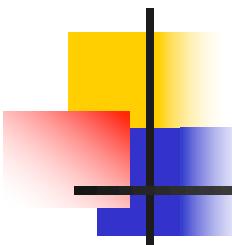


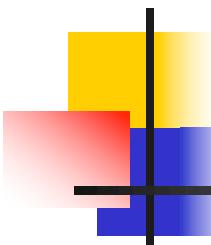
Chapitre 2

Fondements de l'analyse des algorithmes



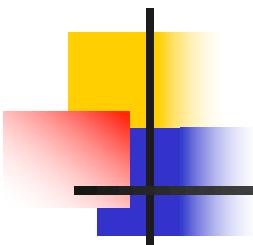
Aperçu du chapitre

- Nous présenterons d'abord le **cadre** que nous utiliserons pour analyser l'efficacité des algorithmes. Les points importants sont:
 - Comment exprimer la **taille des instances** d'un problème
 - Comment exprimer le **temps d'exécution** d'un algorithme d'une manière qui soit indépendante de son implémentation et de la machine utilisée pour exécuter l'algorithme
 - L'importance de **l'ordre de croissance** du temps d'exécution d'un algorithme
 - Les notions d'analyse en **pire cas**, **meilleur cas** et **cas « moyen »**.
- Ensuite nous introduirons la **notation asymptotique** qui sera utilisée pour exprimer l'ordre de croissance du temps d'exécution d'un algorithme en fonction de la taille des instances.
- Finalement, nous examinerons les méthodes d'analyse des algorithmes récursifs et non récursifs.



Efficacité d'utilisation des ressources

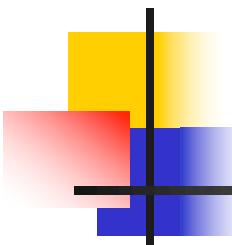
- « Analyser un algorithme » signifie habituellement examiner et étudier un algorithme en vue de déterminer rigoureusement son efficacité
- Un algorithme est **efficace** lorsqu'il utilise peu de ressources
- Les deux principales ressources habituellement utilisées par un algorithme sont:
 - L'espace mémoire
 - Le temps d'exécution (le temps requis pour obtenir une réponse)
- Le temps d'exécution est habituellement l'élément déterminant (le facteur limitatif) de la performance d'un algorithme.
- Dans ce cours, nous nous intéressons donc presque exclusivement au temps d'exécution requis par les algorithmes
 - Mais les méthodes que nous introduirons (ex.: notation asymptotique) s'appliquent également à l'analyse des autres ressources utilisées (comme l'espace mémoire)



Temps d'exécution et opérations élémentaires

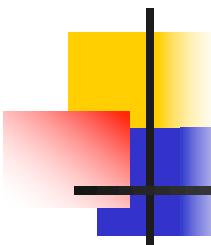
Nous désirons exprimer le **temps d'exécution d'un algorithme** de manière à ce que cette expression caractérise l'**algorithme** et non pas l'implémentation, le langage de programmation ou la machine utilisée.

- Ceci est possible en vertu du **principe d'invariance**: deux implémentations différentes du même algorithme ont un temps (réel) d'exécution qui ne diffère que d'une constante multiplicative
- Nous définissons alors le **temps d'exécution d'un algorithme** (sur une instance donnée) comme étant le nombre d'opérations élémentaires effectuées par l'algorithme (sur cette instance)
- Une **opération élémentaire** est une opération dont le temps d'exécution est majoré (c'est-à-dire borné supérieurement) par une constante qui dépend uniquement de l'implémentation, du langage de programmation ou de la machine utilisée.
 - Cette constante doit être la même pour toutes les instances possibles du problème
- **Chaque opération élémentaire comptera donc pour une seule unité de temps** (quelle que soit cette opération élémentaire)



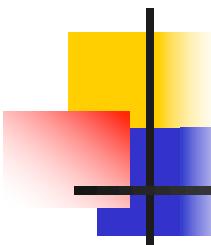
Opérations élémentaires

- Exemples d'opérations élémentaires:
 - Opération arithmétique (addition, soustraction, multiplication...) entre deux nombres lorsque la taille de ces nombres est bornée par une constante k . Ex:
 - Entiers occupant k bits (ou occupant au plus k bits)
 - Réels points flottants codés sur k bits (ou occupant au plus k bits)
 - Comparaison entre deux objets de taille bornée. Ex:
 - Caractères codés sur k bits
 - Nombres codés sur k bits (ou utilisant au plus k bits)
 - Affectations d'une valeur à une variable (utilisant au plus k bits)
- En résumé: une opération élémentaire implique toujours la manipulation d'un certain nombre « d'objets ». Il faut alors que
 - Le nombre d'objets en cause soit majoré par une constante
 - La taille de chaque objet soit majorée par une constante
 - Ces constantes ne doivent pas dépendre de la taille de l'instance



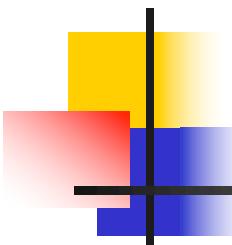
La taille d'une instance

- L'entrée d'un algorithme est une **instance**: c.-à-d. un exemplaire du problème que l'algorithme tente de solutionner
 - La sortie est la solution du problème pour cette instance
- Nous désirons exprimer le temps d'exécution d'un algorithme (c.-à-d. son nombre d'opérations élémentaires effectuées) en fonction de la taille de l'instance
- La **taille de l'instance** est la taille de l'espace mémoire occupée par l'instance exprimée en unités «naturelles » pour le problème en cause.
Exemples:
 - Pour un problème de tri
 - la taille de l'instance est le nombre d'éléments à trier (peu importe la taille des éléments à trier pourvu qu'elle soit fixe)
 - Pour le problème de calculer le produit de deux matrices $n \times n$
 - La taille de cette instance est le nombre d'éléments donc $2n^2$.
 - Mais on préfère généralement exprimer le temps d'exécution en fonction de n .



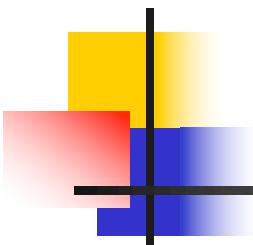
La taille d'une instance (suite)

- Autre exemple:
 - Pour le problème de déterminer $\text{pgcd}(m,n)$ le plus grand commun diviseur entre deux entiers m et n
 - La taille de l'instance est 2 lorsque chaque entier utilise toujours au plus k bits (exemple $k=32$)
 - Mais, dans ce cas, on ne permet pas à la taille des instances de varier. Le problème que l'algorithme tente de solutionner est alors trivial...
 - La taille de cette instance est $\lceil \log_b(m+1) \rceil + \lceil \log_b(n+1) \rceil$ lorsque la taille de m et n n'est pas bornée et que m et n sont exprimés en base b (voir exercices série 1).
 - Le choix de la base modifie la taille seulement par un facteur multiplicatif, car $\log_a(n) = \log_a(b) \times \log_b(n)$
 - (pourvu que a et b soient tous les deux supérieurs à 1)



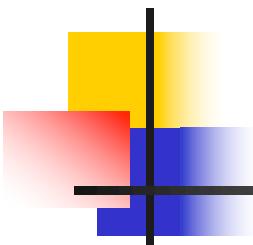
Opération de base d'un algorithme

- Nous désirons exprimer le temps d'exécution d'un algorithme en fonction de la taille de l'instance
- Pour cela il faut, en principe, compter le nombre d'opérations élémentaires effectuées par l'algorithme
 - Ceci nous donnera le temps d'exécution (à une constante près)
- Une autre façon de procéder est d'identifier une **opération de base** (parfois appelée **instruction baromètre**) pour l'algorithme
- Une opération de base d'un algorithme est une opération élémentaire qui, à une constante près, est effectuée au moins aussi souvent que n'importe quelle autre opération élémentaire de l'algorithme
 - Nous ne sommes pas obligés de choisir une opération qui est exécutée au moins aussi souvent que n'importe quel autre. Il suffit qu'elle le soit à une constante près (qui ne croît pas avec la taille de l'instance)



Opération de base d'un algorithme (suite)

- Il est généralement assez simple d'identifier une opération de base pour un algorithme, car la nature du problème qu'il tente de résoudre nous suggère souvent l'opération contribuant le plus au temps d'exécution
- Exemples:
 - Pour les algorithmes de tri (par comparaison), l'opération de base est la comparaison de deux valeurs (ou de deux clés)
 - Pour la multiplication de matrices, l'opération de base est la multiplication de deux nombres
 - Pour l'algorithme d'Euclide qui détermine le plus grand commun diviseur entre deux entiers m et n , l'opération de base est $m \bmod n$
 - Et plusieurs autres exemples à venir dans le cours ...



Temps d'exécution et opération de base

- Soit $C(n)$ le nombre de fois que l'opération de base (choisie) est effectuée par l'algorithme sur une instance de taille n
- Soit $T(n)$ le nombre d'opérations élémentaires effectuées par l'algorithme sur une instance de taille n (c.-à-d. le temps d'exécution)
- Puisque chaque opération élémentaire i présente dans l'algorithme est exécutée au plus $k_i C(n)$ fois (par la définition d'une opération de base)
- Et puisque tout algorithme possède un nombre fini R d'opérations qui ne varie pas avec la taille de l'instance, nous avons:

$$C(n) \leq T(n) \leq \sum_{i=1}^R k_i C(n) = K \times C(n)$$

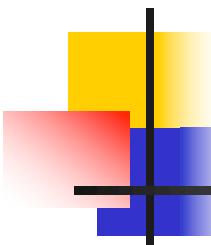
- Où K est une constante qui ne varie pas avec la taille n de l'instance.
- **Le temps d'exécution $T(n)$ de tout algorithme est donc donné par $C(n)$ à une constante près**

L'ordre de croissance du temps d'exécution

- Pour déterminer le temps d'exécution d'un algorithme sur une instance de taille n , il suffit alors de calculer $C(n)$
- Le tableau suivant met en évidence l'importance du comportement asymptotique de $C(n)$: son ordre de croissance lorsque $n \rightarrow \infty$

| n | $\log_2 n$ | n | $n \log_2 n$ | n^2 | n^3 | 2^n | $n!$ |
|--------|------------|--------|------------------|-----------|-----------|---------------------|----------------------|
| 10 | 3.3 | 10^1 | $3.3 \cdot 10^1$ | 10^2 | 10^3 | 10^3 | $3.6 \cdot 10^6$ |
| 10^2 | 6.6 | 10^2 | $6.6 \cdot 10^2$ | 10^4 | 10^6 | $1.3 \cdot 10^{30}$ | $9.3 \cdot 10^{157}$ |
| 10^3 | 10 | 10^3 | $1.0 \cdot 10^4$ | 10^6 | 10^9 | | |
| 10^4 | 13 | 10^4 | $1.3 \cdot 10^5$ | 10^8 | 10^{12} | | |
| 10^5 | 17 | 10^5 | $1.7 \cdot 10^6$ | 10^{10} | 10^{15} | | |
| 10^6 | 20 | 10^6 | $2.0 \cdot 10^7$ | 10^{12} | 10^{18} | | |

Table 2.1 Values (some approximate) of several functions important for analysis of algorithms

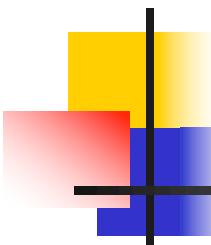


Remarques sur la croissance des fonctions

- La fonction n croît beaucoup plus rapidement que $\log(n)$
- Lorsque l'on double la valeur de n :
 - $\lg(n) \rightarrow \lg(2) + \lg(n) = 1 + \lg(n)$
 - $n \rightarrow 2n$
 - $n^2 \rightarrow 4n^2$
 - $n^3 \rightarrow 8n^3$
 - $2^n \rightarrow (2^n)^2$
- La fonction $n!$ croît beaucoup plus rapidement que l'ordre 2^n
 - Ceci est mis en évidence par la formule de Stirling:

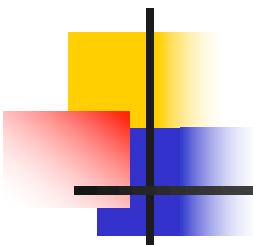
$$n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \quad \text{lorsque } n \rightarrow \infty$$

- Un algorithme dont le temps d'exécution croît exponentiellement rapidement (ou plus) peut généralement être utilisé uniquement sur de très petites instances.



Pires cas et meilleurs cas

- Le temps d'exécution d'un algorithme est donné par $C(n)$: le nombre de fois que l'opération de base est exécutée sur une instance de taille n .
- Pour la majorité des algorithmes, $C(n)$ ne dépend pas uniquement de la taille n de l'instance, mais de l'instance elle-même
 - Dans ces circonstances, il existe plusieurs instances différentes (mais de même taille n) donnant un temps d'exécution différent.
 - Dans ces circonstances, il convient d'identifier:
 - Le(s) pire(s) cas (de taille n) dont le temps d'exécution sera donné par $C_{\text{worst}}(n)$
 - Le(s) meilleur(s) (de taille n) cas dont le temps d'exécution sera donné par $C_{\text{best}}(n)$
 - Ainsi nous aurons: $C_{\text{best}}(n) \leq C(n) \leq C_{\text{worst}}(n)$ pour toute instance de taille n



Pires cas et meilleurs cas (suite)

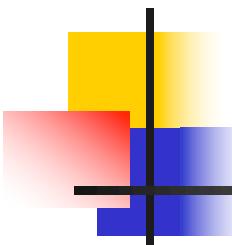
- Dénotons le temps d'exécution sur l'instance x par $C(x)$
- Dénotons la taille de l'instance x par $|x|$
- Le temps d'exécution en pire cas $C_{worst}(n)$ est donné par:

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

- Le temps d'exécution en meilleur cas $C_{best}(n)$ est donné par:

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

- Notez que les opérations min et max se font sur toutes les instances x **dont la taille est n** (et non pas sur toutes les instances peu importe leur taille).



Exemple: recherche séquentielle

ALGORITHME RechSeq($A[0..n-1], K$)

// Retourne le plus petit index j tel

// que $A[j] = K$

$j \leftarrow 0$

while $j < n$ and $A[j] \neq K$ **do**

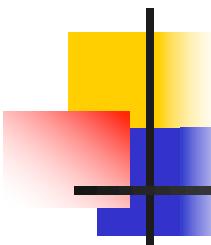
$j \leftarrow j + 1$

if $j < n$ **return** j

else return -1

- L'opération de base de choix est le prédictat $A[j] \neq K$
- Mais notez que le prédictat $j < n$ est exécutée une fois de plus lorsque $A[j] \neq K \forall j$.

- Les pires cas sont obtenus sur les instances où l'on doit tester les n éléments
 - c.-à-d. lorsque $A[j] \neq K \forall j \in \{0..n-2\}$
- Dans ces cas, l'opération de base est exécutée n fois
- Alors $C_{worst}(n) = n$
- Les meilleurs cas sont obtenus sur les instances ayant $A[0] = K$
- Dans ces cas, l'opération de base est exécutée 1 fois
- Alors $C_{best}(n) = 1$
- Donc, pour toute instance de taille n , le temps d'exécution de cet algorithme est compris entre 1 et n .

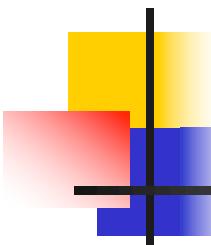


Analyse du cas « moyen »

- On désire obtenir le temps d'exécution pour le cas « typique » : c'est ce que l'on entend normalement par le cas « moyen »
- Mais ce type d'analyse détermine plutôt le **temps d'exécution moyen**: la moyenne des temps d'exécution sur toutes les instances de taille n.
- Soit $C(x)$ le temps d'exécution sur l'instance x .
- Dénotons la taille de l'instance x par $|x|$. Nous nous intéressons à toutes les instances x telles que $|x| = n$.
- Le temps d'exécution moyen $C_{avg}(n)$ des instances de taille n est alors donné par l'espérance des temps d'exécution:

$$C_{avg}(n) = \sum_{x:|x|=n} P(x)C(x)$$

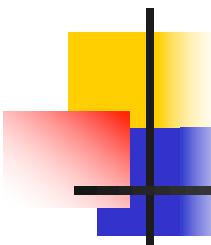
- $P(x)$ = probabilité d'observer l'instance x (parmi les instances de taille n)
- Pour calculer $C_{avg}(n)$, il faut donc fournir une distribution de probabilité « réaliste » sur les instances.



Exemple d'analyse du cas « moyen »

- Illustrons la méthode sur l'algorithme de recherche séquentielle.
- Définissons notre distribution de probabilité comme suit:
 - Supposons que l'on a une probabilité p d'avoir $K \in A[0..n-1]$ (et une probabilité $1-p$ d'avoir $K \notin A[0..n-1]$).
 - Lorsque $K \in A[0..n-1]$, supposons que chaque position j soit équiprobable pour K . Puisque l'on a n éléments, chaque position j possède la même probabilité $1/n$.
- Soit: $C_{avg}^1(n) =$ la moyenne des temps d'exécution quand $K \in A[0..n-1]$
- Soit: $C_{avg}^0(n) =$ la moyenne des temps d'exécution quand $K \notin A[0..n-1]$
- Nous avons alors:

$$C_{avg}(n) = p C_{avg}^1(n) + (1-p) C_{avg}^0(n)$$



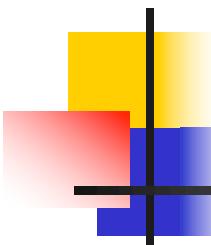
Exemple d'analyse du cas « moyen » (suite)

- Nous avons que $C_{avg}^0(n) = n$ car, dans tous ces cas, l'opération de base est exécutée n fois.
- Lorsque $K \in A[0..n-1]$, chaque position j possède la même probabilité d'occupation $= (1/n)$
- Lorsque $K=A[j]$, l'opération de base est exécutée $1+j$ fois. Alors:

$$\begin{aligned} C_{avg}^1(n) &= \sum_{j=0}^{n-1} \frac{1}{n}(1+j) = 1 + \frac{1}{n} \sum_{j=0}^{n-1} j \\ &= 1 + \frac{1}{n} \cdot \frac{n(n-1)}{2} = \frac{n+1}{2} \end{aligned}$$

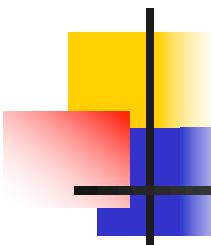
- Alors:

$$\begin{aligned} C_{avg}(n) &= pC_{avg}^1(n) + (1-p)C_{avg}^0(n) \\ &= p\frac{n+1}{2} + (1-p)n \end{aligned}$$



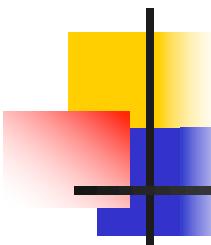
Exemple d'analyse du cas « moyen » (suite)

- Ce résultat caractérise « raisonnablement » le cas « moyen », car:
 - $C_{avg}(n) = (n+1)/2$ lorsque $p=1$
 - En moyenne, K se trouve au milieu du tableau (si $K \in A[0..n-1]$)
 - $C_{avg}(n) = n$ lorsque $p=0$ (lorsque $K \notin A[0..n-1]$)
- Ainsi $C_{avg}(n)$ nous donne une meilleure idée du comportement « typique » sur une instance « quelconque » de l'algorithme que le résultat pour $C_{worst}(n)$ et $C_{best}(n)$
 - C'est habituellement le cas si la distribution utilisée n'est pas trop « irréaliste »
- Nous verrons que certains algorithmes (ex.: le tri rapide) donnent un comportement asymptotique pour $C_{avg}(n)$ qui est nettement avantageux par rapport à celui de $C_{worst}(n)$
- Il est donc en général important de déterminer $C_{avg}(n)$, car $C_{worst}(n)$ est peut-être trop « pessimiste »
- Mais c'est généralement plus difficile d'obtenir $C_{avg}(n)$



Notation asymptotique : motivation

- Nous spécifierons l'efficacité d'un algorithme à l'aide des fonctions $C_{\text{best}}(n)$, $C_{\text{worst}}(n)$ et $C_{\text{avg}}(n)$ (où n = taille de l'instance)
- Ces fonctions caractérisent le temps d'exécution à une constante près
- De plus, l'élément déterminant est le comportement asymptotique de ces fonctions : c.-à-d., leur comportement lorsque $n \rightarrow \infty$
- Nous exprimerons alors ces fonctions à l'aide d'une notation, appelée **notation asymptotique** qui a pour effet de regrouper sous **une même classe toutes les fonctions ayant le même ordre de croissance en n**



Notation asymptotique : motivation (suite)

- Nous pourrions nous objecter à effectuer l'analyse asymptotique de $C(n)$ en prétextant qu'une constante multiplicative peut parfois être plus importante en pratique que son comportement asymptotique
 - Exemple: un algorithme dont $C(n) = n^3$ exécutera plus rapidement qu'un autre algorithme dont $C(n) = 10^6n^2$ lorsque $n < 10^6$
 - Mais de tels cas sont très rares et les constantes multiplicatives ne sont généralement pas « astronomiques ».
- L'élément déterminant pour $C(n)$ est donc son comportement asymptotique
- Puisque que la taille n d'une instance est toujours un entier naturel et que le temps d'exécution est toujours un nombre réel non négatif, nous nous intéressons uniquement aux fonctions $N \rightarrow R^+$
 - Dans ce qui suit, $t(n)$, $f(n)$ et $g(n)$ désignent de telles fonctions

Notation O (grand oh)

- La fonction $t(n)$ appartient à $O(g(n))$, c.-à-d. $t(n) \in O(g(n))$, ssi il existe une constante positive c et un entier positif n_0 tel que $t(n) \leq c \cdot g(n) \forall n \geq n_0$

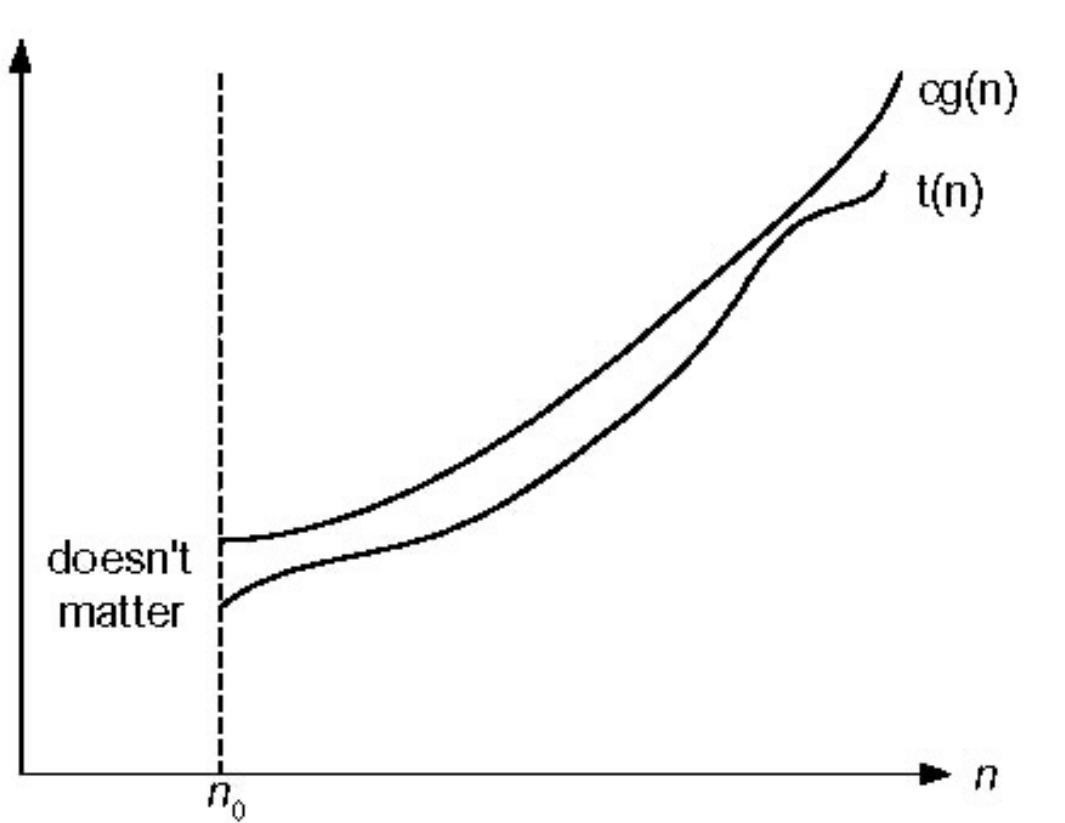
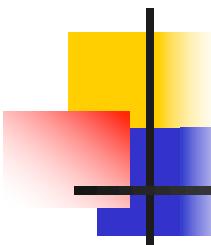


Figure 2.1 Big-oh notation: $t(n) \in O(g(n))$



Notation O (suite)

- Alors $O(g(n))$ est un ensemble de fonctions. C'est l'ensemble suivant:
$$O(g(n)) = \{t(n) : \exists c, n_0 : t(n) \leq cg(n) \quad \forall n \geq n_0\}$$
- Exemples (faciles):
 - $5.4n^2 \in O(n^2)$, car $5.4n^2 \leq 6n^2 \quad \forall n \geq 0$
 - $n(n+1) \in O(n^2)$, car $n(n+1) = n^2 + n \leq 2n^2 \quad \forall n \geq 1$
 - $5n+3 \in O(n)$, car $5n+3 \leq 5n + n = 6n \quad \forall n \geq 3$
 - $5n+3 \in O(n^2)$, car $5n+3 \leq 5n^2 + 3n^2 = 8n^2 \quad \forall n \geq 1$
 - $0.001n^2 \notin O(n)$, car $\nexists c, n_0 : 0.001n^2 \leq cn \quad \forall n \geq n_0$,
 - mais $0.001n^2 \in O(n^2)$, car $0.001n^2 \leq n^2 \quad \forall n \geq 0$
- **Lemme:** Si $t(n) \in O(n)$ alors $t(n) \in O(n^2)$.
 - **Preuve:** $t(n) \in O(n) \Rightarrow \exists c, n_0 : t(n) \leq cn \quad \forall n \geq n_0$. Or $cn \leq cn^2 \quad \forall n \geq 1$. Alors $t(n) \leq cn^2 \quad \forall n \geq n_0$. **CQFD.**
- Puisqu'il existe $t(n) : t(n) \in O(n^2)$ et $t(n) \notin O(n)$. On a $O(n) \subset O(n^2)$.

Notation Ω (grand oméga)

- La fonction $t(n)$ appartient à $\Omega(g(n))$, c.-à-d. $t(n) \in \Omega(g(n))$, ssi il existe une constante positive c et un entier positif n_0 tel que $t(n) \geq c \cdot g(n) \forall n \geq n_0$

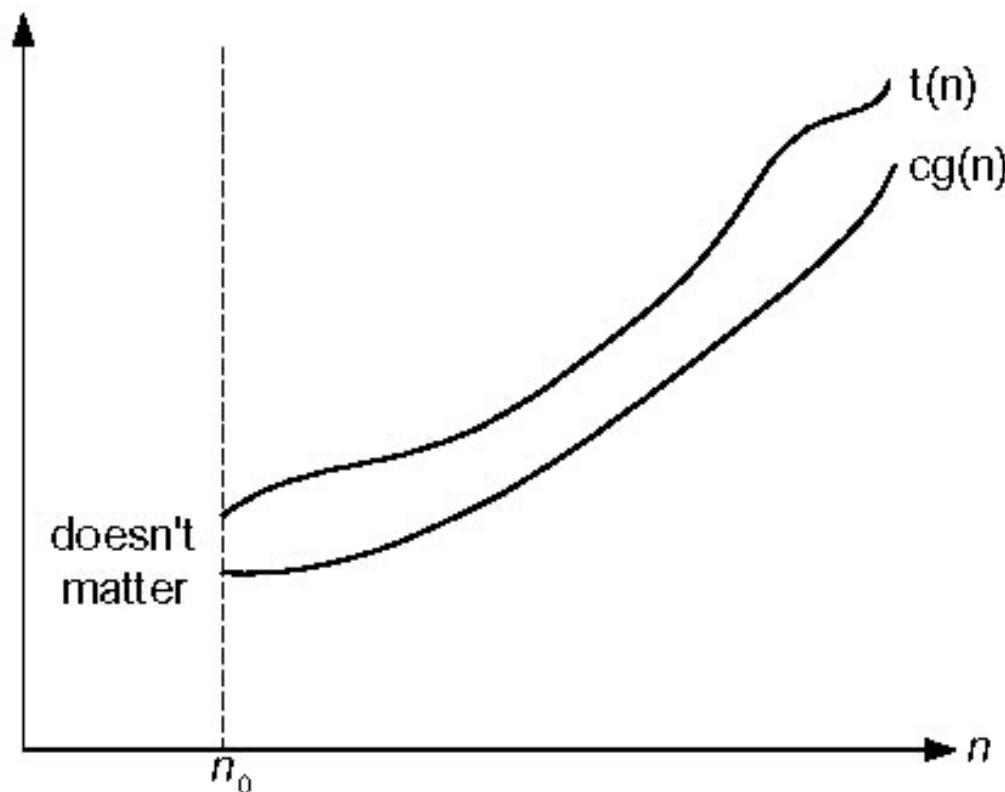
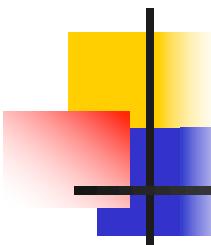
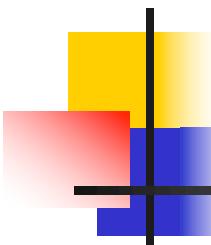


Fig. 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$



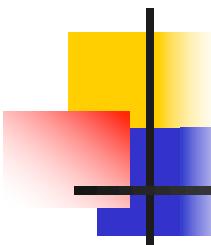
Notation Ω (suite)

- Alors $\Omega(g(n))$ est un ensemble de fonctions. C'est l'ensemble suivant:
$$\Omega(g(n)) = \{t(n) : \exists c, n_0 : t(n) \geq cg(n) \quad \forall n \geq n_0\}$$
- Exemples (faciles):
 - $0.01 n^2 \in \Omega(n^2)$, car $0.01 n^2 \geq 0.001 n^2 \quad \forall n \geq 0$
 - $5n-3 \in \Omega(n)$, car $5n-3 \geq 5n - n = 4n \quad \forall n \geq 3$
 - $n(n-1) \in \Omega(n^2)$, car $n(n-1) = n^2 - n \geq n^2 - 0.5n^2 = 0.5n^2 \quad \forall n \geq 2$
 - $n^2 \in \Omega(n)$, car $n^2 \geq n \quad \forall n \geq 1$
 - $100n \notin \Omega(n^2)$, car $\nexists c, n_0 : 100n \geq cn^2 \quad \forall n \geq n_0$
- **Lemme de dualité entre O et Ω :** $f(n) \in O(g(n))$, $g(n) \in \Omega(f(n))$.
 - **Preuve:**
 - $f(n) \in O(g(n)) \Leftrightarrow \exists c, n_0 : f(n) \leq cg(n) \quad \forall n \geq n_0.$
 - $\Leftrightarrow \exists c, n_0 : g(n) \geq (1/c) f(n) \quad \forall n \geq n_0.$
 - $\Leftrightarrow g(n) \in \Omega(f(n)).$ **CQFD.**



Notation Θ (grand thêta)

- Supposons que $t(n) \in \Omega(f(n))$ et $t(n) \in O(g(n))$. Nous disons alors que:
 - $f(n)$ est une **borne asymptotique inférieure** pour $t(n)$
 - $g(n)$ est une **borne asymptotique supérieure** pour $t(n)$
- Si la même fonction $g(n)$ est à la fois une borne asymptotique inférieure et supérieure pour $t(n)$ nous disons qu'elle est la **borne asymptotique exacte** pour $t(n)$.
 - Nous écrivons alors $t(n) \in \Theta(g(n))$
- Alors: $t(n) \in \Theta(g(n)) \Leftrightarrow t(n) \in \Omega(g(n))$ **et** $t(n) \in O(g(n))$
- Alors: $\Theta(g(n)) = \Omega(g(n)) \cap O(g(n))$
 - Exemple: $n(n-1) \in \Theta(n^2)$. Preuve:
 - $n(n-1) = n^2 - n \geq n^2 - 0.5n^2 = 0.5n^2 \quad \forall n \geq 2 \Rightarrow n(n-1) \in \Omega(n^2)$
 - $n(n-1) \leq n^2 \quad \forall n \geq 0 \Rightarrow n(n-1) \in O(n^2)$.
 - $\Rightarrow n(n-1) \in \Theta(n^2)$. CQFD.



Notation Θ (suite)

- Alors $t(n) \in \Theta(g(n)) \Leftrightarrow$
 $(\exists c_2, n_2 : c_2 g(n) \leq t(n) \forall n \geq n_2) \wedge (\exists c_1, n_1 : t(n) \leq c_1 g(n) \forall n \geq n_1)$
 $\Leftrightarrow \exists c_1, c_2, n_0 : c_2 g(n) \leq t(n) \leq c_1 g(n) \forall n \geq n_0$ (avec $n_0 = \max\{n_1, n_2\}$)
- Certains utilisent cette dernière propriété pour définir $\Theta(g(n))$ qui donne donc cet ensemble suivant de fonctions:

$$\Theta(g(n)) = \{t(n) : \exists c_1, c_2, n_0 : c_2 g(n) \leq t(n) \leq c_1 g(n) \quad \forall n \geq n_0\}$$

Notation Θ (suite)

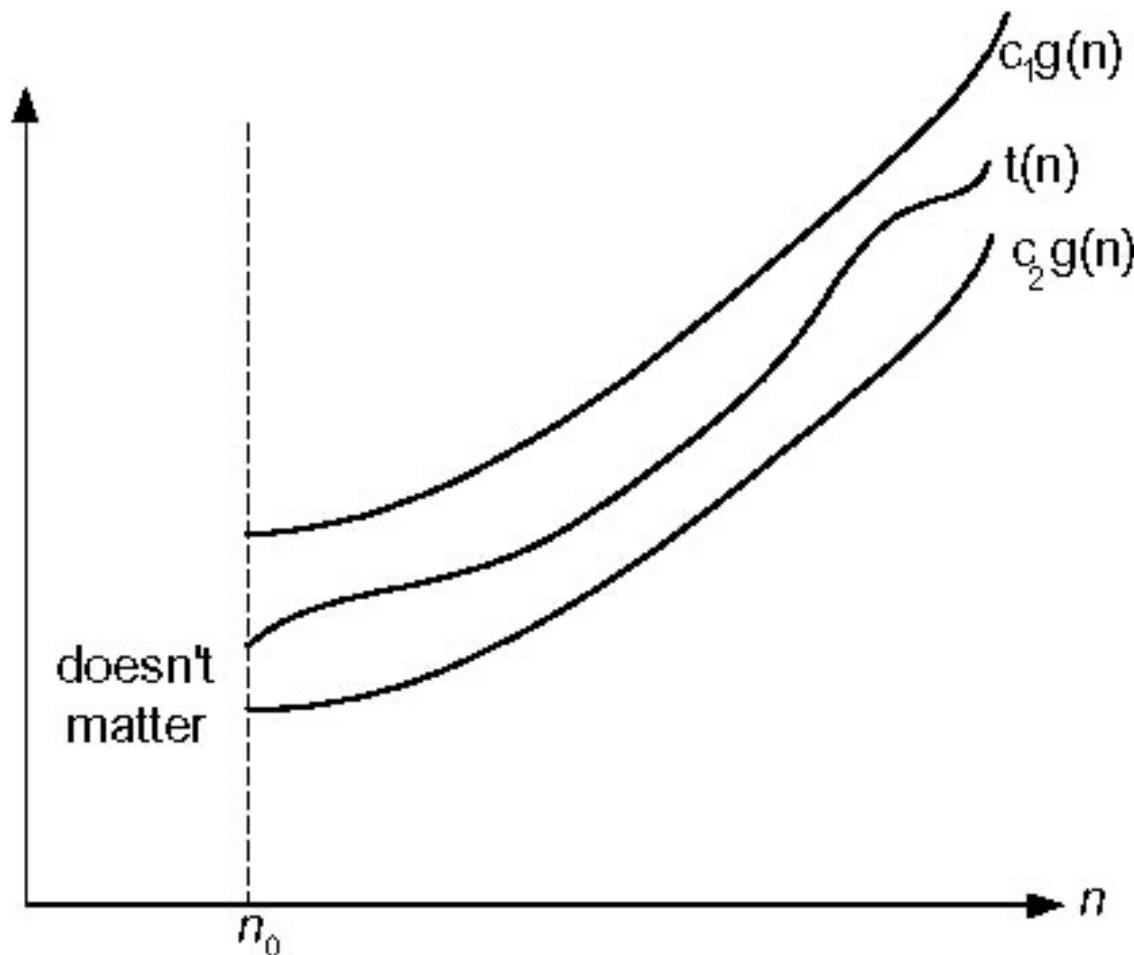
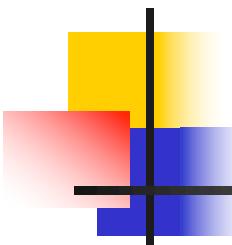
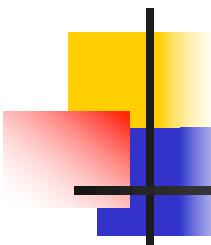


Figure 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$



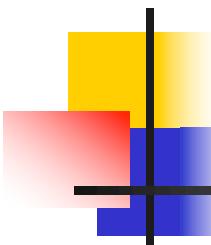
Quelques propriétés

- Les propriétés suivantes sont facilement démontrables:
 - (utilisons Γ pour désigner Ω , O ou Θ)
- **Transitivité:**
 - $f(n) \in \Gamma(g(n)) \wedge g(n) \in \Gamma(h(n)) \Rightarrow f(n) \in \Gamma(h(n))$
- **Réflexivité:**
 - $f(n) \in \Gamma(f(n))$
- **Symétrie:**
 - $f(n) \in \Theta(g(n)), g(n) \in \Theta(f(n))$
- Donc seul Θ définit une relation d'équivalence entre fonctions
- (les preuves de ces propriétés sont laissées en exercice au lecteur)



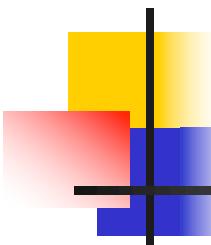
Notation asymptotique et temps d'exécution

- **Rappels:**
 - $T(n)$ = le nombre d'opérations élémentaires effectuées par un algorithme sur une instance de taille n (c.-à-d. le temps d'exécution)
 - $C(n)$ = le nombre d'opérations de base effectuées par un algorithme sur une instance de taille n
 - Nous avions: $C(n) \leq T(n) \leq K \cdot C(n) \quad \forall n \geq 0$ (où K = constante)
- Alors $T(n) \in \Theta(C(n))$. Alors $T(n)$ et $C(n)$ sont des mesures équivalentes du temps d'exécution d'un algorithme
- Puisqu'il est (presque toujours) plus facile de trouver $C(n)$ que de trouver $T(n)$, nous utilisons **C(n)** pour exprimer le temps d'exécution!



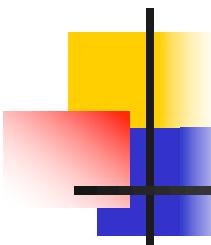
Notation asymptotique et temps d'exécution (suite)

- Pour tout algorithme nous avons:
 - $C_{\text{worst}}(n) \in O(g(n)) \Leftrightarrow C(n) \in O(g(n))$
 - $g(n)$ est donc une borne asymptotique supérieure du temps d'exécution de l'algorithme
 - $C_{\text{best}}(n) \in \Omega(f(n)) \Leftrightarrow C(n) \in \Omega(f(n))$
 - $f(n)$ est donc une borne asymptotique inférieure du temps d'exécution de l'algorithme
 - $C_{\text{worst}}(n) \in \Theta(f(n)) \Rightarrow C(n) \in O(f(n))$
 - $C_{\text{best}}(n) \in \Theta(f(n)) \Rightarrow C(n) \in \Omega(f(n))$
- Est-ce qu'une borne asymptotique exacte $\Theta(f(n))$ sur $C_{\text{worst}}(n)$ nous donne plus d'information qu'une borne asymptotique supérieure $O(f(n))$?
- Est-ce qu'une borne asymptotique exacte $\Theta(f(n))$ sur $C_{\text{best}}(n)$ nous donne plus d'information qu'une borne asymptotique inférieure $\Omega(f(n))$?



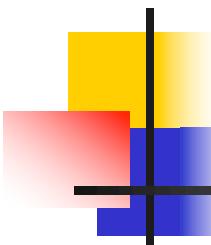
Notation asymptotique et temps d'exécution (suite)

- La borne asymptotique exacte $\Theta(f(n))$ sur $C_{\text{worst}}(n)$ et $C_{\text{best}}(n)$ nous donne plus d'information que les bornes $O(f(n))$ et $\Omega(f(n))$, car:
 - $C_{\text{worst}}(n) \in \Theta(f(n))$ nous assure que $f(n)$ est la borne asymptotique supérieure pour $C(n)$ **qui est la plus faible** qui soit (et donc la meilleure). Nous n'avons pas cette garantie si nous avons seulement $C_{\text{worst}}(n) \in O(f(n))$
 - En effet si (par exemple) $C_{\text{worst}}(n) \in O(n^2)$, il est possible, qu'en fait, nous ayons $C_{\text{worst}}(n) \in \Theta(n)$ (car $n \in O(n^2)$)
 - $C_{\text{best}}(n) \in \Theta(f(n))$ nous assure que $f(n)$ est la borne asymptotique inférieure pour $C(n)$ **qui est la plus élevée** qui soit (et donc la meilleure). Nous n'avons pas cette garantie si nous avons seulement $C_{\text{best}}(n) \in \Omega(f(n))$
 - En effet si (par exemple) $C_{\text{best}}(n) \in \Omega(n)$, il est possible, qu'en fait, nous ayons $C_{\text{best}}(n) \in \Theta(n^2)$ (car $n^2 \in \Omega(n)$)



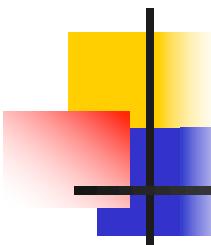
Notation asymptotique et temps d'exécution (suite)

- Que pouvons-nous conclure de $C(n)$ si $C_{\text{worst}}(n) \in \Omega(f(n))$?
 - Réponse: nous ne pouvons pas conclure que $C(n) \in \Omega(f(n))$, car $C_{\text{best}}(n) \in \Omega(g(n))$ avec $f(n) \in \Omega(g(n))$ et la transitivité ne peut pas s'effectuer pour conclure que $C(n) \in \Omega(f(n))$. Nous ne pouvons pas conclure également que $C(n) \in O(f(n))$.
 - Une borne asymptotique inférieure sur $C_{\text{worst}}(n)$ est donc très peu informative, à elle seule, sur $C(n)$
- Que pouvons-nous conclure de $C(n)$ si $C_{\text{best}}(n) \in O(f(n))$?
 - Réponse: nous ne pouvons pas conclure que $C(n) \in O(f(n))$, car $C_{\text{worst}}(n) \in O(g(n))$ avec $f(n) \in O(g(n))$ et la transitivité ne peut pas s'effectuer pour conclure que $C(n) \in O(f(n))$. Nous ne pouvons pas conclure également que $C(n) \in \Omega(f(n))$.
 - Une borne asymptotique supérieure sur $C_{\text{best}}(n)$ est donc très peu informative, à elle seule, sur $C(n)$



La règle du maximum

- Utilisons Γ pour désigner Ω , O ou Θ
- **Théorème (règle du maximum):**
$$t_1(n) \in \Gamma(g_1(n)) \wedge t_2(n) \in \Gamma(g_2(n)) \Rightarrow t_1(n) + t_2(n) \in \Gamma(\max\{g_1(n), g_2(n)\})$$
- **Preuve** (légèrement abrégée):
 - $t_1(n) \in O(g_1(n)) \wedge t_2(n) \in O(g_2(n))$
 - $\Rightarrow (t_1(n) \leq c_1 g_1(n) \quad \forall n \geq n_1) \wedge (t_2(n) \leq c_2 g_2(n) \quad \forall n \geq n_2)$
 - $\Rightarrow t_1(n) + t_2(n) \leq c_1 g_1(n) + c_2 g_2(n) \quad \forall n \geq n_3$ (avec $n_3 = \max\{n_1, n_2\}$)
 $\leq c_3(g_1(n) + g_2(n)) \quad \forall n \geq n_3$ (avec $c_3 = \max\{c_1, c_2\}$)
 $\leq 2c_3 \max\{g_1(n), g_2(n)\} \quad \forall n \geq n_3$
 - $\Rightarrow t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$
 - Similairement on démontre que:
$$t_1(n) \in \Omega(g_1(n)) \wedge t_2(n) \in \Omega(g_2(n)) \Rightarrow t_1(n) + t_2(n) \in \Omega(\max\{g_1(n), g_2(n)\})$$
$$t_1(n) \in \Theta(g_1(n)) \wedge t_2(n) \in \Theta(g_2(n)) \Rightarrow t_1(n) + t_2(n) \in \Theta(\max\{g_1(n), g_2(n)\})$$
 - **CQFD.**



L'utilité de la règle du maximum en algorithmique

- Supposons qu'un algorithme possède deux parties exécutées consécutivement (l'une après l'autre)

- La première partie prend un temps $C_1(n) \in O(g_1(n))$
 - La deuxième partie prend un temps $C_2(n) \in O(g_2(n))$

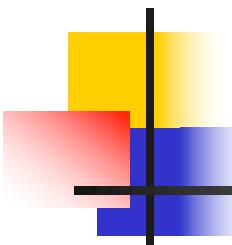
- Le temps total d'exécution $C(n)$ sera alors donné par:

$$C(n) = C_1(n) + C_2(n) \in O(\max\{g_1(n), g_2(n)\})$$

- $C(n)$ est donc déterminé par la partie de l'algorithme ayant l'ordre de croissance le plus élevé

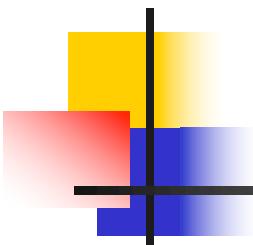
- Exemple:

- supposons que pour déterminer si un tableau A de n éléments possède des éléments identiques, nous trions d'abord A et qu'ensuite, nous parcourions A une seule fois pour déterminer s'il existe $i : A[i] = A[i+1]$.
 - Supposons que notre algorithme de tri prends un temps $C_1(n) \in O(n \log(n))$
 - Puisque la recherche séquentielle prends $C_2(n) \in O(n)$, le temps total sera de $C(n) = C_1(n) + C_2(n) \in O(\max\{n \log(n), n\}) = O(n \log(n))$.
 - Donc $C(n) \in O(n \log(n))$.



Remarques sur la règle du maximum

- Nous pouvons l'utiliser un nombre constant de fois (indépendant de n).
 - Exemple: $O(n^2 + n + \log(n)) = O(\max\{n^2, n, \log(n)\}) = O(n^2)$
- Mais nous ne pouvons pas l'utiliser n fois. (ou $f(n)$ fois)
 - Exemple: $O(n^2) = O(n + n + \dots + n \text{ } (n \text{ fois})) \neq O(\max\{n, n, \dots, n\}) = O(n)$
- Il ne doit pas y avoir de fonctions négatives parmi celles utilisées:
 - Exemple: $O(n) = O(n + n^2 - n^2) \neq O(\max\{n, n^2, -n^2\}) = O(n^2)$

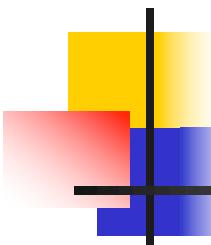


Utilisation des limites pour comparaison d'ordre

- Pour comparer la croissance de deux fonctions $g(n)$ et $f(n)$, il est généralement plus pratique d'utiliser le **théorème** suivant (que d'utiliser les définitions de Ω , O , Θ):

$$\text{si } \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \left\{ \begin{array}{lll} = c \text{ (fini) } > 0 & \text{alors} & f(n) \in \Theta(g(n)) \\ = 0 & \text{alors} & f(n) \in O(g(n)) \text{ et } f(n) \notin \Theta(g(n)) \\ = +\infty & \text{alors} & f(n) \in \Omega(g(n)) \text{ et } f(n) \notin \Theta(g(n)) \end{array} \right.$$

- Lorsque $f(n) \in \Theta(g(n))$ nous disons que $f(n)$ et $g(n)$ ont le **même ordre de croissance**.
- Lorsque $f(n) \in O(g(n))$ et $f(n) \notin \Theta(g(n))$ nous disons que l'ordre de croissance de $f(n)$ est **strictement plus petit** que celui de $g(n)$
- Lorsque $f(n) \in \Omega(g(n))$ et $f(n) \notin \Theta(g(n))$ nous disons que l'ordre de croissance de $f(n)$ est **strictement plus grand** que celui de $g(n)$



Exemples d'utilisation des limites

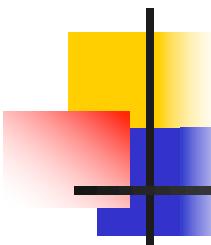
- Exemple: Comparez l'ordre de croissance de $n(n-1)$ avec celui de n^2 .
- Solution:

$$\lim_{n \rightarrow \infty} \frac{n(n-1)}{n^2} = \lim_{n \rightarrow \infty} \left(1 - \frac{1}{n}\right) = 1$$

- alors $n(n-1) \in \Theta(n^2)$. Donc $n(n-1)$ et n^2 ont le même ordre de croissance.
- Exemple: Comparez l'ordre de croissance de $n!$ avec celui de 2^n .
 - Solution:

$$\lim_{n \rightarrow \infty} \frac{n!}{2^n} = \lim_{n \rightarrow \infty} \frac{\sqrt{2\pi n} \left(\frac{n}{e}\right)^n}{2^n} = \lim_{n \rightarrow \infty} \sqrt{2\pi n} \left(\frac{n}{2e}\right)^n = \infty$$

- Donc $n! \in \Omega(2^n)$ et $n! \notin \Theta(2^n)$.
- Donc l'ordre de $n!$ est strictement plus grand que celui de 2^n .



La règle de L'Hôpital

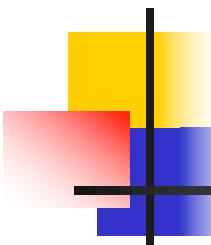
- Il arrive souvent que $f(n)$ et $g(n) \rightarrow \infty$ lorsque $n \rightarrow \infty$. Dans ces cas nous pouvons habituellement utiliser la **règle de L'Hôpital** :

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \lim_{n \rightarrow \infty} \frac{f'(n)}{g'(n)} \quad (\text{si cette limite existe})$$

- où $f'(n)$ désigne la dérivée première de f évaluée à n .
- Cette règle s'applique aux formes $0/0$ et ∞/∞
- Exemple: Comparez l'ordre de croissance de $\lg(n)$ avec celui de $n^{1/2}$.
 - Solution:

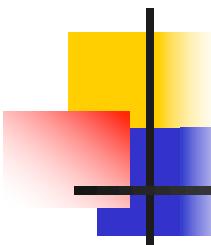
$$\lim_{n \rightarrow \infty} \frac{\log_2(n)}{\sqrt{n}} = \lim_{n \rightarrow \infty} \frac{(\log_2(e) \ln(n))'}{(\sqrt{n})'} = c \cdot \lim_{n \rightarrow \infty} \frac{(1/n)}{(1/2)n^{-1/2}} = 0$$

- Donc $\lg(n) \in O(n^{1/2})$ et $\lg(n) \notin \Theta(n^{1/2})$.
- Donc l'ordre de $\lg(n)$ est strictement plus petit que celui de $n^{1/2}$.



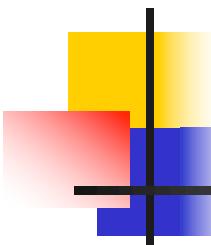
Les classes d'efficacité courantes en algorithmique

- L'ensemble des fonctions appartenant à $\Theta(g(n))$ définie la classe d'efficacité $g(n)$.
- Il existe une infinité de classes d'efficacité distinctes
 - Exemple: $\Theta(2^n) \neq \Theta(3^n) \neq \Theta(4^n) \dots \Theta(n^{1/2}) \neq \Theta(n^{1/3}) \neq \Theta(n^{1/4}) \dots$
- Cependant, en algorithmique, on retrouve surtout les classes d'efficacité suivantes:
 - **Classe 1 (constante)**: sont les algorithmes dont le temps d'exécution demeure constant en fonction de n (quasi impossible)
 - **Classe $\log(n)$ (logarithmique)**: sont les algorithmes qui réduisent n d'un facteur constant à chaque itération (**sans devoir examiner tous les éléments**, sinon l'algorithme serait au moins linéaire)
 - **Classe n (linéaire)**: les algorithmes qui examinent chaque élément un nombre constant de fois
 - **Classe $n \log(n)$** : typique des algorithmes « diviser pour régner » comme le tri fusion



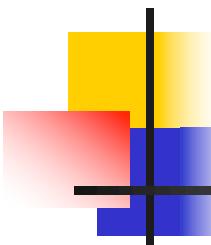
Les classes d'efficacité courantes en algorithmique (suite)

- **Classe n^2 (quadratique):** typiquement les algorithmes constitués de deux boucles imbriquées (ex.: tris élémentaires)
- **Classe n^3 (cubique):** typiquement les algorithmes constitués de trois boucles imbriquées (ex.: algorithmes d'algèbre linéaire)
- **Classe 2^n (exponentielle):** typiquement les algorithmes examinant tous les sous-ensembles des n éléments de l'instance.
 - En fait, la classe exponentielle désigne l'ensemble de toutes les classes k^n
- **Classe $n!$ (factoriel):** typique des algorithmes qui examinent toutes les permutations des n éléments de l'instance



Analyse des algorithmes non récursifs

- Beaucoup d'algorithmes sont non récursifs et ils sont généralement plus simples à analyser que les algorithmes récursifs
 - La difficulté mathématique « majeure » se limite souvent au calcul d'une sommation
- **Plan général d'analyse des algorithmes non récursifs:**
 - Déterminez la taille de l'instance à l'aide d'un (ou des) paramètre(s)
 - Identifiez l'opération de base
 - Déterminez si le nombre de fois que l'opération de base est effectuée dépend uniquement de la taille de l'instance
 - Si ça ne dépend pas uniquement de la taille de l'instance, il faudra faire une analyse séparée des pires cas et des meilleurs cas (et du cas moyen si c'est possible)
 - Effectuez la sommation issue de cette (ou ces) analyse(s)
 - Exprimez uniquement le(s) ordre(s) de croissance à l'aide de la notation asymptotique



Exemple d'un algorithme non récursif

- Considérons le problème de déterminer si les n éléments d'un tableau sont tous distincts
- L'ensemble des instances est l'ensemble de tous les n -tuplets d'éléments
- L'algorithme suivant solutionne correctement ce problème
- La taille de l'instance est le nombre n d'éléments dans le tableau
 - Car nous supposons ici que chaque élément possède une taille bornée par une constante

ALGORITHME ElemDiff($A[0..n-1]$)

// Entrée: tableau A de n éléments
// Sortie: **true** si tous les éléments
// sont distincts et **false** autrement

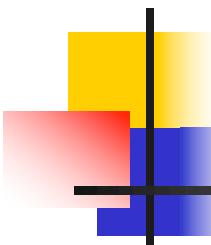
for $i \leftarrow 0$ **to** $n - 2$ **do**

for $j \leftarrow i + 1$ **to** $n - 1$ **do**

if $A[i] = A[j]$ **return false**

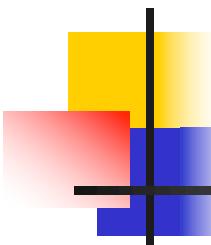
return true

- L'opération de base sera le prédictat de comparaison: $A[i] = A[j]$
 - Cette opération est effectuée au moins aussi souvent que n'importe quelle autre. C'est donc un choix valide



Exemple d'un algorithme non récursif (suite)

- Le nombre de fois que l'opération de base est effectuée dépend de l'existence (ou non) d'éléments identiques et, si c'est le cas, de leur position dans le tableau
- Pour une taille n donnée, le nombre de fois que l'opération de base est effectuée sera différent dans le pire cas et dans le meilleur cas.
- Les meilleurs cas sont ceux où $A[0] = A[1]$, car, pour ces cas, l'algorithme termine après une seule comparaison.
 - Alors $C_{\text{best}}(n) = 1 \in \Theta(1)$
- Les pires cas sont ceux où la comparaison $A[i] = A[j]$ est effectuée un nombre maximal de fois
 - Cela se produit, entre autres, lorsque tous les éléments de A sont distincts
- $C_{\text{worst}}(n)$ sera donc donné par le nombre de fois que cette comparaison est effectuée lorsque tous les n éléments sont distincts



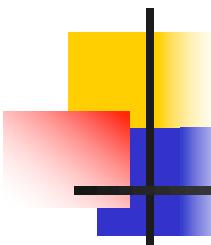
Exemple d'un algorithme non récursif (suite)

- Nous avons alors:

$$\begin{aligned} C_{worst}(n) &= \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i) \\ &= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i = (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2} \\ &= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{n(n-1)}{2} \in \Theta(n^2) \end{aligned}$$

- Nous trouvons donc que $C_{worst}(n) \in \Theta(n^2)$
- Nous nous sommes servis de la « formule » de sommation suivante:

$$\sum_{i=1}^n i = \frac{n(n+1)}{2}$$



Borner une sommation par une intégrale

- Cette approche requiert donc l'utilisation d'un dictionnaire de formules de sommation. Que faire si l'on a besoin d'une formule qui n'est pas dans notre dictionnaire? Par exemple:

$$\sum_{i=1}^n i^4 = ???$$

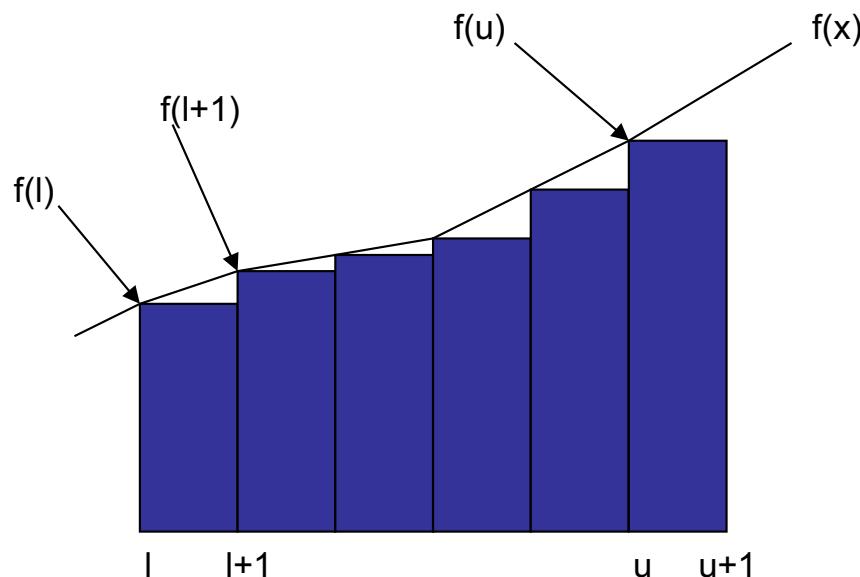
- En fait, puisque l'on ne s'intéresse qu'au comportement asymptotique, nous n'avons pas besoin d'un dictionnaire de formules de sommation.
- Il suffit d'utiliser le **théorème** suivant qui nous donne une borne inférieure et une borne supérieure d'une sommation:

$$\int_{l-1}^u f(x)dx \leq \sum_{i=l}^u f(i) \leq \int_l^{u+1} f(x)dx \text{ pour } f(x) \text{ non décroissant}$$

$$\int_l^{u+1} f(x)dx \leq \sum_{i=l}^u f(i) \leq \int_{l-1}^u f(x)dx \text{ pour } f(x) \text{ non croissant}$$

Preuve pour une fonction non décroissante

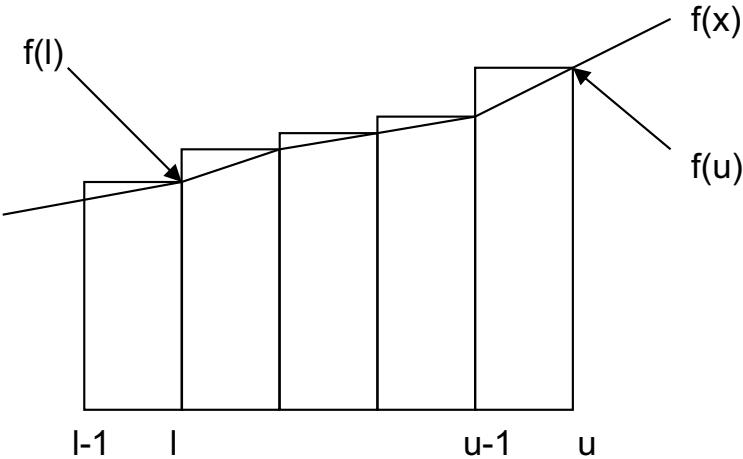
- Rappel: la fonction $f(x)$ est non décroissante dans $[a,b]$ ssi $f(x)$ est croissant ou constant pour tout $x \in [a,b]$.



- La somme des aires de chaque rectangle est inférieure ou égale à l'aire sous $f(x)$ entre l et $u+1$.

$$\sum_{i=l}^u f(i) \leq \int_l^{u+1} f(x) dx$$

Preuve pour une fonction non décroissante (suite)



- La somme des aires de chaque rectangle est supérieure ou égale à l'aire sous $f(x)$ entre $l-1$ et u .

$$\int_{l-1}^u f(x)dx \leq \sum_{i=l}^u f(i)$$

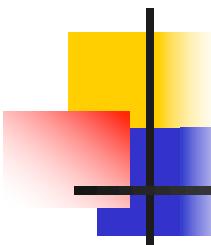
Borner une sommation par une intégrale (suite)

- Par exemple, puisque x^4 est une fonction non décroissante, nous avons:

$$\begin{aligned} \int_0^n x^4 dx &\leq \sum_{i=1}^n i^4 \leq \int_1^{n+1} x^4 dx \\ \Rightarrow \left[\frac{x^5}{5} \right]_0^n &\leq \sum_{i=1}^n i^4 \leq \left[\frac{x^5}{5} \right]_1^{n+1} \\ \Rightarrow \frac{n^5}{5} &\leq \sum_{i=1}^n i^4 \leq \frac{(n+1)^5}{5} - \frac{1}{5} \\ \Rightarrow \frac{n^5}{5} &\leq \sum_{i=1}^n i^4 \leq \frac{(n+n)^5}{5} \\ \Rightarrow \frac{n^5}{5} &\leq \sum_{i=1}^n i^4 \leq \frac{2^5}{5} n^5 \\ \Rightarrow \sum_{i=1}^n i^4 &\in \Theta(n^5) \end{aligned}$$

- De la même manière, nous trouvons que:

$$\sum_{i=1}^n i^k \in \Theta(n^{k+1})$$



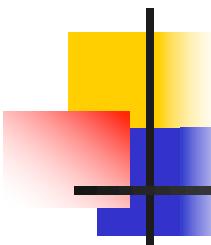
De retour à l'analyse de ElemDiff en pire cas

- Retournons à l'analyse en pire cas de l'algorithme ElemDiff où nous avions:

$$\begin{aligned}C_{worst}(n) &= \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i) \\&= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i = (n-1)^2 - \sum_{i=0}^{n-2} i\end{aligned}$$

- Puisque x est une fonction non décroissante, bornons la dernière sommation par les deux intégrales:

$$\begin{aligned}\int_{-1}^{n-2} x dx &\leq \sum_{i=0}^{n-2} i \leq \int_0^{n-1} x dx \\ \Rightarrow \left[\frac{x^2}{2} \right]_{-1}^{n-2} &\leq \sum_{i=0}^{n-2} i \leq \left[\frac{x^2}{2} \right]_0^{n-1}\end{aligned}$$

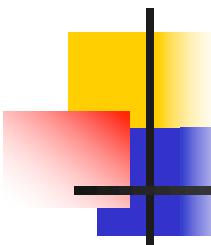


De retour à l'analyse de ElemDiff en pire cas (suite)

- Nous avons alors:

$$\begin{aligned} - \left[\frac{x^2}{2} \right]_0^{n-1} &\leq - \sum_{i=0}^{n-2} i \leq - \left[\frac{x^2}{2} \right]_{-1}^{n-2} \Rightarrow \\ (n-1)^2 - \left[\frac{x^2}{2} \right]_0^{n-1} &\leq (n-1)^2 - \sum_{i=0}^{n-2} i \leq (n-1)^2 - \left[\frac{x^2}{2} \right]_{-1}^{n-2} \\ \Rightarrow \frac{(n-1)^2}{2} &\leq C_{worst}(n) \leq (n-1)^2 - \frac{(n-2)^2}{2} + \frac{1}{2} \\ \Rightarrow \frac{(n-1)^2}{2} &\leq C_{worst}(n) \leq \frac{n^2 - 1}{2} \end{aligned}$$

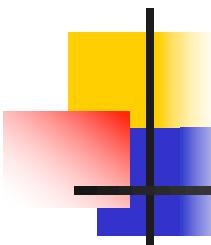
- Nous trouvons donc que $C_{worst}(n) \in \Theta(n^2)$ sans avoir utilisé un dictionnaire de formules de sommation.



Appels de fonction

- Souvent, un algorithme fait un appel à une fonction qui ne s'exécute pas en temps constant.
- C'est le cas de l'algorithme **Compte** qui retourne le nombre d'éléments dans le vecteur B qui se retrouvent dans le vecteur A.
- La taille de l'instance est donnée à la fois par les valeurs de m et de n .
- En meilleur cas, tous les éléments dans B sont identiques à A[0]
- En pire cas, aucun des éléments dans B n'est dans A.

```
ALGORITHME Compte(A[0..n-1],  
                      B[0..m-1])  
  
// Entrée: 2 tableaux de n et m éléments  
// Sortie: Le nombre d'éléments dans B  
// qui se retrouvent dans A  
  
count ← 0  
  
for i ← 0 to m - 1 do  
  
    if RechSeq(A, B[i]) ≥ 0 then  
        count ← count + 1  
  
return count
```

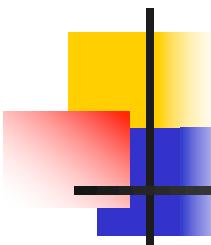


Appels de fonction

- Puisque RechSeq n'est pas une instruction élémentaire s'exécutant en temps $\Theta(1)$, nous ne pouvons pas inscrire la valeur 1 dans la sommation.
- Il faut plutôt y inscrire le temps d'exécution de la fonction qui est appelée.

$$C_{\text{worst}}^{\text{Compte}}(n, m) = \sum_{i=0}^{m-1} C_{\text{worst}}^{\text{RechSeq}}(n) = \sum_{i=0}^{m-1} n = nm \in \Theta(nm)$$

$$C_{\text{best}}^{\text{Compte}}(n, m) = \sum_{i=0}^{m-1} C_{\text{best}}^{\text{RechSeq}}(n) = \sum_{i=0}^{m-1} 1 = m \in \Theta(m)$$



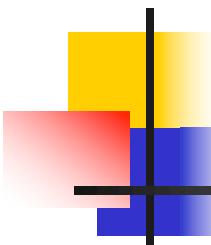
Appels de fonction

- Une autre façon de procéder est de remplacer l'appel de fonction par le code de la fonction elle-même.
- On prend $A[j] \neq B[i]$ comme opération de base.

$$C_{\text{BEST}}(n, m) = \sum_{i=0}^{m-1} 1 \\ = m \in \Theta(m)$$

$$C_{\text{WORST}}(n, m) = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} 1 \\ = \sum_{i=0}^{m-1} n = nm \in \Theta(nm)$$

ALGORITHME Compte($A[0..n-1]$,
 $B[0..m-1]$)
// Entrée: 2 tableaux de n et m éléments
// Sortie: Le nombre d'éléments dans B
qui se retrouvent dans A
count $\leftarrow 0$
for $i \leftarrow 0$ **to** $m - 1$ **do**
 $j \leftarrow 0$
 while $j < n$ **and** $A[j] \neq B[i]$ **do**
 $j \leftarrow j + 1$
 if $j < n$ **then**
 count \leftarrow count + 1
return count



Autre exemple d'un algorithme non récursif

- Il arrive parfois que l'analyse du temps d'exécution d'un algorithme non récursif ne nous donne pas une sommation à effectuer.
- Considérez l'algorithme $\text{Binary}(n)$ pour calculer le nombre de bits de l'entier de valeur n .
- Utilisons la division par deux dans $\lfloor n/2 \rfloor$ pour l'opération de base.
- Le nombre $C(n)$ de fois que l'opération de base est effectuée pour un entier de valeur n est donnée par la récurrence:

$$C(n) = C(\lfloor n/2 \rfloor) + 1 \text{ si } n > 1$$

$$C(n) = 0 \quad \quad \quad \text{si } n \leq 1$$

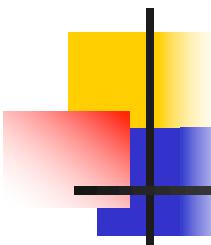
ALGORITHME $\text{Binary}(n)$

```
// Entrée: entier positif de valeur n  
// Sortie: le nombre de bits utilisés pour  
// sa représentation binaire  
  
count ← 1  
  
while n > 1 do  
    count ← count + 1  
    n ←  $\lfloor n/2 \rfloor$   
  
return count
```

- De telles relations de récurrence émergent naturellement des algorithmes récursifs.

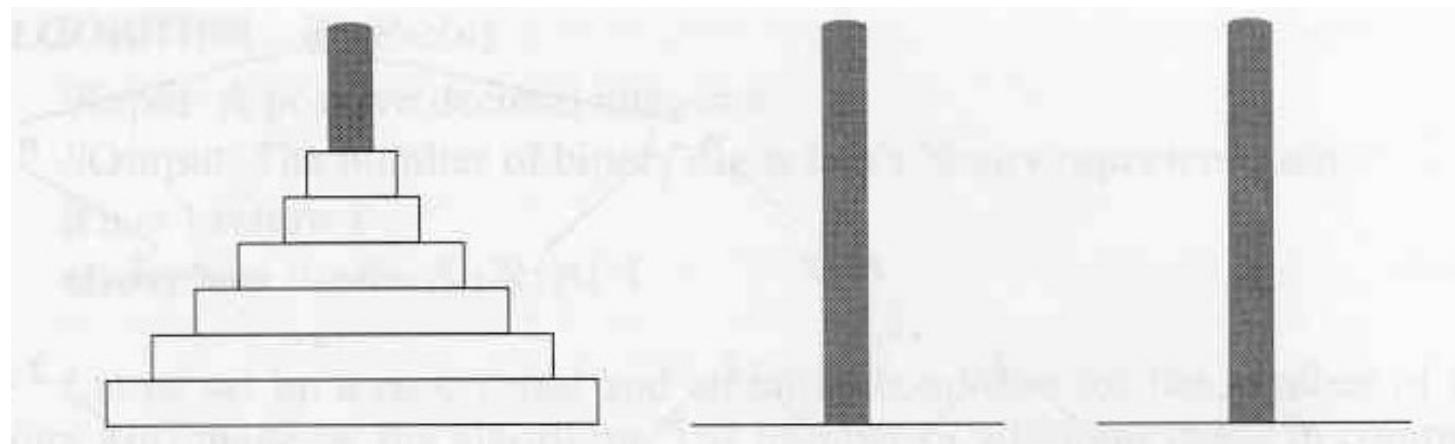
- **Plan général d'analyse des algorithmes récursifs:**

- Déterminez la taille de l'instance à l'aide d'un (ou des) paramètre(s)
- Identifiez l'opération de base
- Déterminez si le nombre de fois que l'opération de base est effectuée dépend uniquement de la taille de l'instance
 - Si ça ne dépend pas uniquement de la taille de l'instance, il faudra faire une analyse séparée des pires cas et des meilleur cas (et du cas moyen si c'est possible)
- Solutionnez la relation de récurrence issue de cette (ou ces) analyse(s)
- Exprimez uniquement le(s) ordre(s) de croissance à l'aide de la notation asymptotique



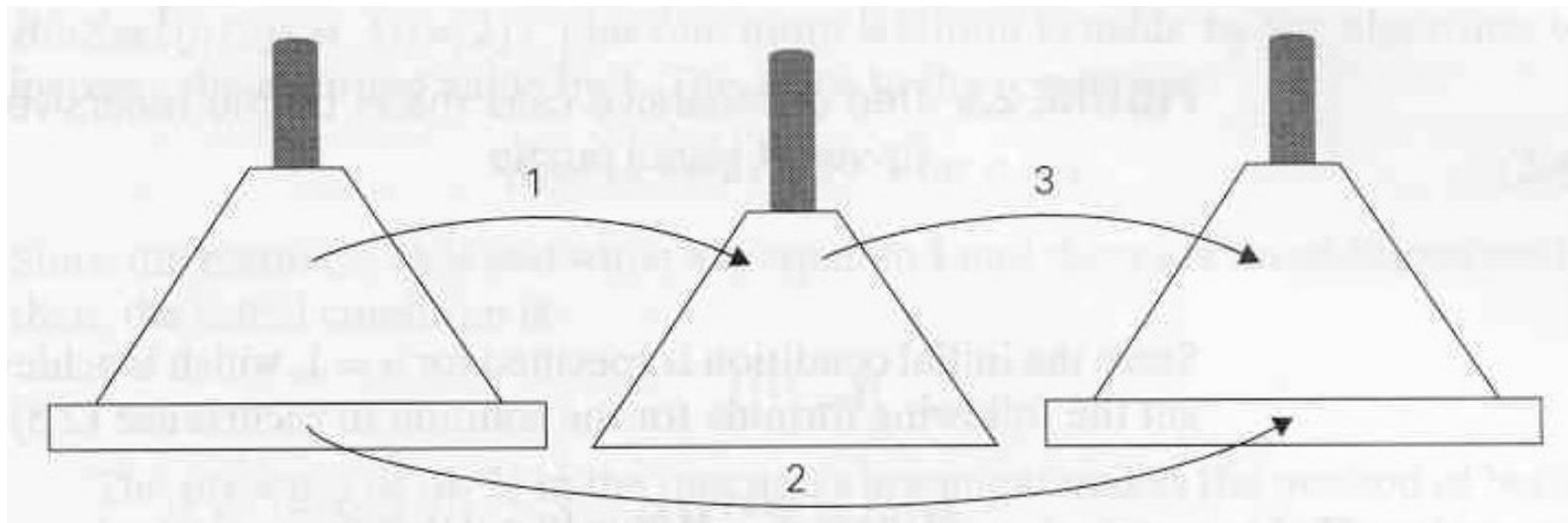
Exemple: les tours de Hanoi

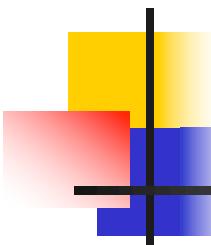
- Nous avons 3 tiges (tours) et n disques de tailles (rayons) distinctes
- Initialement: tous les disques sont sur la 1re tige en ordre de taille
 - Le plus petit disque est au sommet
- **Objectif:** déplacer tous les disques (un à la fois) de la 1re tige à la troisième tige en utilisant (possiblement) la seconde tige comme tige auxiliaire
 - **Contrainte:** à tout moment, il ne peut pas y avoir un disque au-dessus d'un autre disque de plus petite taille
- **Solution pour $n=1$:** nous pouvons déplacer l'unique disque directement de la 1re à la 3e tige



Exemple: les tours de Hanoi (suite)

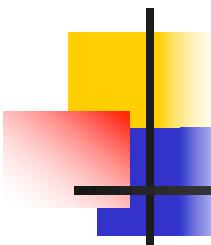
- La solution pour $n > 1$ est donnée par l'algorithme récursif suivant:
 - (1) Déplacer d'abord les $n-1$ disques supérieurs de la tige 1 à la tige 2 en se servant de la tige 3 comme tige auxiliaire
 - (2) Déplacer le plus grand disque de la tige 1 à 3 (directement)
 - (3) Déplacer les $n-1$ disques de la tige 2 à la tige 3 en se servant de la tige 1 comme tige auxiliaire
- Cet algorithme est correct pour $n=1$. S'il est correct pour n , il le sera pour $n+1$ (pour tout $n \geq 1$). L'algorithme est donc correct pour tout $n \geq 1$.





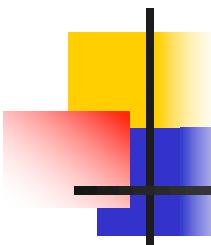
Analyse de l'algorithme des tours de Hanoi

- La taille de l'instance = n (le nombre de disques)
- L'opération de base = déplacement d'un disque d'une tige à une autre
- Il existe une seule instance lorsque n est fixe. Le temps d'exécution $C(n)$ dépend donc uniquement de n .
 - Le pire cas est identique au meilleur cas
- Déterminons alors $C(n)$: le nombre de déplacements effectués par l'algorithme pour solutionner le problème avec n disques
- L'algorithme nous indique que:
 - $C(n) = C(n-1) + 1 + C(n-1) = 2C(n-1) + 1 \quad \forall n > 1$ (**la récurrence**)
 - Avec: $C(1) = 1$ (**condition initiale**)
- Tout changement apporté à la condition initiale (ex.: $C(1) = 2$) entraîne une modification de la solution générée par la récurrence.
 - La solution dépend de la condition initiale: il faut donc la spécifier



Analyse de l'algorithme des tours de Hanoi (suite)

- Solutionnons cette récurrence par la **méthode des substitutions à rebours** (« backward substitutions »)
 - L'idée est d'exprimer la solution en fonction de la condition initiale
- $C(n) = 2C(n-1) + 1$
$$\begin{aligned} &= 2[2C(n-2) + 1] + 1 && (\text{car } C(n-1) = 2C(n-2) + 1) \\ &= 2^2C(n-2) + 2 + 1 \\ &= 2^2[2C(n-3) + 1] + 2 + 1 && (\text{car } C(n-2) = 2C(n-3) + 1) \\ &= 2^3C(n-3) + 2^2 + 2 + 1 \\ &= \dots \\ &= 2^iC(n-i) + 2^{i-1} + 2^{i-2} + \dots + 2 + 1 \quad \forall i > 1 \\ &= 2^{n-1} C(1) + 2^{n-2} + \dots + 2 + 1 \quad (\text{en utilisant } i = n-1) \\ &= 2^{n-1} + 2^{n-2} + \dots + 2 + 1 \quad (\text{en utilisant } C(1) = 1) \\ &= 2^n - 1 \quad (\text{série géométrique très connue; voir annexe A du manuel}) \end{aligned}$$
- **Alors $C(n) = 2^n - 1 \in \Theta(2^n)$ (ordre de croissance exponentiel)**
- La concision d'un algorithme récursif n'est pas garantie de son efficacité
- Mais, dans ce cas-ci, il semble qu'il soit impossible de faire mieux!



Autre exemple d'algorithme récursif

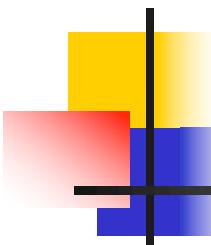
- Considérons BinRec(n) qui est la version récursive de Binary(n)
- Opération de base: « +1 »
 - Autre possibilité: la division dans $\lfloor n/2 \rfloor$
- Le nombre $C(n)$ d'opérations de base effectuées par l'algorithme sur un entier de valeur n est donné par la récurrence:
$$C(n) = C(\lfloor n/2 \rfloor) + 1$$
- Avec la condition initiale:
$$C(1) = 0$$
- Essayons d'abord de résoudre seulement pour les valeurs de n satisfaisant:
$$n = 2^k \text{ pour } k \in \mathbb{N}$$

ALGORITHME BinRec(n)

```
// Entrée: entier positif de valeur n  
// Sortie: le nombre de bits utilisés pour  
// sa représentation binaire
```

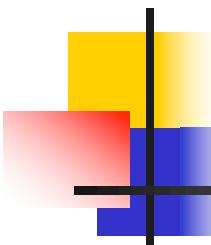
```
If n = 1 return 1  
else return BinRec( $\lfloor n/2 \rfloor$ ) + 1
```

- Nous avons alors:
$$C(2^k) = C(2^{k-1}) + 1$$
- Avec:
$$C(2^0) = 0$$



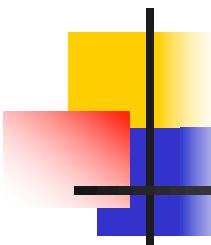
Solution de l'algorithme récursif

- La méthode des substitutions à rebours donne:
 - $$\begin{aligned} C(2^k) &= C(2^{k-1}) + 1 \\ &= [C(2^{k-2}) + 1] + 1 \quad (\text{car } C(2^{k-1}) = C(2^{k-2}) + 1) \\ &= C(2^{k-2}) + 2 \\ &= [C(2^{k-3}) + 1] + 2 \quad (\text{car } C(2^{k-2}) = C(2^{k-3}) + 1) \\ &= C(2^{k-3}) + 3 \\ &\dots \\ &= C(2^{k-i}) + i \quad (\text{pour tout } i > 0) \\ &= C(2^0) + k \quad (\textbf{pour } i = k) \\ &= k \quad (\text{car } C(2^0) = 0) \end{aligned}$$
- Alors $C(n) = \lg(n) \in \Theta(\log(n))$ pour $n = 2^k$.
- Or (voir pages suivantes) $\lg(n)$ est une **fonction harmonieuse**.
- On a alors (voir pages suivantes) $C(n) \in \Theta(\log(n)) \forall n > 1$
- $C(n)$ est donc linéaire en la taille de n ($\forall n > 1$). (Nous savions cela...)



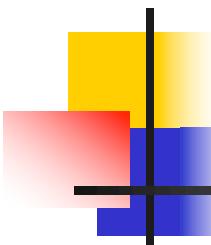
Fonctions harmonieuses

- Une fonction non négative $f(n)$ est **éventuellement non décroissante** ssi il existe n_0 tel que $f(n)$ est non décroissante dans $[n_0, \infty)$
 - Exemples de fonctions éventuellement non décroissantes
 - $\log(n)$ (avec $n_0 = 1$)
 - $n^k \quad \forall k > 0$ (avec $n_0 = 0$)
 - $n \log(n)$ (avec $n_0 = 0$)
 - 2^n (avec $n_0 = 0$)
- Une fonction éventuellement non décroissante est **harmonieuse** ssi
$$f(2n) \in \Theta(f(n))$$
- Exemples:
 - $\log(n)$ est harmonieuse, car $\log(2n) = \log(2) + \log(n) \in \Theta(\log(n))$
 - n^k est harmonieuse, car $(2n)^k = 2^k n^k \in \Theta(n^k)$
 - 2^n n'est pas harmonieuse car $2^{2n} = (2^2)^n = 4^n \notin \Theta(2^n)$
 - $n!$ n'est pas harmonieuse, car $(2n)! \notin \Theta(n!)$



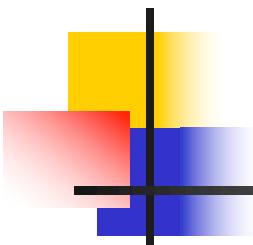
La règle de l'harmonie

- **Théorème:** Si $f(n)$ est harmonieuse alors $f(bn) \in \Theta(f(n)) \quad \forall b \geq 2$.
 - Preuve: voir annexe B du manuel (facultative)
- **Théorème (règle de l'harmonie):** Si $C(n)$ est éventuellement non décroissante, si $f(n)$ est harmonieuse et si $C(n) \in \Theta(f(n))$ pour $n = b^k$ avec $k \in \mathbb{N}$ alors $C(n) \in \Theta(f(n)) \quad \forall n \in \mathbb{N}$
 - Preuve: voir annexe B du manuel (facultative)
 - Ce théorème est également valide pour Θ et Ω
- **Application:** pour un algorithme donnant un $C(n)$ éventuellement non décroissant, chaque fois que trouverons que $C(n) \in \Gamma(f(n))$ pour $n = b^k$, nous pourrons en conclure que $C(n) \in \Gamma(f(n)) \quad \forall n$ lorsque $f(n)$ est harmonieuse. (ici Γ désigne Θ , Ω ou Θ)
 - C'est ce que nous avons fait pour l'analyse de BinRec(n)



La suite de Fibonacci

- La **suite de Fibonacci** est constituée des nombres de Fibonacci:
 - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...
- Elle fut introduite par Leonardo Fibonacci en 1202 et se trouve présente dans la solution de nombreux problèmes
- La suite de Fibonacci est générée par la récurrence:
$$F(n) = F(n-1) + F(n-2)$$
- Avec les conditions initiales:
$$F(0) = 0, \quad F(1) = 1$$
- Solutionnons cette récurrence.
 - Pour cela nous devons trouver le terme général $F(n)$
 - La méthode de substitutions à rebours est impuissante ici
- C'est un cas particulier d'une relation de **récurrence linéaire homogène du second ordre**.
 - Pour résoudre ce type de récurrence, nous utiliserons la **méthode de l'équation caractéristique**
 - (La méthode de substitutions à rebours fonction bien pour les récurrences du premier ordre)



Réurrences linéaires du second ordre

- Une **récurrence linéaire du second ordre** est une récurrence de la forme:

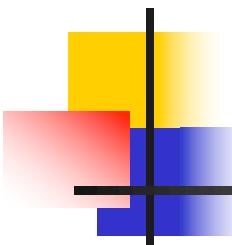
$$a x(n) + b x(n-1) + c x(n-2) = f(n)$$

- où a , b et c sont des constantes réelles, indépendantes de n , et $a \neq 0$.
- La récurrence est dite **homogène** lorsque $f(n) = 0$. Sinon elle est **inhomogène**.
- La récurrence de Fibonacci est une récurrence homogène de cette forme avec $a = 1$, $b = c = -1$, car elle s'écrit:

$$F(n) - F(n-1) - F(n-2) = 0$$

- Analysons alors le cas homogène:

$$a x(n) + b x(n-1) + c x(n-2) = 0$$



Méthode de l'équation caractéristique

- **Théorème:** Considérez **l'équation caractéristique:**

$$a r^2 + b r + c = 0$$

- avec les mêmes coefficients a, b, c , que ceux de **relation de récurrence**:

$$a x(n) + b x(n-1) + c x(n-2) = 0$$

- Désignons les deux racines de l'équation caractéristique par r_1 et r_2

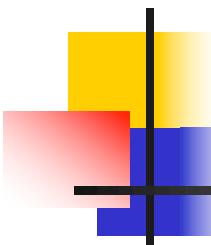
- **Si $r_1 \neq r_2$:** La solution générale de la récurrence est donnée par:

$$x(n) = \alpha r_1^n + \beta r_2^n$$

- **Si $r_1 = r_2$:** La solution générale est donnée par:

$$x(n) = \alpha r_1^n + \beta n r_1^n$$

- Dans tous les cas, α et β sont déterminés par les conditions initiales de la récurrence, c.-à-d. par $x(0)$ et $x(1)$



Méthode de l'équation caractéristique (suite)

- **Preuve du théorème:** Si nous substituons $x(n) = r^n$ dans la relation de récurrence nous obtenons:

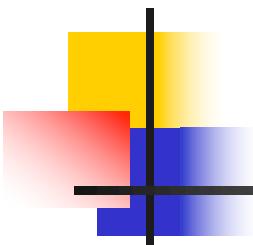
$$a r^n + b r^{n-1} + c r^{n-2} = 0 , \quad (a r^2 + b r + c)r^{n-2} = 0$$

, $a r^2 + b r + c = 0$ ou $r = 0$ (solution triviale sans intérêt)

- Donc $x(n) = r^n$ est une solution de la récurrence lorsque r est une racine de l'équation caractéristique
- Lorsque l'équation caractéristique possède deux racines distinctes r_1 et r_2 , toute combinaison linéaire $x(n) = \alpha r_1^n + \beta r_2^n$ sera également solution de la récurrence linéaire
 - Ceci est la solution la plus générale, car les deux paramètres α et β peuvent satisfaire toute condition initiale $x(0), x(1)$:

$$x(0) = \alpha + \beta$$

$$x(1) = \alpha r_1 + \beta r_2$$



Méthode de l'équation caractéristique (suite)

- **...suite de la preuve...** Mais lorsque l'équation caractéristique admet une seule racine distincte $r_1 = r_2$, la combinaison linéaire devient $x(n) = (\alpha + \beta)n r_1^n = \gamma n r_1^n$ et nous ne pouvons plus satisfaire toutes les conditions initiales possibles
- Nous allons démontrer que, dans ce cas, $x(n) = n r_1^n$ est une autre solution de la récurrence. Ainsi la combinaison linéaire $x(n) = \alpha r_1^n + \beta n r_1^n$ nous donnera la solution générale permettant de satisfaire toute condition initiale

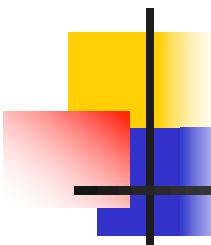
$$x(0) = \alpha$$

$$x(1) = \alpha r_1 + \beta r_1$$

- Pour démontrer que $x(n) = n r_1^n$ est une solution de la récurrence, exploitons le fait que lorsque l'équation caractéristique possède une seule racine r_1 , on a:

$$a r^2 + b r + c = s(r - r_1)^2$$

- Pour une certaine valeur réelle $s \neq 0$

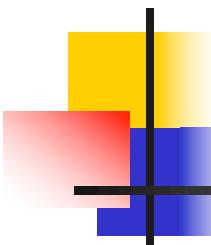


Méthode de l'équation caractéristique (suite)

- ...suite de la preuve...
- Ainsi, lorsque $x(n) = n r^n$, $a x(n) + b x(n-1) + c x(n-2)$ devient égal à:

$$\begin{aligned} &= anr^n + b(n-1)r^{n-1} + c(n-2)r^{n-2} \\ &= r \frac{d}{dr} [ar^n + br^{n-1} + cr^{n-2}] \\ &= r \frac{d}{dr} [(ar^2 + br + c)r^{n-2}] = r \frac{d}{dr} [s(r - r_1)^2 r^{n-2}] \\ &= r [2s(r - r_1)r^{n-2} + s(r - r_1)^2(n - 2)r^{n-3}] \\ &= 0 \quad \text{lorsque } r = r_1 \end{aligned}$$

- Donc $a x(n) + b x(n-1) + c x(n-2) = 0$ lorsque $x(n) = n r_1^n$. **CQFD.**
- La méthode de l'équation caractéristique se généralise aux récurrences linéaires d'ordre k , mais, dans ce cas, il faut trouver les racines d'un polynôme d'ordre k ...



De retour à Fibonacci

- Solutionnons la récurrence de Fibonacci:

$$F(n) - F(n-1) - F(n-2) = 0$$

Avec: $F(0) = 0, F(1) = 1$

- L'équation caractéristique associée à cette récurrence est:

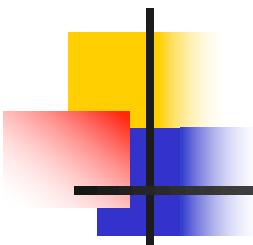
$$r^2 - r - 1 = 0$$

- Les racines de cette équation sont données par:

$$r_{1,2} = \frac{1 \pm \sqrt{1 - 4(-1)}}{2} = \frac{1 \pm \sqrt{5}}{2}$$

- Puisque nous avons deux racines distinctes réelles, la solution générale de la récurrence de Fibonacci est donnée par:

$$F(n) = \alpha \left(\frac{1 + \sqrt{5}}{2} \right)^n + \beta \left(\frac{1 - \sqrt{5}}{2} \right)^n$$



De retour à Fibonacci (suite)

- Les valeurs de α et β sont déterminées par les conditions initiales:

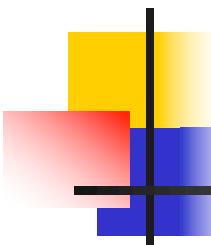
$$F(0) = \alpha + \beta = 0$$

$$F(1) = \alpha \frac{1 + \sqrt{5}}{2} + \beta \frac{1 - \sqrt{5}}{2} = 1$$

- La solution de ce système est obtenue en substituant $\beta = -\alpha$ dans la seconde équation et en isolant α . Nous trouvons alors $\alpha = 5^{-1/2}$ et $\beta = -5^{-1/2}$. Alors:

$$\begin{aligned} F(n) &= \frac{1}{\sqrt{5}} \left(\frac{1 + \sqrt{5}}{2} \right)^n - \frac{1}{\sqrt{5}} \left(\frac{1 - \sqrt{5}}{2} \right)^n \\ &= \frac{1}{\sqrt{5}} (\phi^n - \psi^n) \approx \frac{1}{\sqrt{5}} [(1.61803)^n - (-0.61803)^n] \end{aligned}$$

- $F(n)$ augmente donc exponentiellement rapidement avec n car $|\psi| < 1$ et alors $\psi^n \rightarrow 0$ lorsque $n \rightarrow \infty$.



Algorithme récursif pour générer les nombres de Fibonacci

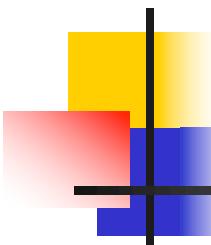
ALGORITHME FibRec(n)

```
// Entrée: un entier  $n \geq 0$ 
// Sortie: le  $n$ ème nombre de Fibonacci

If  $n \leq 1$  return  $n$ 
else return FibRec( $n-1$ )+FibRec( $n-2$ )
```

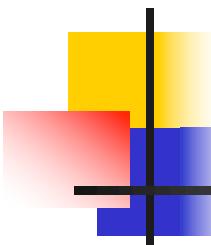
- Quel est le temps d'exécution de cet algorithme?
- L'opération de base est « + »
- Le nombre $C(n)$ de fois que cette opération de base est effectuée ne dépend que de n :
 - $C(n) = C_{\text{best}}(n) = C_{\text{worst}}(n)$
- Notez que la taille de l'instance est $\lceil \lg(n+1) \rceil$

- La valeur de $F(n)$ augmente exponentiellement avec n
- Alors ce n'est pas vraiment réaliste de compter chaque addition comme valant une opération élémentaire
- Mais c'est ce que nous ferons ici (par simplicité)
 - Sinon, un choix valide pour l'opération de base serait l'addition de deux bits
 - Le nombre d'opérations de base pour effectuer $F(n-1) + F(n-2)$ serait alors donné par $\lceil \lg(F(n-1)+1) \rceil \in \Theta(n)$
- Pour simplifier, supposons ici que $F(n-1) + F(n-2)$ nécessite seulement une opération de base



Algorithme récursif pour générer les nombres de Fibonacci (suite)

- Avec cette hypothèse simplificatrice, la valeur de $C(n)$ est donnée par:
 - $C(n) = C(n-1) + C(n-2) + 1$
- Nous obtenons alors la relation de récurrence inhomogène:
 - $C(n) - C(n-1) - C(n-2) = 1$
 - Avec: $C(0) = 0, C(1) = 0$
- **Théorème:** La solution générale d'une récurrence linéaire inhomogène est donnée par la somme de la solution générale à la récurrence homogène correspondante et d'une solution particulière à la récurrence linéaire inhomogène.
 - Il est en général difficile de trouver une solution particulière de la récurrence inhomogène.
 - Mais c'est facile lorsque le terme inhomogène est une constante, car, dans ce cas, une solution particulière constante existe!



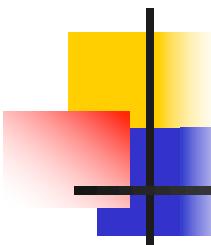
Algorithme récursif pour générer les nombres de Fibonacci (suite)

- Cherchons donc une solution particulière donnée par $C(n) = c$
- En substituant cette solution dans notre récurrence, nous trouvons
 - $c - c - c = 1$. Alors $c = -1$.
- Puisque l'équation caractéristique homogène associée à notre récurrence pour $C(n)$ est:
 - $r^2 - r - 1 = 0$
- La solution générale de la récurrence homogène est alors donnée par:

$$\alpha \left(\frac{1 + \sqrt{5}}{2} \right)^n + \beta \left(\frac{1 - \sqrt{5}}{2} \right)^n$$

- La solution générale de notre récurrence inhomogène pour $C(n)$ est alors donnée par:

$$C(n) = \alpha \left(\frac{1 + \sqrt{5}}{2} \right)^n + \beta \left(\frac{1 - \sqrt{5}}{2} \right)^n - 1$$

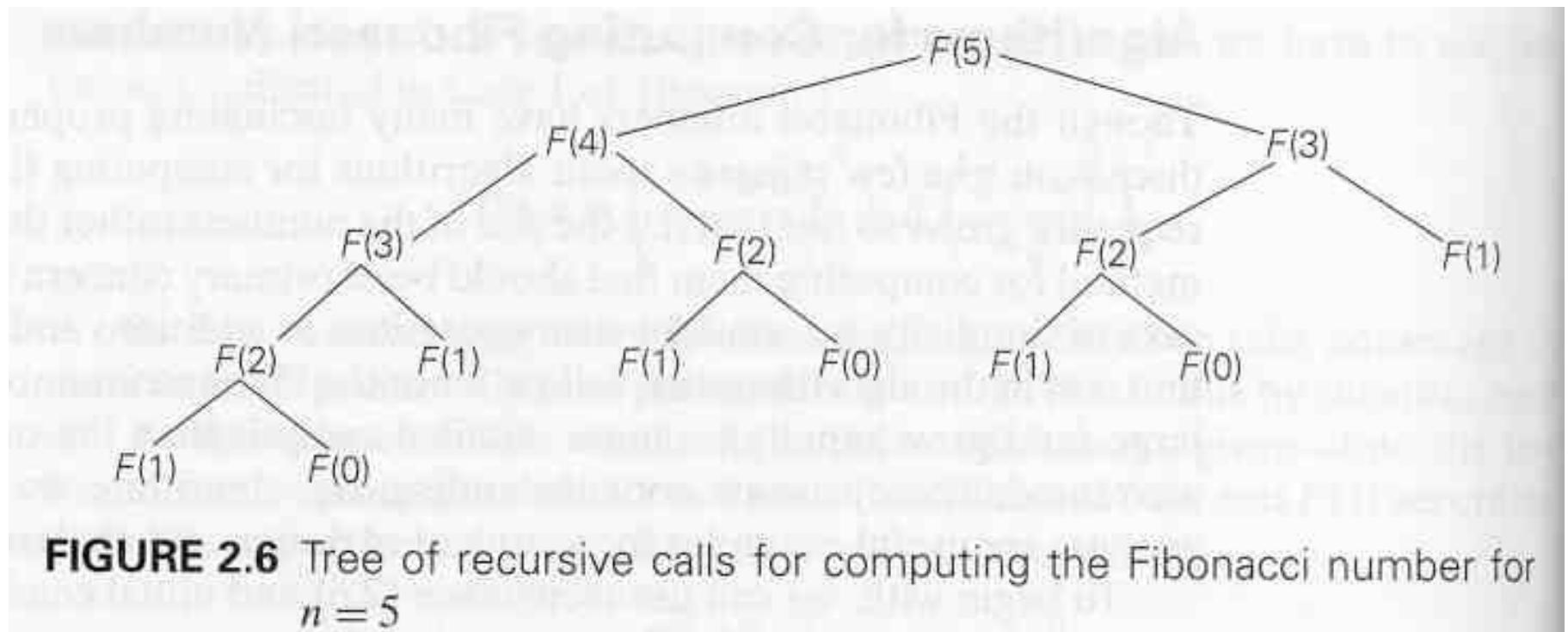


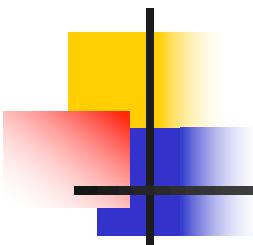
Algorithme récursif pour générer les nombres de Fibonacci (suite)

- La solution générale pour $C(n) + 1 = D(n)$ est donc identique à celle que nous avions obtenue pour $F(n)$.
- Les conditions initiales pour $D(n)$ sont, en fait, celles pour $F(n+1)$:
 - $D(0) = C(0) + 1 = 1 = F(1)$
 - $D(1) = C(1) + 1 = 1 = F(2)$
- On a alors que $D(n) = F(n+1)$) $C(n) = F(n+1) - 1$
- Le nombre d'additions augmente alors exponentiellement avec n
 - C'est donc une exponentielle d'exponentiel en la taille de n
- Une telle inefficacité vient du fait que chaque valeur de $F(n)$ est calculée plusieurs fois.

Algorithme récursif pour générer les nombres de Fibonacci (suite)

- En effet, FibRec(5) génère l'arbre ci-dessous d'appels récursifs:
 - Notez le nombre d'appels $F(1)$ généré par FibRec(5)
 - La concision d'un algorithme récursif n'est pas garantie de son efficacité.
 - Ici c'est particulièrement médiocre!





Algorithme non récursif pour générer les nombres de Fibonacci

ALGORITHME FibNonRec(n)

// Entrée: un entier $n \geq 0$

// Sortie: le n ième nombre de Fibonacci

```
If  $n=0$  return 0
 $u \leftarrow 0$ ;  $v \leftarrow 1$ 
for  $i \leftarrow 2$  to  $n$  do
     $v \leftarrow v + u$ 
     $u \leftarrow v - u$ 
return  $v$ 
```

- Si $v+u$ est notre opération de base, nous avons $C(n) = n - 1$ ($\forall n > 0$)
- Cet algorithme ne nécessite que $\Theta(n)$ additions.
- FibNonRec(n) est donc nettement plus efficace que FibRec(n)

- Chapitre 2
 - 2.1 The Analysis Framework
 - 2.2 Asymptotic Notations and Basic Efficiency Classes
 - 2.3 Mathematical Analysis of Nonrecursive Algorithms
 - 2.4 Mathematical Analysis of Recursive Algorithms