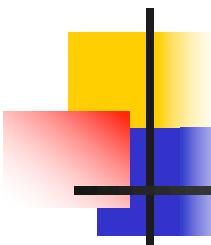


## Chapitre 5

# Diminuer pour régner

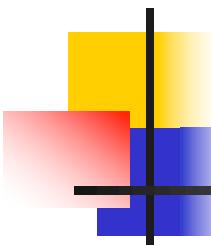


## La technique diminuer pour régner

- Cette technique consiste à exploiter la relation qui existe entre la solution d'une instance et celle d'une instance plus petite.
- Nous solutionnons alors l'instance en « diminuant sa taille ». Cela peut se faire en:
  - Diminuant la taille d'une constante (ex: taille réduite d'une unité)
  - Diminuant la taille par un facteur (ex: taille réduite de moitié)
  - Diminuant la taille d'une quantité variable (à chaque itération)
- Illustrons cette technique sur le **problème d'exponentiation**.
- L'objectif est de calculer  $f(n) = a^n$  pour une constante «  $a$  » donnée.
- Cette fonction se calcule du haut vers le bas à l'aide de la récurrence:

$$f(n) = \begin{cases} f(n - 1) \cdot a & \text{si } n > 1 \\ a & \text{si } n = 1 \end{cases}$$

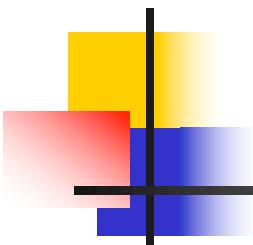
- Ce qui rend explicite le fait que la solution  $f(n)$  est obtenue à l'aide de la solution  $f(n-1)$  d'une instance d'une unité plus petite.



## Solutionner l'exponentiation en diminuant pour régner

- Le nombre  $C(n)$  de multiplications effectuées pour calculer  $f(n)$  est alors donné par:
  - $C(n) = C(n-1) + 1$  avec  $C(1) = 0$
  - Ce qui donne  $C(n) = n-1$  (en pires cas et meilleurs cas).
- **Cette fonction  $f(n)$  se calcule également en diminuant la taille de l'instance de moitié à chaque itération.**
  - En effet, lorsque  $n$  est pair,  $f(n) = (a^{n/2})^2$ . Dans ce cas  $f(n) = f^2(n/2)$  et il suffit alors de multiplier  $f(n/2)$  par lui-même.
  - Si  $n$  est impair alors  $n-1$  est pair. Alors  $f(n) = a \cdot (a^{(n-1)/2})^2$ . Dans ce cas,  $f(n) = a \cdot f^2((n-1)/2)$ .
  - Dans tous les cas,  $f(n)$  se calcule alors à l'aide le la récurrence:

$$f(n) = \begin{cases} f^2(n/2) & \text{si } n \text{ est pair et } > 1 \\ a \cdot f^2((n - 1)/2) & \text{si } n \text{ est impair et } > 1 \\ a & \text{si } n = 1 \end{cases}$$



# Un algorithme diminuer d'un facteur 2 pour régner

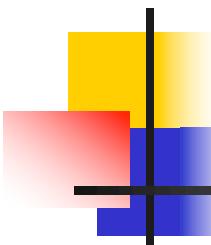
## ALGORITHME Pow1(a, n)

//Entrée: deux entiers a et n  $\geq 1$

//Sortie:  $a^n$

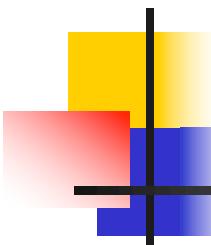
```
if n = 1 return a
temp <- Pow1(a, ⌊n/2⌋)
temp <- temp × temp
if n is even return temp
else return a × temp
```

- Cette dernière récurrence diminuer pour régner est effectuée par l'algorithme Pow1(a,n) ci-haut.
- Notez que chaque appel à Pow1 engendre un seul appel récursif.
- Utilisons **temp** × **temp** pour opération de base. Le nombre C(n) de fois que cette opération est effectuée satisfait la récurrence:
  - $C(n) = 1 + C(\lfloor n/2 \rfloor)$ .
- Lorsque  $n = 2^k$ , C(n) est alors donné (en pires cas et meilleurs cas) par:
  - $C(n) = C(n/2) + 1$  avec  $C(1) = 0$



## Analyse de l'algorithme « diminuer de moitié » pour régner

- La méthode des substitutions à rebours donne alors:
  - $$\begin{aligned} C(2^k) &= C(2^{k-1}) + 1 \\ &= C(2^{k-2}) + 2 \\ &= C(2^{k-i}) + i \\ &= C(2^0) + k \\ &= 0 + k \\ &= \lg(n) \text{ car } 2^k = n \end{aligned}$$
- Alors  $C(n) = \lg(n)$  pour  $n = 2^k$ . Or  $\lg(n) \in \Theta(\log(n))$  et  $\log(n)$  est harmonieuse. Alors  $C(n) \in \Theta(\log(n))$  pour tout  $n$ .
- L'approche « diminuer de moitié » est donc substantiellement plus efficace que l'approche « diminuer d'une unité » car cette dernière nécessitait  $\Theta(n)$  multiplications



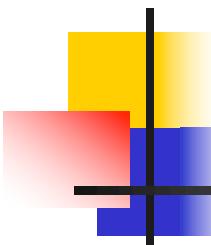
# Solutionner l'exponentiation en divisant pour régner

- Il importe de réaliser que l'approche « diminuer de moitié » n'est pas une approche diviser pour régner.
- En effet, l'approche diviser pour régner appliquée au problème de l'exponentiation consiste à diviser l'instance  $n$  en deux instances distinctes  $\lfloor n/2 \rfloor$  et  $\lceil n/2 \rceil$  et solutionner (régner sur) les **deux** instances!
- Alors, l'approche diviser pour régner calcule  $f(n)$  à l'aide le la récurrence:

$$f(n) = \begin{cases} f(\lfloor n/2 \rfloor) \cdot f(\lceil n/2 \rceil) & \text{si } n > 1 \\ a & \text{si } n = 1 \end{cases}$$

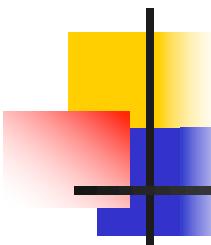
- Cette récurrence est correcte car  $f(\lfloor n/2 \rfloor) \times f(\lceil n/2 \rceil) = a^{\lfloor n/2 \rfloor} \times a^{\lceil n/2 \rceil} = a^{\lfloor n/2 \rfloor + \lceil n/2 \rceil} = a^n = f(n)$ . L'algorithme Pow2 effectue cette récurrence:

```
ALGORITHME Pow2(a, n)
//Entrée: deux entiers a et n ≥ 1
//Sortie: an
if n = 1 return a
return Pow2(a,  $\lfloor n/2 \rfloor$ ) × Pow2(a,  $\lceil n/2 \rceil$ )
```



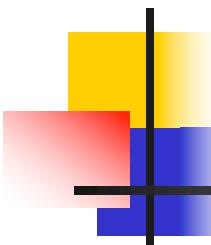
# Analyse de l'algorithme diviser pour régner

- Le nombre  $C(n)$  de multiplications effectuées par Pow2 est donnée par la récurrence:
  - $C(n) = C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + 1$  avec  $C(1) = 0$
  - $C(n) = 2 C(n/2) + 1$  avec  $C(1) = 0$  lorsque  $n = 2^k$
- La méthode des substitutions à rebours donne:
  - $$\begin{aligned} C(2^k) &= 2 C(2^{k-1}) + 1 \\ &= 2 [ 2 C(2^{k-2}) + 1 ] + 1 \\ &= 2^2 C(2^{k-2}) + 2 + 1 \\ &= 2^2 [ 2 C(2^{k-3}) + 1 ] + 2 + 1 \\ &= 2^3 C(2^{k-3}) + 2^2 + 2 + 1 \\ &= 2^i C(2^{k-i}) + 2^{i-1} + 2^{i-2} + \dots + 2 + 1 \\ &= 2^k C(2^0) + 2^{k-1} + 2^{k-2} + \dots + 2 + 1 \\ &= 0 + 2^k - 1 \quad (\text{série géométrique très connue: voir annexe A}) \\ &= n - 1 \quad (\text{car } 2^k = n) \end{aligned}$$
- Alors  $C(n) = n - 1 \in \Theta(n)$



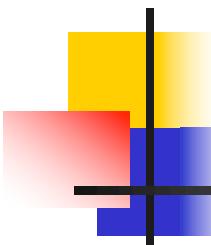
## Conclusions

- Donc, pour le problème de l'exponentiation:
  - L'efficacité de l'algorithme diviser pour régner est nettement inférieure à celle de diminuer pour régner car elle se trouve à calculer récursivement deux fois la même quantité (au lieu d'une seule fois comme c'est le cas pour diminuer pour régner)
- De manière générale, un algorithme « diminuer de moitié pour régner » donne un temps d'exécution logarithmique (en la taille de l'instance) lorsque, en diminuant de moitié, on génère uniquement un nombre constant d'opérations
  - En effet, la récurrence  $C(n) = C(n/2) + r$  avec  $C(1) = 0$  a pour solution  $C(n) = r \lg(n)$ . (pourquoi?)
  - La recherche binaire dans un tableau trié est un autre exemple d'algorithme « diminuer de moitié pour régner »
  - Malheureusement, il est rare que l'on puisse introduire seulement un nombre constant d'opérations en diminuant l'instance de moitié



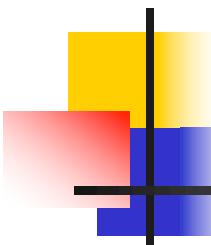
## Diminuer d'une quantité variable pour régner

- L'algorithme d'Euclide nous fourni un exemple d'algorithme dont la taille de l'instance diminue d'une quantité variable à chaque itération
  - Pour trouver le  $\text{pgcd}(m,n)$ , l'algorithme d'Euclide solutionne  $\text{pgcd}(n, m \bmod n)$ .
  - La taille de l'instance est donc diminuée d'une quantité qui dépend des valeurs de  $m$  et  $n$
  - Néanmoins, nous avons vu au chapitre 1 que  $m$  diminuait au moins de moitié à chaque paire d'itérations successive d'Euclide
  - Puisque le nombre d'opérations introduites par cette réduction était constant, cela nous a donné un temps d'exécution en  $O(\log m)$ .
- Nous verrons plus loin d'autres algorithmes « diminuer d'une quantité variable pour régner ».



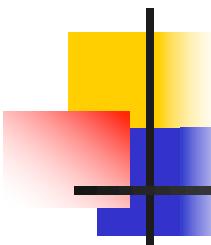
## Le tri par insertion

- C'est un exemple d'algorithme « diminuer de 1 pour régner »
- L'idée est la suivante: pour trier  $A[0..n-1]$ , il suffit de trier  $A[0..n-2]$  et, ensuite, insérer  $A[n-1]$  dans  $A[0..n-2]$  à une position appropriée pour que  $A[0..n-1]$  soit trié
- Puisque  $A[0..n-2]$  est trié, nous pouvons utiliser la recherche binaire pour déterminer l'endroit où nous pouvons insérer  $A[n-1]$ . L'algorithme résultant est appelé le **tri par insertion binaire**.
- Cependant pour insérer un élément  $K$  à la position  $s$  dans un tableau  $A[0..n-2]$ , nous devons faire:
  - $A[j+1] \leftarrow A[j]$  pour  $j = n-2$  jusqu'à  $j = s$
  - et ensuite:  $A[s] \leftarrow K$
- Cela nécessite alors  $\Theta(n)$  affectations, en pire cas, pour insérer un élément  $K$  dans un tableau  $A[0..n-2]$
- Le temps d'exécution du tri par insertion binaire sera alors  $\Theta(n^2)$  en pire cas (voir problèmes série 5).



## Le tri par insertion (suite)

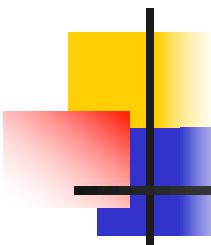
- Mais ça prend uniquement  $\Theta(1)$  opérations pour placer K dans une liste chaînée (car il suffit de réaffecter deux pointeurs)
- Cependant, la recherche binaire est « impossible » à effectuer sur une liste chaînée et il faut alors se résoudre à effectuer une recherche séquentielle nécessitant  $\Theta(n)$  comparaisons en pire cas.
- Le temps d'exécution du tri par insertion binaire sur une liste chaînée sera alors  $\Theta(n^2)$  en pire cas.
  
- Examinons alors le tri par insertion (ordinaire) sur un tableau.
  - Nous abandonnons l'idée de la recherche binaire car il faudra déplacer  $\Theta(n)$  éléments (en pire cas) pour placer A[n-1] à la position désirée (annulant ainsi le bénéfice de la recherche binaire).



## Le tri par insertion (suite)

- L'idée de base du tri par insertion est récursive du « haut vers le bas »:
  - on insère  $A[n-1]$  dans un tableau trié  $A[0..n-2]$ ).
- Cependant l'algorithme est légèrement plus efficace lorsqu'il fonctionne du bas vers le haut de manière non récursive
  - En débutant par  $A[1]$ , on insère  $A[i]$  à la position appropriée dans le sous tableau trié  $A[0..i-1]$ .
  - Cette position est la plus grande valeur de  $s \in \{0, \dots, i\}$  telle que:  
$$A[s-1] \leq A[s] \leq A[i]$$

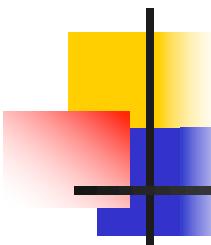
```
ALGORITHM InsertionSort( $A[0..n-1]$ )
    for  $i \leftarrow 1$  to  $n-1$  do
         $v \leftarrow A[i]$  // élément à insérer
         $j \leftarrow i - 1$  // positions possibles d'insertion
        while  $j \geq 0$  and  $A[j] > v$  do
             $A[j+1] \leftarrow A[j]$ 
             $j \leftarrow j - 1$ 
         $A[j+1] \leftarrow v$  // insertion
```



## Analyse du tri par insertion

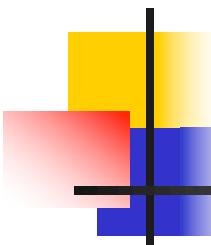
- Comme opération de base, choisissons la comparaison:  $A[j] > v$  .
- Dans les meilleurs cas, ce prédictat  $A[j] > v$  est testé une seule fois pour chaque valeur de  $i \in \{1..n-1\}$  de la boucle **for**.
  - Cela se produit lorsque  $A[i-1] \leq A[i]$  pour tous les  $i$ : c'est-à-dire lorsque le tableau  $A$  est déjà trié.
  - Alors  $C_{best}(n) = n - 1$
- Dans les pires cas, le prédictat  $A[j] > v$  est testé  $i$  fois pour chaque valeur de  $i \in \{1..n-1\}$ .
  - Cela se produit lorsque  $A[i-1] > A[i]$  pour tous les  $i$ : c'est-à-dire lorsque le tableau  $A$  est trié en ordre inverse et que tous ses éléments sont distincts.
  - Alors:

$$C_{worst}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \in \Theta(n^2)$$



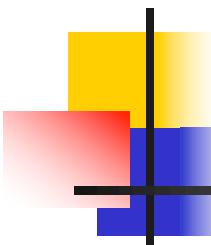
## Analyse du tri par insertion (suite)

- Ainsi, en pire cas, le nombre de comparaisons effectuées par le tri par insertion est identique à celui du tri par sélection
- Cependant, en meilleurs cas (pour les tableaux déjà triés), le nombre de comparaisons effectuées par le tri par insertion ( $n-1$ ) est nettement inférieur à celui du tri par sélection ( $n(n-1)/2$ )
- Pour les tableaux « quasiment triés », la performance du tri par insertion devrait être excellente.
  - Par exemple: si  $A[0..n-k-1]$  est déjà trié et que c'est seulement les  $k$  derniers éléments  $A[n-k..n-1]$  qui sont mal positionnés, le tri par sélection nécessitera seulement  $\Theta(kn)$  comparaisons.
- Par contre, si tous les éléments de  $A$  sont distincts et si tous les ordres possibles sont équiprobables, nous obtenons (voir problèmes série 5):
  - $C_{avg}(n) \in \Theta(n^2)$
  - Ce qui est très désavantageux par rapport au tri rapide



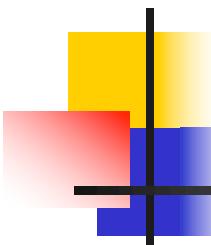
## Le problème de sélection

- Le problème de sélection est celui de trouver le  $k$ ième plus petit élément dans un tableau de  $n$  nombres
- Lorsque  $k=1$  (ou  $k=n$ ), il suffit de balayer le tableau une seule fois pour trouver le plus petit (ou le plus grand) élément.
  - Ces cas triviaux prennent un temps  $\in \Theta(n)$ .
- Un cas plus intéressant est celui de trouver **l'élément médian**. Cet élément est le  $k$ ième plus petit élément pour  $k = \lceil n/2 \rceil$ .
- Pour résoudre ce problème nous pouvons d'abord trier le tableau  $A[1..n]$  et, ensuite, aller directement à l'élément  $A[k]$ .
  - Selon la règle du maximum, l'ordre de croissance du temps requis est déterminé par la partie la plus lente: trier le tableau.
  - Dans ce cas, cet algorithme trier-d'abord-et-accéder, nécessite un temps  $\in \Theta(n \log n)$  si on utilise le tri fusion.



## Le problème de sélection (suite)

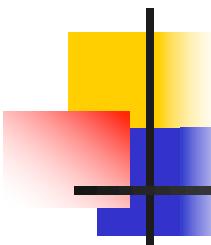
- Essayons de résoudre le problème de trouver le  $k$ ième plus petit élément sans trier d'abord le tableau
- Considérons la procédure  $\text{Partition}(A[1..n])$  utilisée pour le tri rapide.
- Cette procédure utilise l'élément  $A[1]=p$  pour pivot, partitionne le tableau  $A[1..n]$  autour du pivot  $p$  et retourne l'index  $s$  du pivot.
  - $\text{Partition}(A[1..n])$  doit permuter certains éléments de  $A$  pour accomplir cette tâche
- Cela signifie que lorsque  $\text{Partition}(A[1..n])$  retourne, le tableau  $A$  est partitionné comme suit:
  - $A[i] \leq A[s]$  pour  $i \in \{1..s-1\}$
  - $A[s] = p$
  - $A[i] \geq A[s]$  pour  $i \in \{s+1..n\}$
- Donc  $A[s]$  est le  $s$ -ième plus petit élément de  $A[1..n]$



## Le problème de sélection (suite)

- Donc, lorsque Partition( $A[1..n]$ ) retourne:
  - si  $s = k$ ,  $A[s]$  est le  $k$ ième plus petit élément de  $A[1..n]$  (et le problème est résolu).
  - si  $s > k$ , le  $k$ ième plus petit élément de  $A[1..n]$  se trouve dans  $A[1..s-1]$ . Ce sera le  $k$ ième plus petit élément de  $A[1..s-1]$ .
  - si  $s < k$ , le  $k$ ième plus petit élément de  $A[1..n]$  se trouve dans  $A[s+1..n]$ . Ce sera  $(k-s)$ ième plus petit élément de  $A[s+1..n]$ .
- L'algorithme suivant trouve donc le  $k$ ième plus petit élément de  $A[1..n]$ .
  - C'est un algorithme « diminuer d'une quantité variable pour régner ».
  - Contrairement au tri rapide, **un seul des sous tableaux est traité à chaque appel**

**ALGORITHME** SelectionRec( $A[l..r]$ ,  $k$ )  
//Entrée: un sous tableau  $A[l..r]$  et un entier  $k : 1 \leq k \leq r - l + 1$   
//Sortie: le  $k$ ième plus petit élément dans  $A[l..r]$   
 $S \leftarrow \text{Partition}(A[l..r])$   
**if**  $s-l+1 = k$  **return**  $A[s]$   
**if**  $s-l+1 > k$  **return** SelectionRec( $A[l..s-1]$ ,  $k$ )  
**if**  $s-l+1 < k$  **return** SelectionRec( $A[s+1..r]$ ,  $k-s+l-1$ )



## Le problème de sélection (suite)

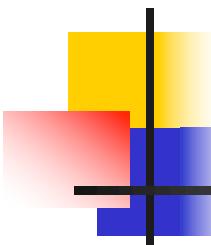
- **Exemple:** trouvez l'élément médian de 4, 1, 10, 9, 7, 12, 8, 2, 15
- Puisque nous avons  $n = 9$  éléments, l'élément médian est le  $\lceil n/2 \rceil$ ième = 5ième plus petit élément. Donc  $k = 5$ .
- Le premier appel à  $\text{Partition}(A[1..9])$ , choisira  $A[1]=4$  pour le pivot et partitionnera A de la manière suivante:

2, 1, **4**, 9, 7, 12, 8, 10, 15

- Nous avons alors  $s = 3 \leq k = 5$ . L'algorithme cherchera alors le (5-3)ième = 2ième plus petit élément dans 9, 7, 12, 8, 10, 15
  - (posons maintenant  $k = 2$ )
- L'appel à  $\text{Partition}$  pour ce sous tableau donnera la partition suivante:

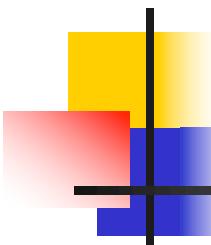
8, 7, **9**, 12, 10, 15

- Nous avons maintenant  $s = 3 \geq 2 = k$ . L'algorithme cherchera alors le 2ième plus petit élément dans 8, 7
- L'appel à  $\text{Partition}$  donnera alors 7, **8** et  $s = 2 = k$ . L'élément médian recherché est donc 8.



## Analyse de l'algorithme SelectionRec(A[1..n])

- En pire cas et meilleur cas nous avons  $C_{\text{partition}}(n) \in \Theta(n)$  (voir le tri rapide, chapitre 4)
- En meilleur cas, un seul appel à Partition est effectué lorsque le pivot coïncide avec le  $k$ ème plus petit élément.
  - Alors  $C_{\text{best}}(n) = C_{\text{partition}}(n) \in \Theta(n)$
- En pire cas, chaque appel récursif de SelectionRec( $A[l..r], k$ ) s'effectue sur un tableau dont la taille a diminuée d'un seul élément.
  - Cela se produit par exemple lorsque  $k=n$  et que le tableau initial  $A[1..n]$  est déjà trié et que tous ses éléments sont distincts.
  - Dans ces pire cas, nous obtenons la même récurrence que celle obtenue lors de l'analyse du tri rapide en pire cas:
    - $C_{\text{worst}}(n) = C_{\text{worst}}(n-1) + C_{\text{partition}}(n)$
    - Donc  $C_{\text{worst}}(n) \in \Theta(n^2)$  d'après notre analyse du tri rapide



## Analyse du temps d'exécution moyen de SelectionRec( $A[1..n]$ )

- En résumé, le temps d'exécution de SelectionRec est donné par:
  - $C_{\text{best}}(n) \in \Theta(n)$
  - $C_{\text{worst}}(n) \in \Theta(n^2)$
- Le pire cas est décevant car l'algorithme trier-d'abord-et-accéder possède un temps d'exécution  $\in \Theta(n \log n)$  en pire cas.
- **Quel est alors le temps d'exécution moyen?**
- Pour effectuer cette analyse, supposons que tous les éléments sont distincts et que toutes les  $n!$  ordres possibles des éléments de  $A[1..n]$  sont équiprobables.
- Dans ces circonstances, la partition du tableau  $A[1..n]$  peut survenir à chaque position  $s \in \{1..n\}$  avec une même probabilité de  $1/n$ .

## Analyse du temps d'exécution moyen de SelectionRec(A[1..n]) ...

- En examinant l'algorithme SelectionRec(A[1..n], k), nous voyons que son temps d'exécution  $C(n)$  est donné par:
  - $C(n) = C_{\text{partition}}(n)$  lorsque  $s = k$
  - $C(n) = C_{\text{partition}}(n) + C(s-1)$  lorsque  $s > k$
  - $C(n) = C_{\text{partition}}(n) + C(n-s)$  lorsque  $s < k$
- Puisque chacune des  $n$  positions possibles pour  $s$  est assignée à la même probabilité de  $1/n$ . Le temps moyen  $C_{\text{avg}}(n)$  est donné par:

$$\begin{aligned}C_{\text{avg}}(n) &= \frac{1}{n} \left[ \sum_{s=1}^{k-1} \left( C_{\text{avg}}(n-s) + C_{\text{partition}}(n) \right) + C_{\text{partition}}(n) \right. \\&\quad \left. + \sum_{s=k+1}^n \left( C_{\text{avg}}(s-1) + C_{\text{partition}}(n) \right) \right] \\&= C_{\text{partition}}(n) + \frac{1}{n} \left[ \sum_{s=1}^{k-1} C_{\text{avg}}(n-s) + \sum_{s=k+1}^n C_{\text{avg}}(s-1) \right] \\&\leq C_{\text{partition}}(n) + \frac{1}{n} \sum_{s=1}^n \max \{ C_{\text{avg}}(n-s), C_{\text{avg}}(s-1) \}\end{aligned}$$

## Analyse du temps d'exécution moyen de SelectionRec(A[1..n]) ...

- Utilisons maintenant le fait que  $C_{\text{partition}}(n) \leq n + 1$  (voir tri rapide) et que  $C_{\text{avg}}(n)$  est une fonction non décroissante de  $n$ . Alors:
  - $\max\{C_{\text{avg}}(n-s), C_{\text{avg}}(s-1)\} = C_{\text{avg}}(\max\{n-s, s-1\})$
- Lorsque  $s \leq \lceil n/2 \rceil$  nous avons:
  - $s - 1 \leq \lceil n/2 \rceil - 1 \leq \lfloor n/2 \rfloor = n - \lceil n/2 \rceil \leq n - s$ 
    - $\max\{n-s, s-1\} = n - s$
- Lorsque  $s > \lceil n/2 \rceil$  nous avons:
  - $s - 1 > \lceil n/2 \rceil - 1 = n - \lfloor n/2 \rfloor - 1 \geq n - \lceil n/2 \rceil - 1 = n - s$ 
    - $\Rightarrow \max\{n-s, s-1\} = s - 1$
- Alors notre borne supérieure sur  $C_{\text{avg}}(n)$  devient:

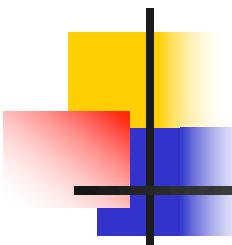
$$C_{\text{avg}}(n) \leq (n+1) + \frac{1}{n} \left[ \sum_{s=1}^{\lceil n/2 \rceil} C_{\text{avg}}(n-s) + \sum_{s=\lceil n/2 \rceil + 1}^n C_{\text{avg}}(s-1) \right]$$

## Analyse du temps d'exécution moyen de SelectionRec(A[1..n]) ...

- La deuxième sommation donne:
  - $C_{avg}(n-1) + C_{avg}(n-2) + \dots + C_{avg}(\lceil n/2 \rceil)$
- La première sommation donne:
  - $C_{avg}(n-1) + C_{avg}(n-2) + \dots + C_{avg}(n - \lceil n/2 \rceil)$
- Or:  $C_{avg}(n - \lceil n/2 \rceil) = C_{avg}(\lfloor n/2 \rfloor)$
- Lorsque  $n$  est pair, la somme de ces deux sommations donne:
  - $2 [ C_{avg}(n-1) + C_{avg}(n-2) + \dots + C_{avg}(\lceil n/2 \rceil) ]$
- Lorsque  $n$  est impair, la somme de ces deux sommations donne:
  - $2 [ C_{avg}(n-1) + C_{avg}(n-2) + \dots + C_{avg}(\lceil n/2 \rceil) ] + C_{avg}(\lfloor n/2 \rfloor)$
- Ainsi, peu importe la valeur de  $n$ , on a toujours:

$$C_{avg}(n) \leq (n+1) + \frac{2}{n} \sum_{s=\lfloor n/2 \rfloor}^{n-1} C_{avg}(s)$$

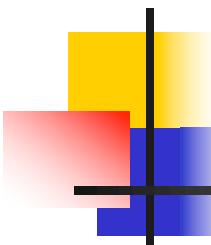
- Notez que cette récurrence pour  $C_{avg}(n)$  est très similaire à celle que nous avions obtenue pour le tri rapide. Il y a cependant deux différences importantes:
  - Nous avons une inégalité ( $\leq$ ) à la place d'une égalité. Cette récurrence nous permettra alors de trouver uniquement une borne supérieure pour  $C_{avg}(n)$
  - La sommation ne comprends pas les termes:  $s = 0, 1, \dots \lfloor n/2 \rfloor - 1$
- Nous avons donc espoir de trouver une borne supérieure qui soit plus faible que celle du tri rapide qui est en  $\Theta(n \log n)$
- Essayons alors de trouver une borne linéaire en  $n$ .
- Pour cela, essayons  $C_{avg}(n) \leq c \times n$  et tentons de trouver une valeur pour la constante  $c$  qui satisfera la récurrence.



## Analyse du temps d'exécution moyen de SelectionRec(A[1..n]) ...

- En utilisant l'hypothèse que  $C_{avg}(s) \leq c \times s$ , nous avons:

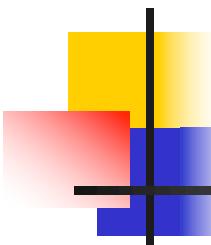
$$\begin{aligned} C_{avg}(n) &\leq (n+1) + \frac{2}{n} \sum_{s=\lfloor n/2 \rfloor}^{n-1} c \times s \\ &= (n+1) + \frac{2c}{n} \left( \sum_{s=1}^{n-1} s - \sum_{s=1}^{\lfloor n/2 \rfloor - 1} s \right) \\ &= (n+1) + \frac{2c}{n} \left( \frac{(n-1)n}{2} - \frac{(\lfloor n/2 \rfloor - 1)\lfloor n/2 \rfloor}{2} \right) \\ &\leq (n+1) + \frac{2c}{n} \left( \frac{(n-1)n}{2} - \frac{(n/2 - 2)(n/2 - 1)}{2} \right) \end{aligned}$$



## Analyse du temps d'exécution moyen de SelectionRec(A[1..n]) ...

- Alors:

$$\begin{aligned}C_{avg}(n) &\leq (n+1) + \frac{2c}{n} \left( \frac{(n-1)n}{2} - \frac{(n/2-2)(n/2-1)}{2} \right) \\&= (n+1) + \frac{2c}{n} \left( \frac{n^2-n}{2} - \frac{n^2/4 - 3n/2 + 2}{2} \right) \\&= (n+1) + \frac{c}{n} \left( \frac{3n^2}{4} + \frac{n}{2} - 2 \right) \\&= (n+1) + c \left( \frac{3n}{4} + \frac{1}{2} - \frac{2}{n} \right) \\&\leq (n+1) + \frac{3cn}{4} + \frac{c}{2} = cn - \left( \frac{cn}{4} - n - 1 - \frac{c}{2} \right)\end{aligned}$$

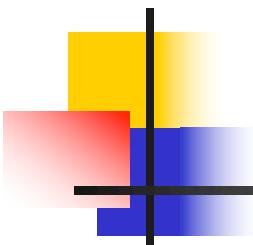


## Analyse du temps d'exécution moyen de SelectionRec(A[1..n]) ...

- Ainsi, nous aurons  $C_{avg}(n) \leq c \times n$  lorsqu'il existe une valeur pour  $c$  et un nombre  $n_0$  tel que:  $cn/4 - n - 1 - c/2 \geq 0 \quad \forall n \geq n_0$
- Si nous choisissons  $c = 8$ , il faut alors satisfaire  $n - 1 - 4 \geq 0 \quad \forall n \geq n_0$ .
  - Ce qui est satisfait pour  $n_0 = 5$ .
- Nous avons alors  $C_{avg}(n) \leq 8n \quad \forall n \geq 5$ . Alors  $C_{avg}(n) \in O(n)$ .
- Or, nous savons que  $C_{avg}(n) \in \Omega(n)$  car  $C_{avg}(n) \geq C_{best}(n) \in \Theta(n)$ .
- Alors:

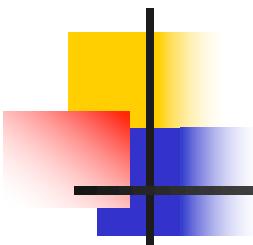
$$C_{avg}(n) \in \Theta(n)$$

- Ainsi, en moyenne, il est plus avantageux d'utiliser SelectionRec que de trier d'abord et accéder directement au  $k$ ième élément.



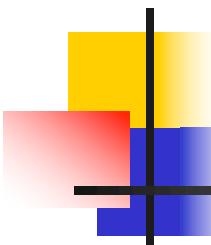
# Introduction aux algorithmes probabilistes

- Considérons l'algorithme  $\text{Partition}(A[l..r])$  utilisé pour le tri rapide et le problème de sélection.
- Le fait d'utiliser toujours le premier élément  $A[l]$  comme pivot, rend  $\text{Partition}(A[l..r])$  (et les algorithmes qui l'utilisent) vulnérable aux particularités des instances
  - C'est ce qui occasionne  $\text{Quicksort}(A[0..n-1])$  et  $\text{SelectionRec}(A[1..n])$  à avoir  $C_{\text{worst}}(n) \in \Theta(n^2)$
- En choisissant le pivot **aléatoirement** parmi  $A[l..r]$  nous rendons  $\text{Partition}(A[l..r])$  moins sensible aux particularités des instances
- Nous allons ici analyser et étudier cette possibilité.
- Pour cela nous devons d'abord avoir un moyen de tirer **au hasard** un nombre entier compris entre  $l$  et  $r$
- Par définition, l'appel à **Uniform( $l,r$ )** nous donnera un tel nombre entier compris entre  $l$  et  $r$  tiré au hasard selon la **distribution uniforme**
  - Cela signifie que chaque nombre compris entre  $l$  et  $r$  possède la même probabilité de  $1/(r-l+1)$  d'être tiré (ou généré)
  - De plus, chaque tirage s'effectue **indépendamment** des autres: un appel à  $\text{Uniform}(l,r)$  n'influence pas les autres appels à  $\text{Uniform}(l,r)$



# Générateurs de nombres pseudo aléatoires

- De tels nombres aléatoires peuvent être obtenus de manière approximative (mais souvent satisfaisante) à l'aide d'un générateur de nombres **pseudo aléatoires**
- De tels générateurs sont fournis dans la librairie de langages de programmation comme Java ou C++
  - La majorité de ces générateurs utilisent la **congruence linéaire** suivante pour générer un entier  $r_i \in \{0..m-1\}$  à partir du nombre précédent  $r_{i-1}$ :
    - $r_i \leftarrow (a \times r_{i-1} + b) \text{ mod } m$  avec une certaine valeur initiale  $r_0$
  - Pour que la séquence des  $r_i$  soit la plus aléatoire possible, nous choisissons les paramètres  $m$ ,  $b$ , et  $a$ , habituellement comme suit:
    - $m = 2^w$  où  $w$  est le nombre de bits utilisés pour coder les entiers (souvent  $w = 32$ )
    - $a$  est un entier compris entre  $0.01m$  et  $0.99m$  tel que  $a \text{ mod } 8 = 5$
    - La valeur de  $b = 1$  donne (semble-t-il) de bons résultats
    - $r_0$  est arbitraire (ex: la date et l'heure)



## Algorithmes probabilistes (ou randomisés)

- Un **algorithme probabiliste (ou randomisé)** est un algorithme qui laisse au hasard le choix de certaines décisions
  - Ex: le choix de l'élément parmi  $A[l..r]$  qui sera utilisé comme pivot
- Pour analyser un algorithme probabiliste, on suppose que notre générateur de nombres pseudo aléatoires nous donne des nombres qui sont vraiment aléatoires
  - Nous faisons donc cette hypothèse pour  $\text{Uniform}(l, r)$
- Le pseudo code de la version randomisée de  $\text{Partition}(A[l..r])$  se trouve à la page suivante.
- Le nombre d'opérations introduites par l'ajout de la randomisation est relativement faible et ne croît pas avec le nombre d'éléments du tableau (c'est-à-dire la taille de l'instance)
- Par contre si l'on désire prendre toujours (de manière déterministe) la meilleure décision, il faudrait choisir l'élément médian pour le pivot
  - Cela est (en général) trop coûteux et est une décision insensée lorsque l'on utilise  $\text{Partition}()$  pour trouver l'élément médian...

# Version randomisée de Partition( $A[l..r]$ )

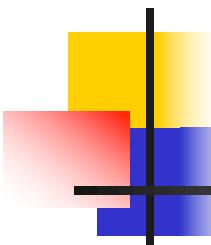
---

## Procédure PartitionRand ( $A[l..r]$ )

---

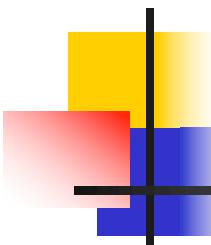
```
// Partitionne le tableau en prenant le premier élément comme pivot
// Entrée : Un sous-tableau de  $A[0..n - 1]$  défini par les bornes  $l$  et  $r$ 
// Sortie : Le sous-tableau partitionné et la position du pivot
s ← Uniform( $l, r$ )
swap( $A[l], A[s]$ ) // Le reste du code est inchangé
p ←  $A[l]$ 
j ←  $l$ 
// 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  $A[j]$ 
    Fin Si
    // Invariant :  $A[k] > p$  pour  $k$  tel que  $j < k \leq i$ 
interchange  $A[j]$  et  $A[l]$ 
Retourner  $j$ 
```

---



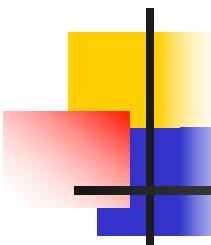
## Le temps d'exécution d'un algorithme probabiliste

- La randomisation d'un algorithme modifie fondamentalement notre manière de calculer son temps d'exécution.
- En effet, contrairement aux algorithmes déterministes, le temps d'exécution d'un algorithme probabiliste exécutant sur une instance  $x$  dépendra non seulement de l'instance  $x$  mais des choix aléatoires effectués par l'algorithme.
  - i.e.: si on exécute deux fois un algorithme probabiliste sur la même instance  $x$ , les temps d'exécutions différeront (d'une fois à l'autre) car les nombres aléatoires utilisés par l'algorithme au cours des deux exécutions seront différents
  - Par contre, si on exécute deux fois un algorithme déterministe sur la même instance  $x$ , les temps d'exécutions seront identiques
- Cette différence fondamentale entre les algorithmes déterministes et probabilistes nous force à adopter une autre définition (plus générale) du temps d'exécution



## Le temps d'exécution d'un algorithme probabiliste (suite)

- Considérons un algorithme probabiliste exécutant sur une instance  $x$ 
  - Son temps d'exécution dépendra de  $x$  et des nombres aléatoires utilisés par cet algorithme au cours de son exécution sur  $x$
- Désignons par  $\rho$  la collection de nombres aléatoires utilisés par cet algorithme au cours de son exécution sur  $x$
- Désignons par  $C(x,\rho)$  le temps d'exécution d'un algorithme probabiliste sur l'instance  $x$  lorsque la collection de nombres aléatoires est  $\rho$
- Pour obtenir une mesure du temps d'exécution qui ne dépende que de  $x$  (et non de  $\rho$ ) il semble que le choix le plus raisonnable est de faire l'**espérance** de  $C(x,\rho)$  sur les différents tirages de  $\rho$ 
  - Supposons que les nombres aléatoires sont tous des entiers et utilisons  $p(\rho)$  pour désigner la probabilité d'obtenir  $\rho$

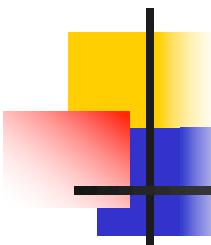


## Le temps d'exécution d'un algorithme probabiliste (suite)

- **Le temps d'exécution  $C(x)$  d'un algorithme probabiliste sur une instance  $x$  est alors défini par l'espérance:**

$$C(x) = E_{\rho} C(x, \rho) = \sum_{\rho} p(\rho) C(x, \rho)$$

- (la somme s'effectue sur tous les  $\rho$  réalisables)
- Puisque c'est une espérance sur  $\rho$ , ce temps d'exécution  $C(x)$  est appelé le **temps d'exécution attendu** pour l'instance  $x$
- Ainsi  $C(x)$  est le temps d'exécution que l'on s'attend d'obtenir si on exécute l'algorithme plusieurs fois sur l'instance  $x$  et que l'on fait ensuite la moyenne des temps d'exécution obtenus



## Le temps d'exécution d'un algorithme probabiliste (suite)

- Soit  $|x|$  la taille de l'instance  $x$  et  $P_n(x)$  la probabilité d'obtenir l'instance  $x$  ( sachant que  $|x| = n$ )
- Pour un algorithme déterministe nous avions:

$$C_{best}(n) = \min_{x:|x|=n} C(x)$$

$$C_{worst}(n) = \max_{x:|x|=n} C(x)$$

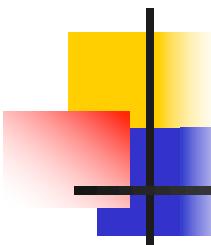
$$C_{avg}(n) = \mathbb{E}_{x:|x|=n} C(x) = \sum_{x:|x|=n} P_n(x) C(x)$$

- Pour un algorithme probabiliste nous aurons alors:

$$\mathcal{C}_{best}(n) = \min_{x:|x|=n} \mathbb{E}_\rho C(x, \rho) = \min_{x:|x|=n} \sum_\rho p(\rho) C(x, \rho)$$

$$\mathcal{C}_{worst}(n) = \max_{x:|x|=n} \mathbb{E}_\rho C(x, \rho) = \max_{x:|x|=n} \sum_\rho p(\rho) C(x, \rho)$$

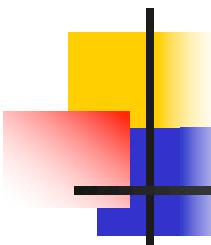
$$\mathcal{C}_{avg}(n) = \mathbb{E}_{x:|x|=n} \mathbb{E}_\rho C(x, \rho) = \sum_{x:|x|=n} P_n(x) \sum_\rho p(\rho) C(x, \rho)$$



## Le temps d'exécution d'un algorithme probabiliste (suite)

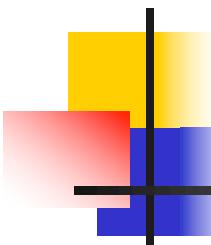
- Ainsi,  $C_{\text{worst}}(n)$ ,  $C_{\text{best}}(n)$  et  $C_{\text{avg}}(n)$  sont respectivement les temps d'exécution attendus sur une instance de taille  $n$  en pire cas, en meilleur cas, et en moyenne.
- Pour le calcul de chacune de ces quantités il faut d'abord effectuer l'espérance sur  $\rho$  de  $C(x,\rho)$
- Effectuons cette espérance pour l'algorithme probabiliste SelectionRecRand( $A[1..n]$ ,  $k$ ) ci-dessous:

**ALGORITHME** SelectionRecRand( $A[l..r]$ ,  $k$ )  
//Entrée: un sous tableau  $A[l..r]$  et un entier  $k : 1 \leq k \leq r - l + 1$   
//Sortie: le  $k$ ième plus petit élément dans  $A[l..r]$   
 $S \leftarrow \text{PartitionRand}(A[l..r])$   
**if**  $s-l+1 = k$  **return**  $A[s]$   
**if**  $s-l+1 > k$  **return** SelectionRecRand( $A[l..s-1]$ ,  $k$ )  
**if**  $s-l+1 < k$  **return** SelectionRecRand( $A[s+1..r]$ ,  $k-s+l-1$ )



## Calcul du temps d'exécution attendu pour l'algorithme SelectionRecRand()

- Considérons n'importe quelle instance  $A[1..n]$  où tous les éléments de  $A[1..n]$  sont distincts
- Puisque  $\text{PartitionRand}(A[1..n])$  choisit au hasard (et uniformément) un élément pivot parmi les  $n$  éléments de  $A$ , la partition résultante sera effectuée en position  $s$  avec une probabilité identique pour toutes les  $n$  positions possibles.
  - Alors  $\Pr(s=i) = 1/n$  pour tous les  $i \in \{1..n\}$
- Alors avec probabilité  $1/n$  nous aurons  $s = k$  et le temps d'exécution attendu  $C(A[1..n])$  sera donné par  $C_{\text{partition}}(n)$ 
  - $C_{\text{partition}}(A[1..n]) = C_{\text{partition}}(n)$  et dépend uniquement de la taille  $n$
- Pour chaque valeur de  $s > k$ ,  $C(A[1..n])$  sera donné par  $C(A[1..s-1]) + C_{\text{partition}}(n)$
- Pour chaque valeur de  $s < k$ ,  $C(A[1..n])$  sera donné par  $C(A[s+1..n]) + C_{\text{partition}}(n)$



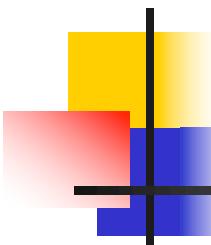
## Calcul du temps d'exécution attendu pour l'algorithme SelectionRecRand() (suite)

- Le temps d'exécution attendu  $C(A[1..n])$  est alors donné par:

$$\begin{aligned} C(A[1..n]) = \frac{1}{n} & \left[ \sum_{s=1}^{k-1} \left( C(A[s+1..n]) + C_{\text{partition}}(n) \right) + C_{\text{partition}}(n) \right. \\ & \left. + \sum_{s=k+1}^n \left( C(A[1..s-1]) + C_{\text{partition}}(n) \right) \right] \end{aligned}$$

- Puisque  $C_{\text{partition}}(n)$  dépend uniquement de  $n$  (la taille de  $A[1..n]$ ), la solution  $C(A[1..n])$  de la récurrence ci-dessus doit uniquement dépendre de  $n$ .
- Alors  $C(A[1..n]) = C(n) = C_{\text{worst}}(n) = C_{\text{best}}(n)$  (**indépendant de l'instance**)
- Le temps d'exécution attendu sera donc identique en pire cas et meilleur cas.
- La récurrence ci-dessus devient alors:

$$\begin{aligned} C(n) = \frac{1}{n} & \left[ \sum_{s=1}^{k-1} \left( C(n-s) + C_{\text{partition}}(n) \right) + C_{\text{partition}}(n) \right. \\ & \left. + \sum_{s=k+1}^n \left( C(s-1) + C_{\text{partition}}(n) \right) \right] \end{aligned}$$

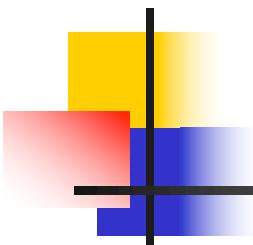


## Calcul du temps d'exécution attendu pour l'algorithme SelectionRecRand() (suite)

- Or cette récurrence est identique à celle que nous avions obtenue pour le temps d'exécution moyen  $C_{avg}(n)$  de l'algorithme SelectionRec() non randomisé
- Nous avions uniquement obtenu la borne supérieure  $C_{avg}(n) \in O(n)$  de la solution de cette récurrence.
- Nous pouvons donc conclure que  $C(n) \in O(n)$ .
- Or, selon notre récurrence, il est clair que  $C(n) \geq C_{partition}(n) \in \Theta(n)$
- Donc  $C(n) \in \Omega(n)$ . Alors:

**$C(n) \in \Theta(n)$  (en pire cas et meilleur cas)**

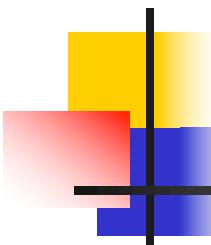
- Or nous avons  $C_{worst}(n) \in \Theta(n^2)$  pour la version déterministe du même algorithme
- Cet exemple illustre bien la puissance additionnelle que peut nous fournir la randomisation (mais c'est le temps d'exécution **attendu**)



## Calcul du temps d'exécution attendu pour l'algorithme QuickSortRand()

```
ALGORITHME QuickSortRand(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 ← PartitionRand(A[l..r])
    QuickSortRand(A[l..s-1])
    QuickSortRand(A[s+1..r])
```

- Considérez la version randomisée de l'algorithme du tri rapide obtenu en remplaçant Partition() par PartitionRand()
- Considérons n'importe quelle instance telle que A[0..n-1] contiennent uniquement des éléments distincts
- Avec probabilité de  $1/n$  pour chaque valeur de  $s$ , le temps d'exécution  $C(A[0..n-1])$  est donné par:
  - $C(A[0..n-1]) = C(A[0..s-1]) + C(A[s+1..n-1]) + C_{\text{partition}}(n)$



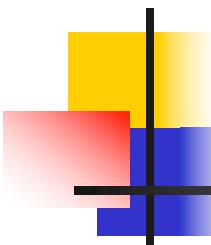
## Calcul du temps d'exécution attendu pour l'algorithme QuickSortRand() (suite)

- Le temps d'exécution attendu  $C(A[0..n-1])$  sera donc donné par:

$$C(A[0..n-1]) = \frac{1}{n} \sum_{s=0}^{n-1} \left[ C(A[0..s-1]) + C(A[s+1..n-1]) + C_{partition}(n) \right]$$

- Puisque  $C_{partition}(n)$  dépend uniquement de  $n$ , la solution  $C(A[0..n-1])$  de la récurrence ci-dessus doit uniquement dépendre de  $n$ .
- Alors  $C(A[0..n-1]) = C(n) = C_{worst}(n) = C_{best}(n)$
- La récurrence ci-dessus devient alors:

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

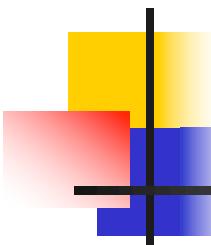


## Calcul du temps d'exécution attendu pour l'algorithme QuickSortRand() (suite)

- Or, cette récurrence est identique à celle que nous avions obtenue pour le temps d'exécution moyen  $C_{avg}(n)$  de l'algorithme QuickSort() non randomisé
  - Nous avions obtenu  $C_{avg}(n) \in \Theta(n \log n)$
- Le temps d'exécution attendu  $C(n)$  sera alors donné par:

$$C(n) \in \Theta(n \log n) \text{ (en pire cas et meilleur cas)}$$

- Or nous avons  $C_{worst}(n) \in \Theta(n^2)$  pour la version déterministe du même algorithme
- Encore une fois, la randomisation nous a permis d'améliorer le temps d'exécution en pire cas
  - Mais il s'agit du temps d'exécution attendu
- Beaucoup considèrent que la version randomisé du tri rapide est l'algorithme à privilégier (en général) pour le triage de gros tableaux



# Lecture (Levitin)

---

- Chapitre 4 Decrease-and-Conquer
  - 4.1 Insertion Sort
  - 4.4 Decrease-by-a-Constant-Factor Algorithms (Binary Search)
  - 4.5 Variable-Size-Decrease Algorithms
    - Computing a Median and the Selection Problem