



# INFO0501

## ALGORITHMIQUE AVANCÉE

### COURS 1

#### PRÉSENTATION DE LA MATIÈRE, ORGANISATION ET INTRODUCTION



UNIVERSITÉ  
DE REIMS  
CHAMPAGNE-ARDENNE

Pierre Delisle  
Département de Mathématiques, Mécanique et Informatique  
Septembre 2018

# Pierre Delisle

- Maître de Conférences au département de Math-Meca-Info de l'URCA
- Mail
  - pierre.delisle@univ-reims.fr
- Site Web
  - <http://cosy.univ-reims.fr/~pdelisle>



# Plan de la séance

---

- Description de la matière
  - Objectifs, pré-requis, organisation
- Introduction/Rappels
  - Premiers algorithmes
    - tri par insertion
- Analyse des algorithmes
- Conception des algorithmes
  - Méthode diviser pour régner : tri par fusion
- Croissance des fonctions

# Objectifs de la matière

---

- Objectif général
  - Apprendre à utiliser des structures de données avancées en algorithmique et en programmation
- Compétences spécifiques
  - Définition et utilisation des tables de hachage
  - Définition, représentation et utilisation des graphes
  - Algorithmes de la théorie des graphes
  - Implémentations en langage C
- Pré-requis
  - Info0301 : Programmation en langage C
  - Info0401 : Algorithmique

# Organisation

---

- CM : 10 x 2 heures
  - Mardi 14h00-16h00
- TD : 10 x 2 heures
  - S5O6 : Jeudi 14h00 à 16h00
  - S5O7 : Vendredi 10h15-12h15
  - Les TD seront probablement fusionnés
- TP : 10 x 2 heures
  - S5O7a : Mardi 16h15-18h15
  - S5O7b : Mercredi 8h00-10h00
  - S5O6a : Vendredi 14h00-16h00
  - À confirmer

- Évaluation

Nature de l'évaluation	Nombre de points
DS	30
CRTP (TPs + Projet)	20
Examen Terminal	50
Total	100

- Changements possibles sur les groupes et sur les emplois du temps
- Surveillez votre bureau virtuel et le site de la Licence Info !

# Quelques informations pratiques...

---

- Emplois du temps et informations
  - Bureau virtuel
    - [ebureau.univ-reims.fr/](http://ebureau.univ-reims.fr/)
  - Site Web de la Licence Informatique
    - <http://www.licenceinfo.fr/>
  - Site Web de la scolarité
    - <http://www.univ-reims.fr/ufrsciences>
- Demi-groupes de TP
  - Voir sur le site de la Licence
- Supports de cours et énoncés de TD/TP
  - <http://cosy.univ-reims.fr/~pdelisle/>

# Structure de la matière

---

- Partie 1 : Introduction et rappels
  - Analyse et conception des algorithmes
  - Piles, files, listes chaînées, arbres, tas
- Partie 2 : Tables de hachage
  - Définition
  - Représentation
  - Principaux algorithmes
- Partie 3 : Graphes
  - Définition et représentation
  - Principaux algorithmes
    - Tri topologique
    - Connexité
    - Plus courts chemins
    - Arbres couvrants de poids minimal
    - Flux maximal
    - ...
- Bibliographie
  - T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, "Algorithmique", 3<sup>e</sup> édition, Dunod, 2010



# LE RÔLE DES ALGORITHMES EN INFORMATIQUE



# Algorithme

---

- Procédure de calcul bien définie qui
  - Prend en entrée une valeur ou un ensemble de valeurs
  - Donne en sortie une valeur ou un ensemble de valeurs
- Séquence d'étapes de calcul qui transforment l'entrée en sortie
- Outil permettant de résoudre un problème de calcul
- Des problèmes complexes à résoudre par des ordinateurs nécessitent des algorithmes performants
  - Calcul scientifique : simulation de systèmes
  - Internet : recherche et manipulation de grands volumes de données
  - Commerce électronique : sécurité, encryptage
  - Industrie : allocation de ressources
  - ...

# Algorithmes en tant que technologie

---

- Les ordinateurs
  - ne sont pas infiniment rapides
  - n'ont pas une mémoire infinie
- Le temps machine et l'espace mémoire sont des ressources limitées
- Une machine moins performante qui exécute de bons algorithmes pourra surpasser une machine plus performante qui exécute de mauvais algorithmes



# ANALYSE DES ALGORITHMES

# Exemple : le problème de tri

---

- Entrée
  - Une suite de  $n$  nombres  $\langle a_1, a_2, \dots, a_n \rangle$
- Sortie
  - Une permutation (réorganisation)  
 $\langle a_1', a_2', \dots, a_n' \rangle$   
de la suite donnée en entrée
  - De sorte que
$$a_1' \leq a_2' \leq \dots \leq a_n'$$
- La suite  $\langle 31, 41, 59, 26, 41, 58 \rangle$ 
  - Est une **instance** du problème de tri
- Opération majeure en informatique
  - Employée par une multitude de programmes comme phase intermédiaire
- L'algorithme optimal pour une application donnée dépend
  - Du nombre d'éléments à trier
  - De la façon dont les éléments sont plus ou moins triés initialement
  - Des restrictions sur les valeurs des éléments
  - ...



# Un 1<sup>er</sup> algorithme : le tri par insertion

---

- Algorithme naturel pour l'être humain
  - Comment triez-vous vos cartes quand vous jouez aux cartes (si vous jouez aux cartes !) ?
  - On prend les cartes une à une et on les place au bon endroit dans sa main
- Efficace quand il s'agit de trier un petit nombre d'éléments
- Exemple 1
  - Application sur l'instance  $\langle 5, 2, 4, 6, 1, 3 \rangle$

- Pseudo-code

```
TRI-INSERTION ( $t$ )  
  pour  $j = 2$  à  $t.\text{longueur}$   
     $\text{clé} = t[j]$   
    //insère  $t[j]$  dans la séquence triée  $t[1.. j - 1]$   
     $i = j - 1$   
    tant que  $i > 0$  et  $t[i] > \text{clé}$   
       $t[i + 1] = t[i]$   
       $i = i - 1$   
     $t[i + 1] = \text{clé}$ 
```

# Correction d'un algorithme

---

- Un algorithme est dit correct si, pour chaque instance en entrée, il se termine en produisant la bonne sortie
  - Un algorithme correct résoud le problème donné
- Un algorithme incorrect peut
  - Ne pas se terminer pour certaines instances
  - Se terminer sur une réponse autre que celle voulue
- On montre qu'un algorithme est correct par les invariants de boucle

# Invariant de boucle

---

- Propriété qui est vraie à chaque passage dans la boucle
- On doit montrer 3 choses concernant un invariant de boucle
  - **Initialisation** : il est vrai avant la première itération de la boucle
  - **Conservation** : s'il est vrai avant une itération de la boucle, il le reste avant l'itération suivante
  - **Terminaison** : Une fois terminée la boucle, l'invariant fournit une propriété utile qui aide à montrer la validité de l'algorithme

# Validité du tri par insertion

---

## ■ Invariant de boucle

Au début de chaque itération de la boucle **pour** le sous-tableau  $t[1 .. j - 1]$  se compose des éléments qui occupaient initialement les positions  $t[1 .. j - 1]$  mais qui sont maintenant triés

## ■ Initialisation

- Avant la 1ère itération de la boucle,  $j = 2$
- Le sous-tableau  $t[1 .. j - 1]$  se compose donc uniquement de l'élément  $t[1]$
- ... qui est l'élément originel de  $t[1]$
- ... qui est trié (trivialité : un élément seul est nécessairement trié)
- L'invariant est donc vérifié avant la 1ère itération de la boucle

## ■ Conservation

- Le corps de la boucle **pour** fonctionne en déplaçant  $t[j - 1]$ ,  $t[j - 2]$ ,  $t[j - 3]$ , etc. d'une position vers la droite jusqu'à ce qu'on trouve la bonne position pour  $t[j]$
- Le sous-tableau  $t[1 .. j]$  se compose alors des éléments situés initialement dans  $t[1 .. j]$ , mais en ordre trié
- L'incrément de  $j$  pour l'itération suivante de la boucle **pour** préserve alors l'invariant

## ■ Terminaison

- La condition forçant la boucle **pour** à se terminer est que  $j > t.\text{longueur} = n$
- Comme chaque itération de la boucle augmente  $j$  de 1, on doit avoir  $j = n + 1$  à cet instant
- En substituant  $n + 1$  à  $j$  dans la formulation de l'invariant de boucle, on a que le sous-tableau  $t[1 .. n]$  se compose des éléments qui appartenaient originellement à  $t[1 .. n]$ , mais qui ont été triés depuis
- Or, le sous-tableau  $t[1 .. n]$  est le tableau complet, donc le tableau tout entier est trié, et donc l'algorithme est correct



# Analyse des algorithmes

---

- Prévoir les ressources nécessaires à cet algorithme
  - Temps de calcul
  - Mémoire
  - Largeur de bande,
  - ...
- C'est généralement le temps de calcul qui nous intéresse
- En analysant plusieurs algorithmes pour un problème, on peut identifier le plus efficace
- Nécessite un modèle de la technologie employée (ressources, coûts) : le modèle RAM

# Modèle RAM

---

- Modèle de calcul générique basé sur une machine à accès aléatoire à processeur unique
- Instructions exécutées l'une après l'autre
- Permet les instructions
  - Arithmétiques (addition, soustraction, ...)
  - De transfert de données (lecture, copie, ...)
  - De contrôle (branchement, appel de routine, ...)
- Chaque instruction a un temps d'exécution constant
- Types de données : entier et réel (avec taille limitée)

# Analyse du tri par insertion

---

## Durée d'exécution de TRI-INSERTION

- Dépend de l'entrée
  - Temps pour 1000 nombres  $>$  temps pour 3 nombres
  - Temps peut être différent pour 2 entrées de même taille, selon qu'elles sont plus ou moins triées partiellement
- En général
  - Le temps d'exécution d'un algorithme croît avec la taille de l'entrée
  - On exprime donc le temps d'exécution en fonction de cette taille

## Taille de l'entrée

- Dépend du problème étudié
- Pour le problème de tri
  - Nombre d'éléments constituant l'entrée
  - Longueur  $n$  du tableau à trier
- Peut être le nombre total de bits nécessaire à la représentation de l'entrée en notation binaire
- Peut être plusieurs nombres plutôt qu'un seul
  - Pour les algorithmes de graphes
    - Nombre de sommets et nombre d'arcs

# Temps d'exécution

---

- Nombre d'opérations élémentaires exécutées
- On considère que chaque ligne de pseudo-code demande un temps constant
- Pour TRI-INSERTION
  - Temps d'exécution de la ligne  $i \rightarrow c_i$
  - On multiplie  $c_i$  par le nombre de fois que l'instruction est exécutée

# Analyse du temps d'exécution de TRI-INSERTION

TRI-INSERTION ( $t$ )

{1} pour  $j = 2$  à  $t.\text{longueur}$

{2}      $\text{clé} = t[j]$

{3}     //insère  $t[j]$  dans la séquence triée  $t[1..j-1]$

{4}      $i = j - 1$

{5}     tant que  $i > 0$  et  $t[i] > \text{clé}$

{6}          $t[i+1] = t[i]$

{7}          $i = i - 1$

{8}          $t[i+1] = \text{clé}$

coût

fois

$c_1$

$n$

$c_2$

$n - 1$

0

$n - 1$

$c_4$

$n - 1$

$c_5$

$\sum_{j=2}^n t_j$

$c_6$

$\sum_{j=2}^n (t_j - 1)$

$c_7$

$\sum_{j=2}^n (t_j - 1)$

$c_8$

$n - 1$

$t_j$  = nombre de fois que le test de la boucle **tant que** est exécuté pour cette valeur de  $j$

$n = t.\text{longueur}$

Temps d'exécution

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^n t_j + c_6 \sum_{j=2}^n (t_j - 1) + c_7 \sum_{j=2}^n (t_j - 1) + c_8(n-1)$$

Cas le plus favorable  
( $t$  déjà trié  $\rightarrow t_j = 1$ )

$$T(n) = (c_1 + c_2 + c_4 + c_5 + c_8) n - (c_2 + c_4 + c_5 + c_8)$$

Forme  $an + b \rightarrow$  **Fonction linéaire de  $n$**

Cas le plus défavorable  
( $t$  trié décroissant  $\rightarrow t_j = j$ )

$$T(n) = (c_5/2 + c_6/2 + c_7/2) n^2 + (c_1 + c_2 + c_4 + c_5/2 - c_6/2 - c_7/2 + c_8) n - (c_2 + c_4 + c_5 + c_8)$$

Forme  $an^2 + bn + c \rightarrow$  **Fonction quadratique de  $n$**

# Analyse du temps d'exécution

---

## Ce qui nous intéresse généralement le plus

- Temps d'exécution dans le cas le plus défavorable
  - Temps d'exécution maximal pour une quelconque entrée de taille  $n$
- Borne supérieure du temps d'exécution
  - Certitude qu'on ne pourra faire pire
  - Pour certains algorithmes, ce cas arrive souvent
  - Souvent, cas moyen presque aussi mauvais que le pire

## Ce qui nous intéresse vraiment

- *Taux de croissance* du temps d'exécution
  - Ou *ordre de grandeur* du temps d'exécution
- On ne considérera que le terme dominant d'une formule, sans les coefficients constants
  - $an + b = \Theta(n)$  (théta de  $n$ )
  - $an^2 + bn + c = \Theta(n^2)$  (théta de  $n$ -deux)
- Un algorithme est plus efficace qu'un autre si son temps d'exécution du cas le plus défavorable a un ordre de grandeur inférieur



# CONCEPTION DES ALGORITHMES

Méthode diviser-pour-régner

# Techniques de conception

---

- Tri par insertion → approche incrémentale
  - Après avoir trié  $t[1..j-1]$ , on produit  $t[1..j]$
- Approche diviser-pour-régner
  - Méthode récursive
  - L'algorithme s'appelle lui-même pour traiter des sous-problèmes similaires, mais de taille inférieure
  - Les solutions des sous-problèmes sont combinées pour produire la solution du problème original
- Les 3 étapes de l'approche diviser-pour-régner
  - Diviser
    - Création de sous-problèmes qui sont des instances plus petites du même problème
  - Régner
    - Traiter les sous-problèmes de façon récursive
    - Lorsque la taille d'un sous-problème est assez petite, on peut le résoudre directement
  - Combiner
    - Production de la solution du problème en combinant les solutions des sous-problèmes



# Exemple : Le tri par fusion

---

- Diviser
  - Diviser la suite de  $n$  éléments à trier en 2 sous-suites de  $n/2$  éléments chacune
- Régner
  - Trier les 2 sous-suites récursivement avec le tri par fusion
- Combiner
  - Fusionner les 2 sous-suites triées pour produire la réponse
- Arrêt de la récursivité
  - Séquence de longueur 1
- Exemple 2
  - Application de la fusion sur l'instance  $\langle 2, 4, 5, 7, 1, 2, 3, 6 \rangle$

# La procédure FUSION

---

```
FUSION ( $t, p, q, r$ )  
   $n_1 = q - p + 1$   
   $n_2 = r - q$   
  Créer tableaux  $g[1..n_1 + 1]$  et  $d[1..n_2 + 1]$   
  pour  $i = 1$  à  $n_1$   
     $g[i] = t[p + i - 1]$   
  pour  $j = 1$  à  $n_2$   
     $d[j] = t[q + j]$   
   $g[n_1 + 1] = \infty$   
   $d[n_2 + 1] = \infty$   
   $i = 1$   
   $j = 1$   
  pour  $k = p$  à  $r$   
    Si  $g[i] \leq r[j]$   
       $t[k] = g[i]$   
       $i = i + 1$   
    Sinon  
       $t[k] = d[j]$   
       $j = j + 1$ 
```

- 2 premières boucles pour
  - $\Theta(n_1 + n_2) \rightarrow \Theta(n)$
- 3<sup>e</sup> boucle pour
  - $n$  itérations ( $n = r - p + 1$ )
  - Temps constant pour chaque itération
  - $\Theta(n)$
- Temps d'exécution
  - $\Theta(n)$

# La procédure TRI-FUSION

---

```
TRI-FUSION ( $t, p, r$ )  
  si  $p < r$   
     $q = \lfloor (p + r) / 2 \rfloor$   
    TRI-FUSION ( $t, p, q$ )  
    TRI-FUSION ( $t, q + 1, r$ )  
    FUSION( $t, p, q, r$ )
```

- Exemple 3

Application du tri par fusion sur l'instance  
 $\langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$

- Analyse ?

# Analyse des algorithmes diviser-pour-régner

---

- Algorithme avec appel récursif à lui-même
  - Temps d'exécution décrit par une réurrence
  - Décrit le temps d'exécution global pour un problème de taille  $n$  à partir du temps d'exécution pour des entrées de taille moindre
  - On peut alors se servir d'outils mathématiques pour résoudre la réurrence
- $T(n)$ 
  - Temps d'exécution d'un problème de taille  $n$
- Si la taille du problème est suffisamment petite ( $n \leq c$ )
  - La solution directe prend un temps constant  $\rightarrow \Theta(1)$
- Si on divise le problème en  $a$  sous-problèmes
  - La taille de chacun étant  $1/b$  de la taille du problème initial
  - Il faut le temps  $T(n/b)$  pour résoudre un sous problème de taille  $n/b$
  - Il faut  $aT(n/b)$  pour résoudre  $a$  sous-problèmes
  - Il faut un temps  $D(n)$  pour diviser le problème en sous-problèmes
  - Il faut un temps  $C(n)$  pour construire la solution finale
- On a donc la réurrence

$$T(n) = \begin{cases} \Theta(1) & \text{si } n \leq c \\ aT(n/b) + D(n) + C(n) & \text{sinon} \end{cases}$$

# Analyse du tri par fusion

---

- Diviser
  - On calcule le milieu du sous-tableau
  - $D(n) = \Theta(1)$
- Régner
  - On résout récursivement 2 sous-problèmes, chacun ayant la taille  $n / 2$
  - $2T(n / 2)$
- Combiner
  - Utiliser la procédure fusion sur un sous-tableau à  $n$  éléments
  - $C(n) = \Theta(n)$

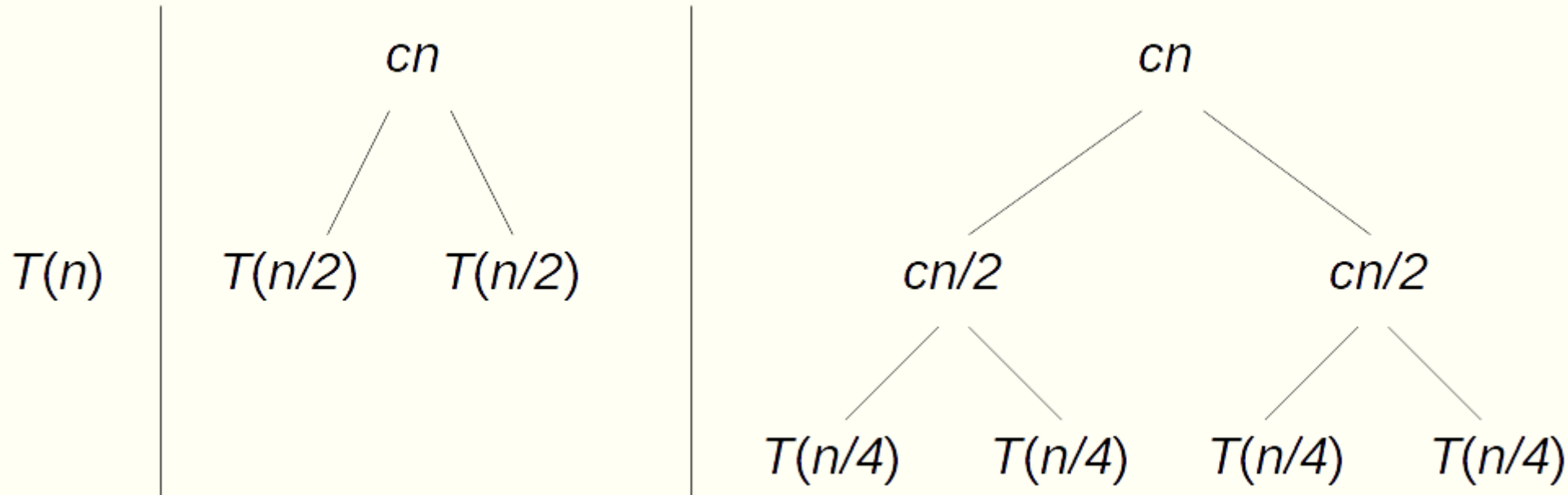
$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ 2T(n / 2) + \Theta(n) & \text{si } n > 1 \end{cases}$$

# Arbre récursif

---

- $c$  :
- Temps requis pour résoudre des problèmes de taille 1
  - Temps par élément de tableau des étapes diviser et combiner

$$T(n) = \begin{cases} c & \text{si } n = 1 \\ 2T(n/2) + cn & \text{si } n > 1 \end{cases}$$



# Arbre récursif

Coût par niveau

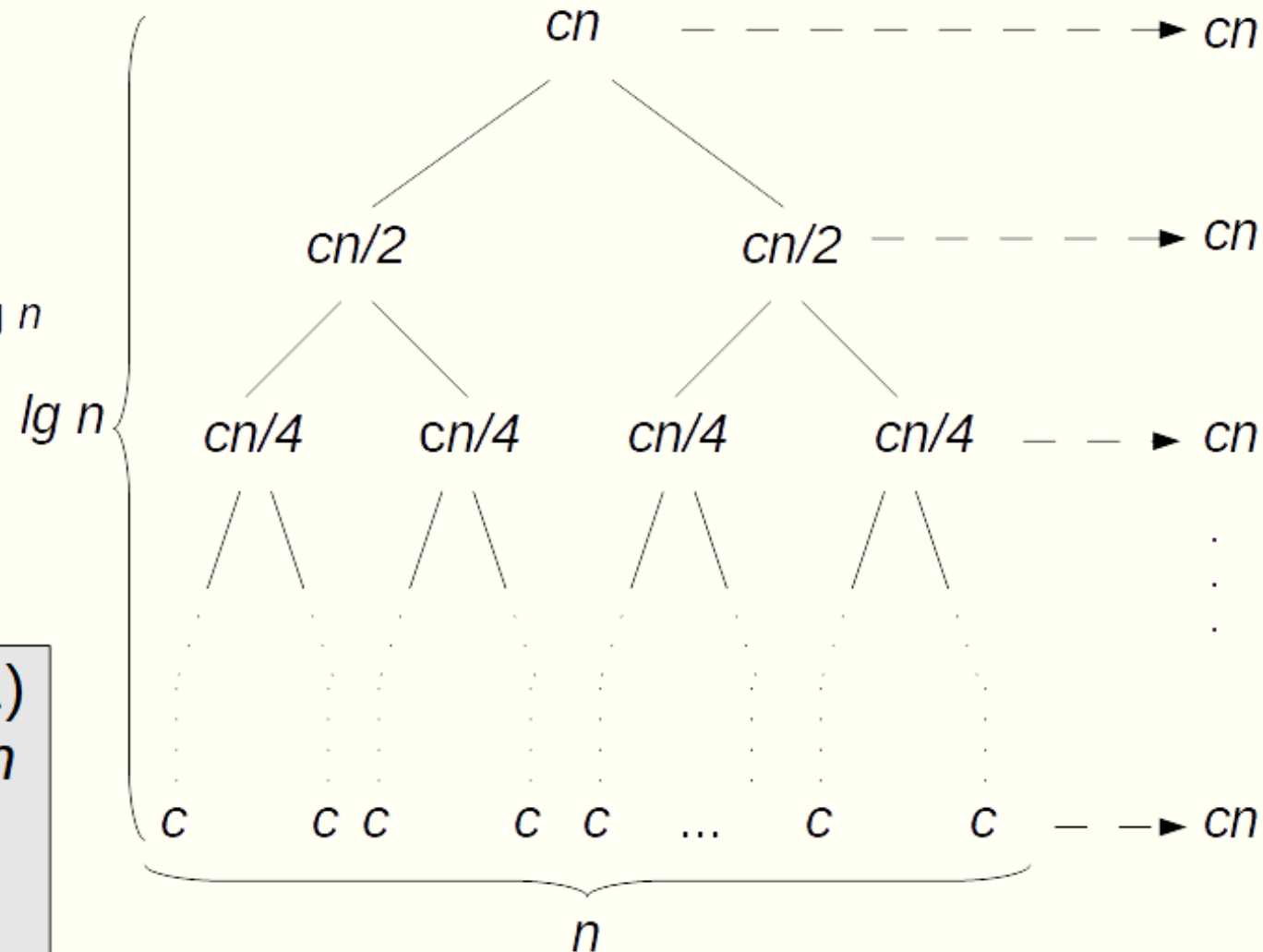
$cn$

Arbre de hauteur  $\lg n$

$\lg n + 1$  niveaux

Coût total

$$\begin{aligned} & cn (\lg n + 1) \\ &= cn \lg n + cn \\ &= \Theta(n \lg n) \end{aligned}$$






# CROISSANCE DES FONCTIONS

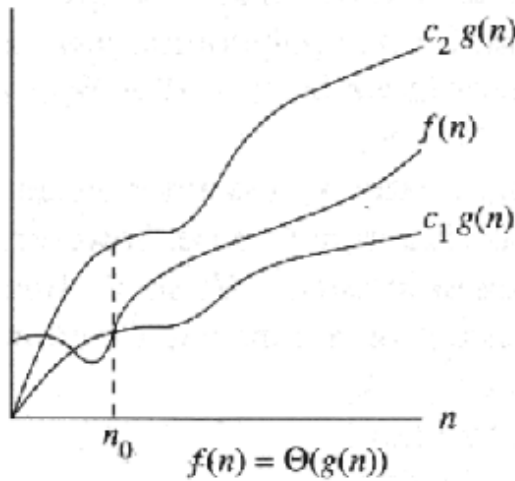


# Ordre de grandeur du temps d'exécution

---

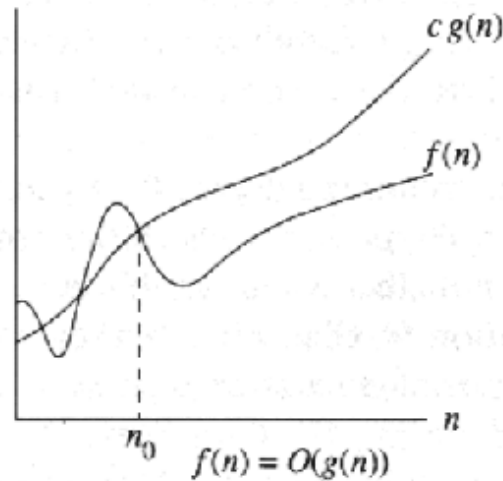
- Caractérisation de l'efficacité de l'algorithme
- Comparatif des performances relatives de plusieurs algorithmes pour un même problème
  - Tri par insertion :  $\Theta(n^2)$  dans le pire des cas
  - Tri par fusion :  $\Theta(n \lg n)$  dans le pire des cas 
- Performance asymptotique
  - Augmentation du temps d'exécution à la limite, quand la taille de l'entrée augmente indéfiniment

# Notation asymptotique



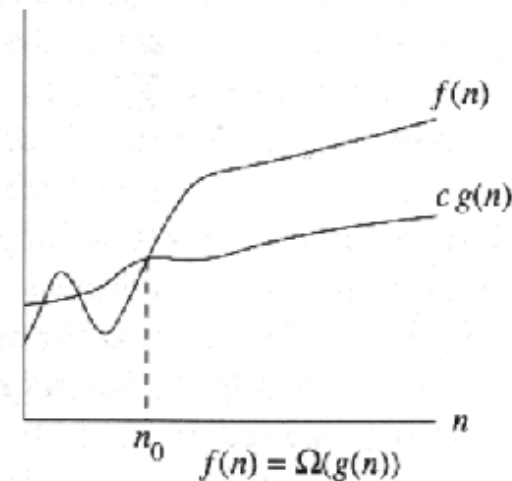
Il existe des constantes positives  $c_1$  et  $c_2$  telles que  $f(n)$  puisse être prise en sandwich entre  $c_1 g(n)$  et  $c_2 g(n)$ , pour  $n$  assez grand

(borne asymptotiquement approchée)



Il existe une constante positive  $c$  telle que la valeur de  $f(n)$  est inférieure ou égale à  $g(n)$ , pour  $n$  assez grand

(borne supérieure asymptotique)



Il existe une constante positive  $c$  telle que la valeur de  $f(n)$  est supérieure ou égale à  $g(n)$ , pour  $n$  assez grand

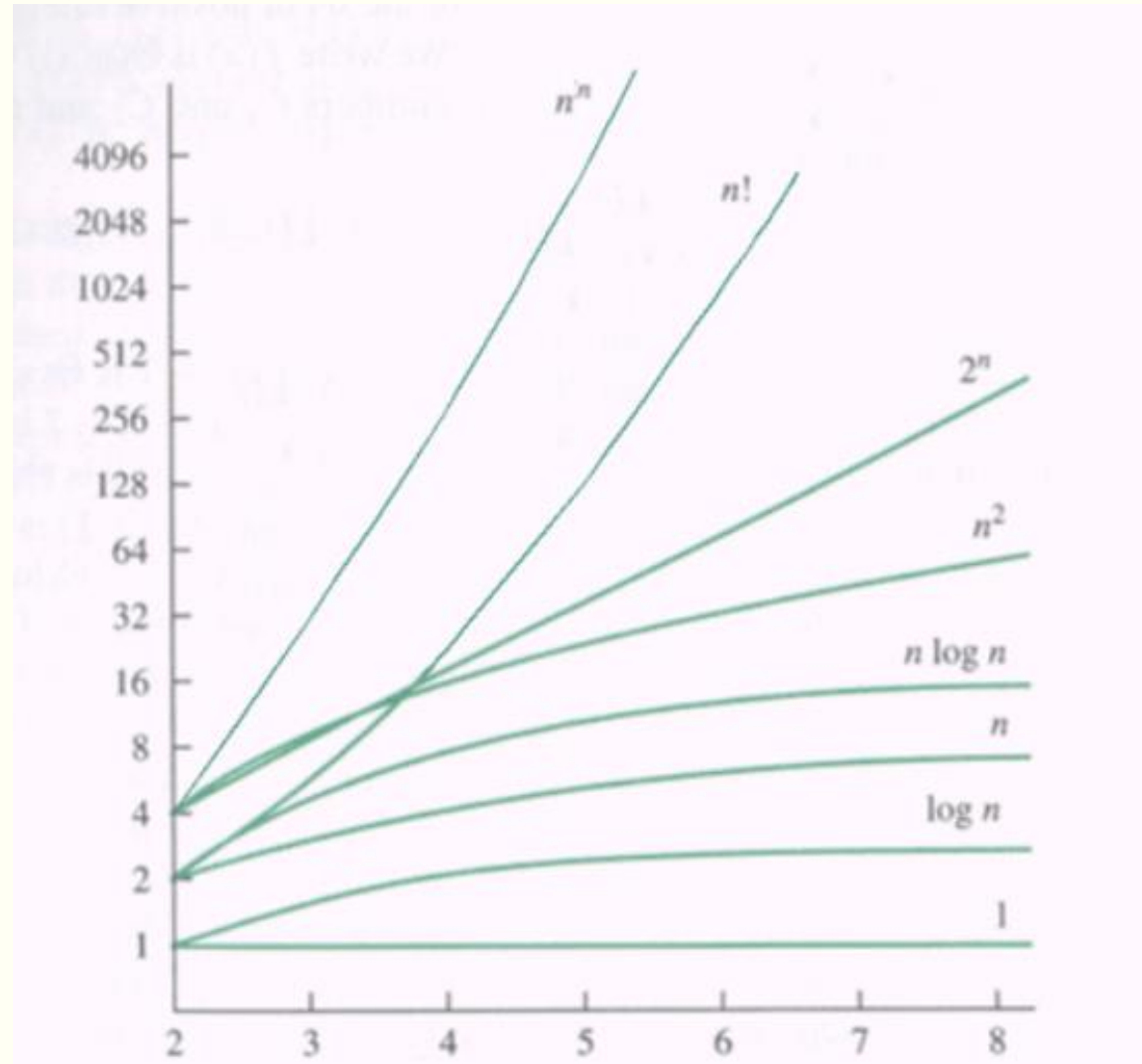
(minorant asymptotique)

Temps d'exécution du tri par insertion

- $\Omega(n)$
- $O(n^2)$

# Ordre de grandeur de quelques fonctions parmi les plus connues

---





PROCHAIN COURS

STRUCTURES DE DONNÉES  
ÉLÉMENTAIRES

TAS, PILES, FILES, LISTES CHAÎNÉES, ARBRES