Complexité

Donner un ordre de grandeur de la durée d'exécution du programme, indépendamment de la machine sur lequel il est exécuté

- Quelques rappels mathématiques indispensables en informatique
- Notations asymptotiques ou comment négliger les constantes
- Recettes pour évaluer la complexité de programmes itératifs

Rappels mathématiques

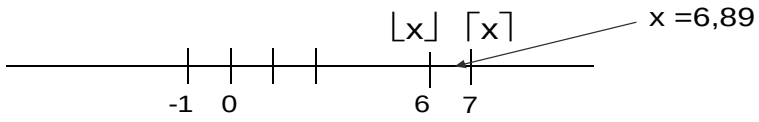
Sommations

- $\sum_{i=0}^{n} f(i) = f(0) + f(1) + \ldots + f(n)$
- $\sum_{i=0}^{n} i = n(n+1)/2$
- $\sum_{i=0}^{n} a\, r^{i} = a\,(r^{n+1} - 1)/(r - 1)$

Parties entières

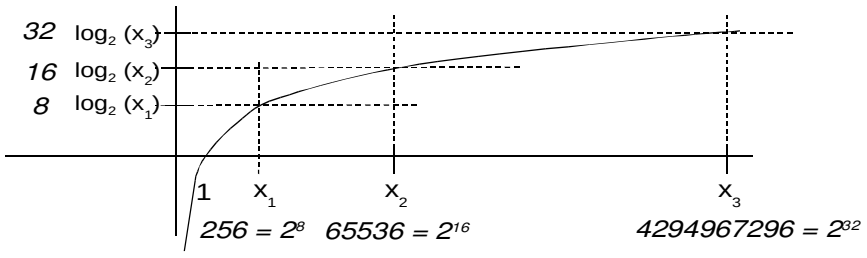
Soit x un nombre réel.

- $\lfloor x \rfloor$ est le plus grand entier inférieur ou égal à x
- $\lceil x \rceil$ est le plus petit entier supérieur ou égal à x



Logarithmes

Pour b>1, x>0, y est le logarithme en base b de x si et seulement si $b^y = x$ (noté $\log_b x = y$)



croissance très lente !

- Propriétés :

Pour b et c >1

- $\log_b b^a = a$
- $\log_b (x\ y) = \log_b (x) + \log_b (y)$
- $\log_b (x^a) = a \log_b (x)$
- $\log_c (x) = \log_b (x) / \log_b (c)$
- $\forall$ n entier strictement positif, $\exists$ k tel que $2^k \leq n < 2^{k+1}$

Notations asymptotiques ou comment négliger les constantes

- Quelles sont les opérations élémentaires significatives ?

```
int j = 8;
for (int i=0;i<tab.length-1;i++) {
    int k = tab[i];
    if (tab[i] < tab[i-1]) tab[i] = tab[i] + tab[i+1];
}
```

- Que choisir entre un algorithme qui fait $n^3/2$ opérations et un algorithme qui fait $5n^2$ opérations ?
  - si n=3, $n^3/2$ vaut 13,5 et $5n^2$ vaut 45
  - si n=100, $n^3/2$ vaut 500 000 et $5n^2$ vaut 5000

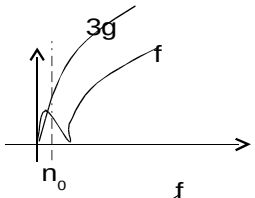
Besoin d'ignorer les facteurs constants et les petites valeurs des entrées $\rightarrow$ notations asymptotiques

Ordres de grandeur des fonctions

- $O(g)$

  f est un « grand O de g » ssi

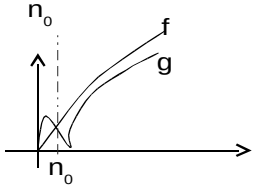
  $\exists\, c > 0, \exists\, n_0 > 0 \;\forall\, n \geq n_0,\ f(n) \leq c\, g(n)$



- $\Omega(g)$

  f est un « grand oméga de g » ssi

  $\exists\, c > 0, \exists\, n_0 > 0 \;\forall\, n \geq n_0,\ f(n) \geq c\, g(n)$

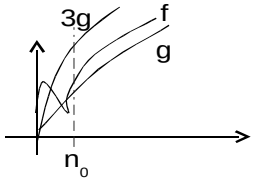


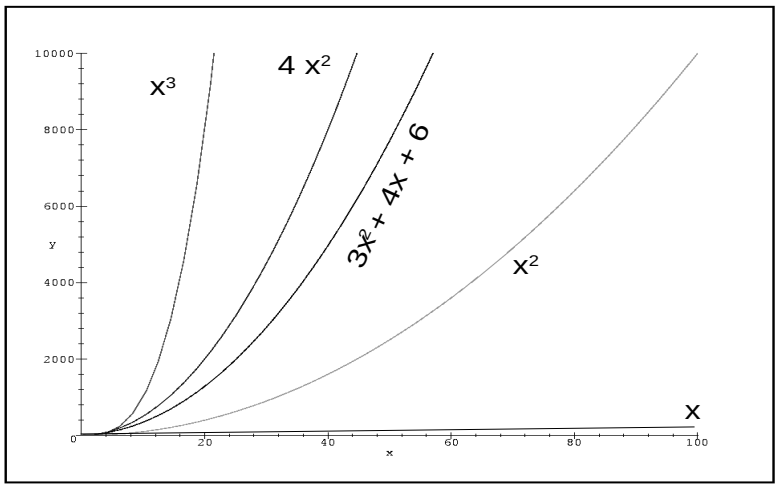
- $\Theta(g)$

  f est « du même ordre que g » ssi

  f est un « grand O de g » et

  f est un « grand oméga de g »

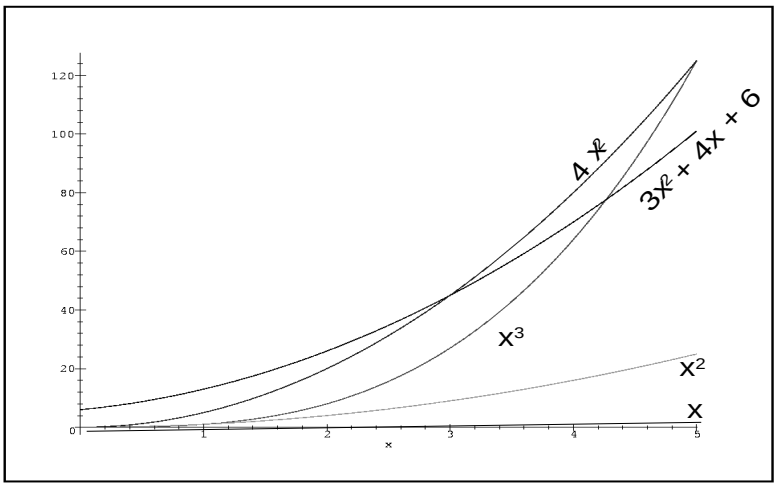




$3x^2 + 4x + 6 = \Theta(x^2)$

$= O(x^3)$

$= \Omega(x)$



Propriétés

- $f = \Theta(g) \Leftrightarrow g = \Theta(f)$
- $O(c\,f) = O(f)$ si c constante — *peu importe les constantes*
  vrai aussi pour Θ et Ω
- $O(f + g) = O(\max(f,g))$
  vrai aussi pour Θ et Ω
- $\sum_{i=0}^{d} a_{d-i}\, n^{d-i} = \Theta(n^d)$ — *un polynôme de degré d est en $\Theta(n^d)$*
- $\sum_{i=0}^{d} a_{d-i}\, n^{d-i} = O(n^{d+1})$ — *un polynôme de degré d est en $O(n^{d+1})$*
- $\log_a n = \Theta(\log_2 n)$ — *peu importe la base du logarithme*

Complexités les plus courantes

- $\Theta(n)$ algorithme linéaire
- $\Theta(n^2)$ algorithme quadratique
- $\Theta(n^3)$ algorithme cubique
- $\Theta(\log_2 n)$ algorithme logarithmique
- $\Theta(n \log_2 n)$

- $\Theta(2^n)$ algorithme exponentiel

Un algorithme $\mathcal{A}$ est un :

- $O(g)$: si $\mathcal{A}$ effectue *au plus* de l'ordre de g opérations
  "borne supérieure"

- $\Omega(g)$: si $\mathcal{A}$ effectue *au moins* de l'ordre de g opérations
  "borne inférieure"

- $\Theta(g)$: si $\mathcal{A}$ effectue *exactement* de l'ordre de g opérations

- Temps d'exécution avec un µproc 100 MIPS

(10^9 instructions par s)

n	**Θ(n)**	**Θ($\log_2$ n)**	**Θ(n $\log_2$ n)**	**Θ(n^2)**	**Θ(2^n)**
5	$5\ 10^{-9}$ s	$3\ 10^{-9}$ s	$12\ 10^{-9}$ s	$25\ 10^{-9}$ s	$32\ 10^{-9}$ s
10	10^{-8} s	$4\ 10^{-9}$ s	$3\ 10^{-8}$ s	10^{-7} s	10^{-6} s
1000	10^{-6} s	10^{-8} s	10^{-5} s	10^{-3} s	$3\ 10^{282}$ siècles
10^6	10^{-3} s	$2\ 10^{-8}$ s	$2\ 10^{-2}$ s	10^3 s 20mn	siècles

- Nombre d'éléments que l'on peut traiter en 24h

n	**$\log_2$ n**	**n $\log_2$ n**	**n^2**	**2^n**
$9\ 10^{13}$	10^{3e13}	$2\ 10^{12}$	10^7	32

Complexité

Complexité en fonction de quoi ?

de la taille des données auxquelles s'appliquent l'algorithme

- *sur un entier* : l'entier (ou la taille nécessaire pour le coder en binaire)
- *sur une chaîne de caractères* : longueur de la chaîne
- *sur un objet java* : taille des attributs
- *sur un tableau* : nombre d'éléments du tableau

  NB: si on applique sur chaque élément une méthode dont la complexité dépend de la taille des éléments, il faut en tenir compte.
- *sur un arbre* : nombre d'éléments de l'arbre
- *sur un graphe* : nombre de sommets et de liaisons

Complexité en espace

Espace mémoire nécessaire pour stocker les données (ici, on se préoccupe des constantes).

Exemple : pour trier un tableau contenant n éléments, la complexité en espace optimale est n; certains algorithmes utilisent 2 x n.

Complexité en temps

Ordre de grandeur du temps d'exécution de l'algorithme (indépendamment de la machine) en fonction de la taille des données

Cette complexité peut dépendre de la configuration des données

- Complexité dans le meilleur des cas
  Temps d'exécution le plus faible; utile pour vérifier que l'algorithme ne perd pas de temps inutilement
  Exemple : algorithme de tri appliqué à un tableau déjà trié !

- Complexité dans le pire des cas
  La plus importante. Donne une borne supérieure du temps d'exécution.

  Commenter chaque méthode java avec sa complexité dans le pire des cas

- Complexité en moyenne
  Intéressante quand on sait la calculer; donne le temps d'exécution moyen, quand on traite successivement des données n'ayant aucune propriété particulière.

Soit D_n l'ensemble des données de taille n. Soit I un sous ensemble de D_n et soit t(I) le nombre d'opérations élémentaires pour exécuter I.

- Complexité dans le meilleur des cas
  meilleur(n) = min $\{t(I), I \in D_n\}$

- Complexité dans le pire des cas
  pire(n) = max $\{t(I), I \in D_n\}$

- Complexité en moyenne
  moyenne(n) = $\sum_{I \subseteq D_n}$ Pr (I) t(I) où Pr(I) est la probabilité de I

Complexité temporelle des algorithmes itératifs

- Les instructions élémentaires (affectation, comparaison) sont en temps constant

- Séquence de blocs

  I_1 ;

  I_2 ;

  est de complexité Θ *(max($f_1(n)$,$f_2(n)$))*

  où $\Theta(f_1(n))$ est la complexité de I_1 et $\Theta(f_2(n))$ la complexité de i_2

- If then else

  if (C) I_1 ; else I_2 ;

  est de complexité O(max(f(n),$f_1(n)$,$f_2(n)$))

  où $\Theta(f(n))$ est la complexité de C, $\Theta(f_1(n))$ la complexité de I_1 et $\Theta(f_2(n))$ la complexité de i_2

- Itération for

  for (int i=0 ; i< n ; i++)

  I ;

  est de complexité Θ *(n (f(n)))*

  si I n'a aucun effet sur les variables i et n et que $\Theta(f(n))$ est la complexité de I.

  Si, la complexité de I dépend de i, la complexité est en $\sum_{i=0}^{n} \Theta(f(i))$

- Itération while

  While (C)

  I;

  est de complexité Θ *(g(n)*max(f_1 (n),f_2 (n)))*

  si C est en $\Theta(f_1(n))$, I en $\Theta(f_2(n))$ et que la boucle while est exécutée $\Theta(g(n))$

Exemple : Recherche du maximum d'un tableau

```
public static int chercheMax(int[] tab) {
   int maxCourant = tab[0];
   for (int i=1;i<tab.length;i++)
       if (maxCourant < tab[i])
              maxCourant = tab[i];
   return maxCourant;
}
```

- taille des données : n = tab.length
- opérations comptées : comparaison ou affectation ?
- pire(n) = meilleur(n) = moyenne(n) = Θ(n)

Exemple : Recherche d'un élément dans un tableau

```
public static int recherche(int[] tab, int x) {
   for (int i=0;i<tab.length;i++)
       if (x==tab[i]) return i;
   return -1;
}
```

- Taille des données : n = tab.length
- Opération effectuée dans la boucle : une comparaison d'entiers en Θ(1)
- Complexité dans le meilleur des cas : Θ(1)
  x est l'élément d'indice 0; une comparaison et sortie de la boucle
- Complexité dans le pire des cas : Θ(n)
  sortie de la boucle quand i = tab.length
  correspond à quelles données ?

- Complexité en moyenne

  On suppose que *les éléments du tableau sont distincts* et que si x est dans le tableau, il peut être placé n'importe où, avec la même probabilité.

  - complexité en moyenne quand x ***est*** dans le tableau :
    - Les données qui contiennent x sont les tableaux qui contiennent x à l'indice 0, les tableaux qui contiennent x à l'indice 1, …, les tableaux qui contiennent x à l'indice i, … , les tableaux qui contiennent x à l'indice n-1.
    - Quand x est dans le tableau, la probabilité pour que x soit à la place i est 1/n (1 chance sur n possibilités)
    - Quand x est à la place i on fait i+1 comparaisons

$$\text{moyenne}_{\text{trouvé}}(n) = \sum_{\substack{I \subseteq D_n \\ x \in I}} Pr(I)\, t(I) = \sum_{i=0}^{n-1} (1/n) * (i+1)$$

$$\text{moyenne}_{\text{trouvé}}(n) = \sum_{i=0}^{n-1} (1/n) * (i+1) = 1/n * \sum_{i=0}^{n-1} (i+1)$$

$$= 1/n * \sum_{i=1}^{n} i$$

$$= 1/n * n(n+1)/2$$

$$= (n+1)/2$$

- complexité en moyenne quand x ***n'est pas*** dans le tableau

Quelle que soit la donnée, il y a n comparaisons.

$\text{moyenne}_{\text{pasTrouvé}}(n) = n$

- La complexité en moyenne est

  $moyenne(n) = p\ moyenne_{trouvé}(n) + (1-p)\ moyenne_{pasTrouvé}(n)$

où p est la probabilité pour que x soit dans le tableau.

moyenne(n) = p * (n + 1)/2 + (1-p) * n

- Si x est dans le tableau ou x n'est pas dans le tableau, on retrouve les complexités précédentes
- Si x a 50% de chance d'être dans le tableau, on fait en moyenne (n + 1)/4 + n / 2 comparaisons c'est à dire environ 3 comparaisons sur 4.