviser l'instance, il faut 6 additions et 4 soustractions de matrices de dimensions  $n / 2 \times n / 2$ .
  - Nous avons donc  $C_{\text{diviser}}(n) \in \Theta(n^2)$
- Pour recombiner les résultats, nous avons 8 additions ou soustractions de matrices  $n / 2 \times n / 2$ .
  - Nous avons donc  $C_{\text{combiner}}(n) \in \Theta(n^2)$
- $C(n) = 7C(n / 2) + C_{\text{diviser}}(n) + C_{\text{combiner}}(n)$ .
- Appliquons le théorème générale
  - Nous avons  $r = 7$  sous instances
  - Nous divisons par  $b = 2$  la taille des instances
  - Puisque  $C_{\text{diviser}}(n) + C_{\text{combiner}}(n) \in \Theta(n^2)$ , nous avons  $d = 2$ .
- Puisque  $r > b^d$ , nous obtenons

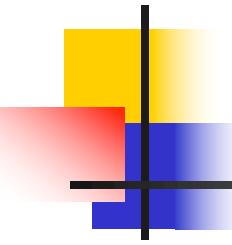
$$C(n) \in \Theta(n^{\log_2 7}) = \Theta(n^{2.8074})$$



## Recherche binaire dans un tableau trié

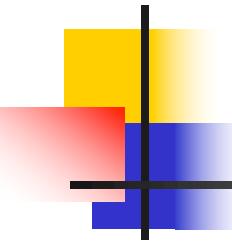
- Le problème est de déterminer si un élément K se trouve dans un tableau trié A[0..n-1]. S'il s'y trouve, on doit obtenir l'index de la position de cet élément.
- L'idée de la recherche binaire est de comparer K à l'élément A[m] situé au milieu du tableau.
  - Si K = A[m] alors on retourne m
  - Si K < A[m] alors on cherche K parmi A[0..m-1]
  - Si K > A[m] alors on cherche K parmi A[m+1..n-1]

```
ALGORITHM BinarySearch(A[0..n-1], K)
    I ← 0; r ← n-1
    while I ≤ r do
        m ← ⌊(I + r)/2⌋
        if K = A[m] then return m
        else if K < A[m] then r ← m – 1
        else I ← m + 1
    return -1
```



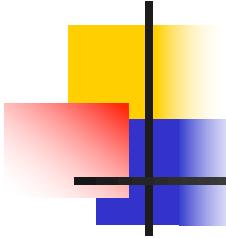
## Analyse de la recherche binaire

- Choisissons le prédicat  $K = A[m]$  pour l'opération de base
- $C_{\text{best}}(n) = 1$  car, dans les meilleurs cas,  $K = A[\lfloor(n-1)/2\rfloor]$
- Pour l'analyse des pires cas, supposons que  $n = 2^k$  car, pour ces cas, nous divisons la taille du tableau exactement par 2 après l'exécution d'une opération de base.
- Ainsi, pour  $n = 2^k$ , nous avons:
  - $C_{\text{worst}}(n) = C_{\text{worst}}(n/2) + 1$
- Alors, le théorème général nous dit que pour  $r = 1$ ,  $b = 2$  et  $d = 0$ , nous avons:
  - $C_{\text{worst}}(n) \in \Theta(\log(n))$
- C'est donc extrêmement rapide!



## Analyse de la recherche binaire (suite)

- Pour connaître exactement le nombre de fois que l'opération de base est effectuée, nous pouvons utiliser la méthode des substitutions à rebours pour  $n = 2^k$ . Cela donne:
  - $C_{\text{worst}}(2^k) = C(2^{k-1}) + 1 = C(2^{k-2}) + 2 \dots = C(2^{k-i}) + i = C(1) + k$
  - Or  $C(1) = 1$ . Alors:  $C_{\text{worst}}(n) = k + 1 = \lg(n) + 1$  (lorsque  $n = 2^k$ )
- Mais ce n'est pas un algorithme diviser pour régner car, à chaque itération, on divise l'instance en deux parties mais on ignore une des parties (pour travailler sur l'autre).
- C'est donc un algorithme **diminuer pour régner** (prochain chapitre).



## Lecture (Levitin)

---

- Chapitre 5 Divide-and-Conquer
  - 5.1 Mergesort
  - 5.2 Quicksort
  - 5.4 Multiplication of Large Integers and Strassen's Matrix Multiplication