M415 : Notions de Complexité

Stéphane Perennes

DUT INFO - IUT Nice Côte d'Azur

1er mars 2022

Ce chapitre reprend certains supports du cours de M. Syska, H. Collavizza et M. Gaetano, aussi modifié avec N. Stolfi

Introduction

- Estimer la durée d'exécution du programme, comme celui-ci dépend de la machine utilisée on cherche un **ordre de grandeur**
- Autrement dit : Pour une entrée de taille n, quel est l'ordre de grandeur, (fonction de n), du nombre d'opérations $T(n)$ qu'il va effectuer ?
- Est ce que mon algorithme induit un programme rapide ?
- Existe t-il un algorithme/programme beaucoup plus rapide ?
- Quel est l'espace mémoire utilisé pour une entrée de taille n.

Rappels mathématiques : sommations usuelles

- $\sum_{i=0}^{n} f(i) = f(0) + \ldots + f(n) = sum([i\ for\ i\ in\ range(n+1)])$
- $\sum_{i=0}^{n} i = \frac{n(n+1)}{2} \sim \frac{n^2}{2}$ (on parle de série arithmétique)
- $\sum_{i=0}^{n} i^k \sim \frac{i^{k+1}}{k+1}$ (legère approximation)
- $\sum_{i=0}^{n} aq^i = a\frac{1-q^{n+1}}{1-q}, \forall q \neq 1, \forall a \in \mathbb{R}$
- $\sum_{i=0}^{n} aq^i \sim a\frac{q^{n+1}}{q-1}, \forall q > 1, \forall a \in \mathbb{R}$
- $\sum_{i=0}^{n} aq^i$ est bornée par $\frac{a}{1-q}$ si $q < 1$.

Parties entières

Soit $x \in \mathbb{R}$. On note :

- $\lfloor x \rfloor$ le plus grand entier inférieur ou égal à x.
- $\lceil x \rceil$ le plus petit entier supérieur ou égal à x.
- Fonction conventionellement nommées *ceil*() ($\lceil \rceil$ for ceiling,plafond) et *floor*() ($\lfloor \rfloor$ pour plancher).

Exemple

Prenons $x = 6.23$, alors $\lfloor x \rfloor = 6$ et $\lceil x \rceil = 7$

Logarithmes

Pour $b > 1, x > 0, y$ est le logarithme en base b de x $\Leftrightarrow b^y = x$

- On note : $y = log_b(x)$ et $log_b(x) = \frac{ln(x)}{ln(b)}$.
- Le logarithme est la **réciproque** (en un sens l'inverse) de la fonction exponetielle.
- *ln* est le logarithme naturel ou logarithme népérien.
- $\ln(e^x) = x$, $e = 2.718\ldots = exp(1)$.
- la dérivée (la pente de la courbe) de $ln(x)$ est $\frac{1}{x}$.
- la fonction logarithme est croissante et tend vers l'infini, mais sa croissance très lente.
- $\sum_{i=1}^{i=n} \sim \ln(n)$ (on parle de **série harmonique**).

Logarithmes

Exemples

logarithme en base 2 : on notera $log_2 = log$.

$$log(256) = log(2^8) = 8log(2) = 8$$

$$log(65536) = log(2^{16}) = 16log(2) = 16$$

$$log(4294967296) = log(2^{32}) = 32log(2) = 32$$

Ordre de grandeur

- $log2(1000) \sim 10$ (car $2^{10} = 1024 \sim 1000$
- $log2(x) = 10 \rightarrow x$ de l'ordre de 1000
- $log2(x) = 20 \rightarrow x$ de l'ordre de 10^6
- $log2(x) = 30 \rightarrow x$ de l'ordre de 10^9

Propriétés du logarithme

Soient deux réels $b > 1$ et $c > 1$.

- $log_b b^a = a$
- $log_b(xy) = log_b x + log_b y$
- $log_b(x^a) = a log_b(x)$
- $log_c(x) = \frac{log_b(x)}{log_b(c)}$
- $\forall n \in \mathbb{N}^*, \exists k, 2^k \leq n \leq 2^{k+1}$

Exponentielle

- $2^n = 2 \times 2^{n-1}$ pour n un entier positif,
  $2^0 = 1, 2^1 = 2, 2^2 = 4 \ldots$
- $2^x = exp(\ln 2 \times x)$ pour $x \in R$.
- $exp(x) = e^x = \sum_{i=0}^{i=\infty} x^i / i!$.
- $exp(1) = e^1 = ee = 2.718 \ldots$.
- la dérivée (la pente de la courbe) de $exp(x)$ est $exp(x)$.
- la fonction exponentielle est croissante et tend vers l'infini
- sa croissance est très rapide.
- la fonction exponentielle croit plus vite que tout polynome.
- $2^{2x} = (2^x)^2$.

Opérations élémentaires significatives

- Qu'est-ce que l'on compte ?

```
1 int j = 8;
2 for (int i=0; i<tab.length-1; i++) {
3   int k = tab[i];
4   if (tab[i] < tab[i-1])
5     tab[i] = tab[i] + tab[i+1];
6 }
```

- 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 500000 et $5n^2$ vaut 50000

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

Ordre de grandeur des fonctions (positives)

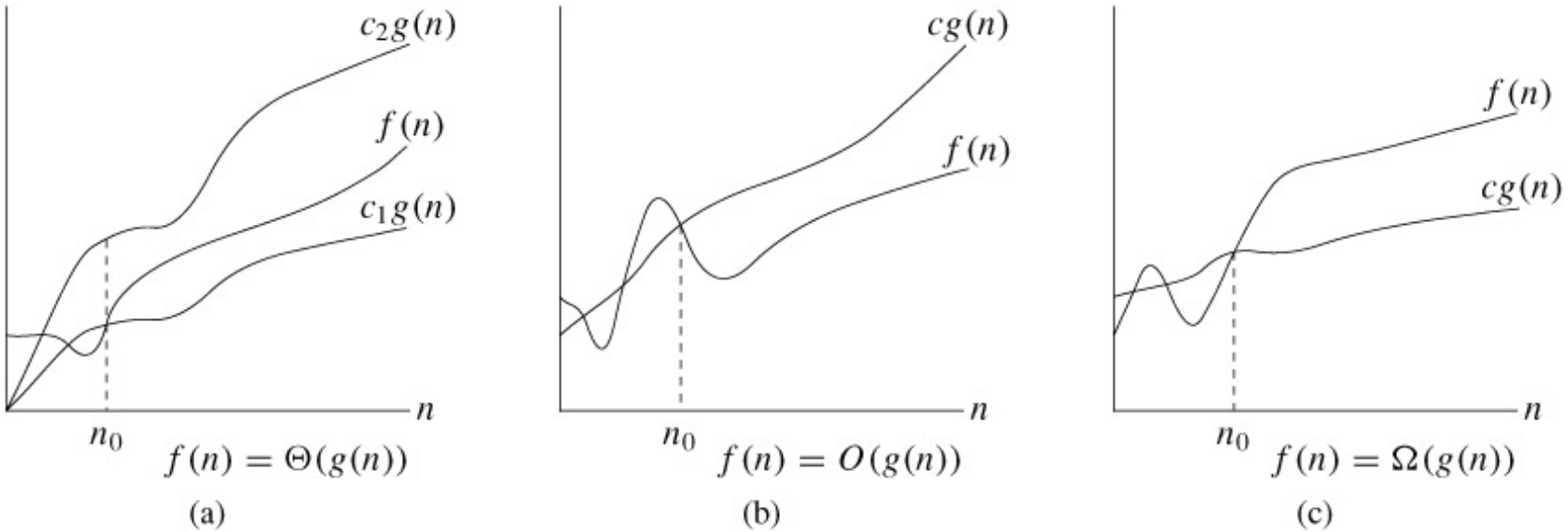


Figure – Notations Θ, O, Ω (voir [1])

1. [?]

Ordre de grandeur des fonctions (positives) : Notation O

On dit que g domine f (en $+\infty$) lorsqu'il existe des constantes n_0 et C telles que $\forall n > n_0 \quad f(n) \leq Cg(n)$. On le note $f = O(g)$.

Exemple

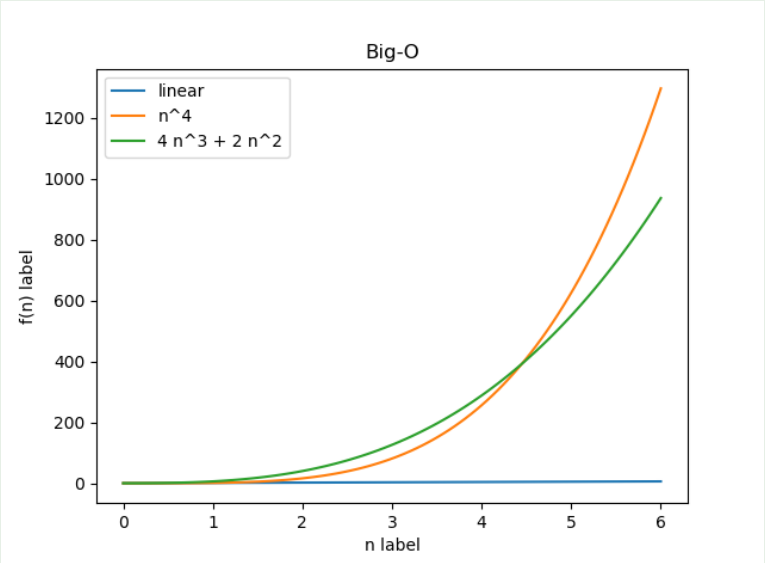


Figure – $4n^3 + 2n^2 = O(n^4)$

Avec $C = 1$ et $n_0 = 100$. $100^4 = 10^8$ et $4 \times 100^3 + 2 \times 100^2 = 4.02 \times 10^6$

Ordre de grandeur des fonctions (positives) : Notation Ω

On dit que g est soumise à f lorsqu'il existe des constantes n_0 et C telles que $\forall n > n_0\ Cg(n) \leq f(n)$. On le note $f = \Omega(g)$.

Exemple

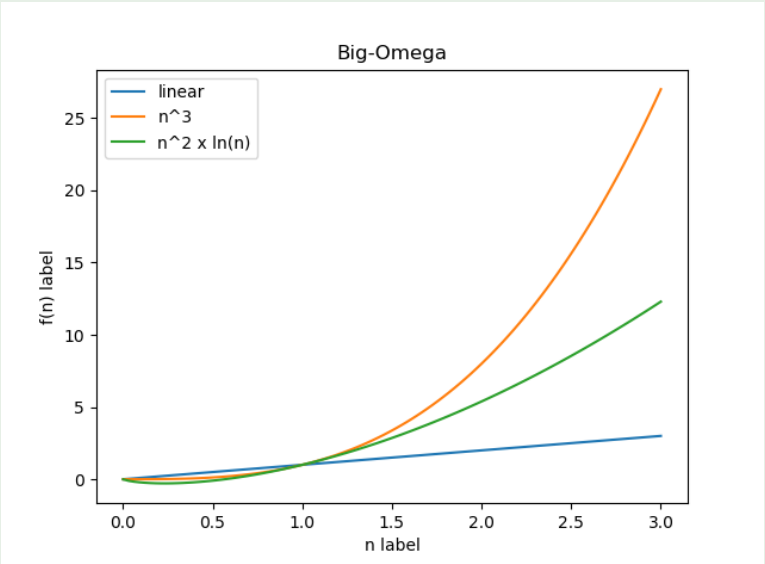


Figure – $n^3 = \Omega(n^2 ln(n))$

Avec $C = 1$ et $n_0 = 5$. $5^3 = 125$ et $5^2 ln(5) \cong 40.23594780$

Ordre de grandeur des fonctions (positives) : Notation Θ

On dit que f est du **même ordre** que g lorsque f est dominée et soumise à g c'est à dire lorsque $f = \Omega(g)$ et $f = O(g)$. On note :

$$f = \Theta(g)$$

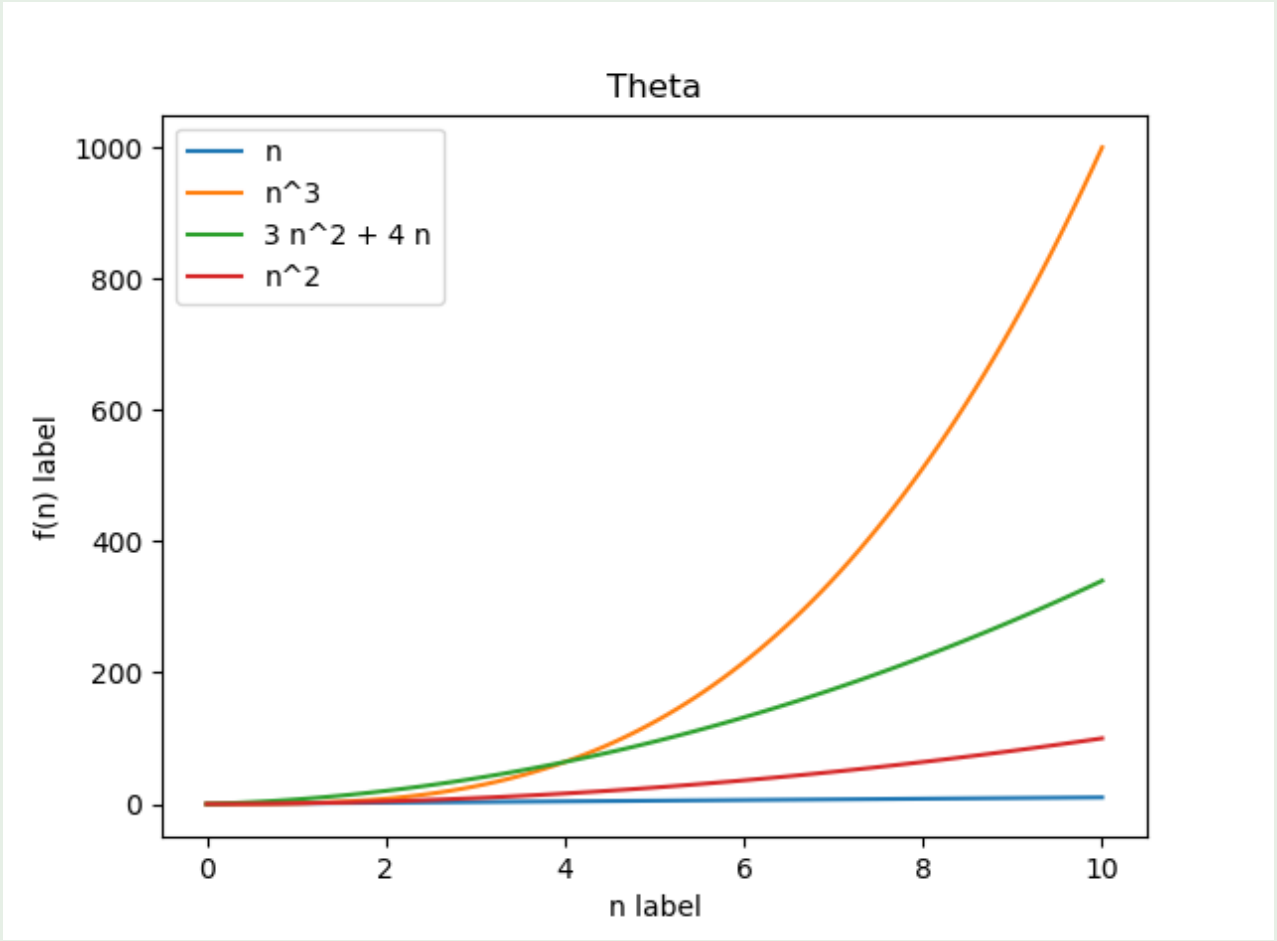
Remarque : abus de notation. En réalité, $f \in \Theta(g)$.

Exemple :

$f(n) = 5n^3 + 2n^2 + n + n^2 logn = \Theta(n^3)$

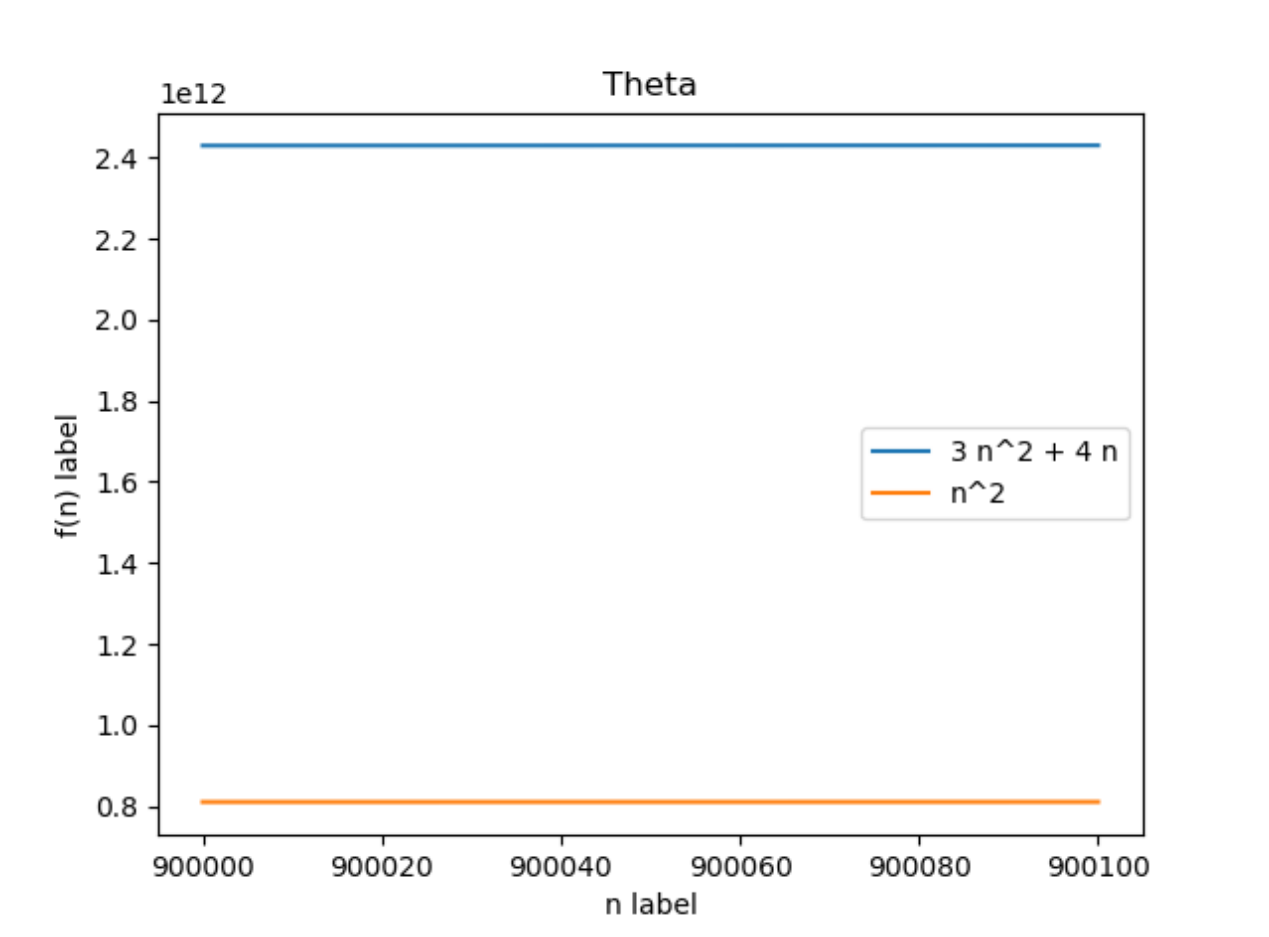
$f(n)$ est d'ordre n^3.

Exemple



On a : $3n^2 + 4n = \Theta(n^2)$, avec $3n^2 + 4n = O(n^3)$ et $3n^2 + 4n = \Omega(n)$

Exemple



Quand n est grand, $3n^2 + 4n$ et n^2 sont du même ordre à $C = 3$ près.

Propriétés

- $f = \Theta(g) \Leftrightarrow g = \Theta(f)$
- $O(cf) = O(f), \forall c \in \mathbb{R},$ de même pour Θ et Ω
- $O(f + g) = O(max(f, g)),$ de même pour Θ et Ω
- un polynôme de degré d est en $\Theta(n^d)$:

$$\sum_{i=0}^{d} a_{d-i} n^{d-i} = \Theta(n^d)$$

- un polynôme de degré d est en $O(n^{d+1})$:

$$\sum_{i=0}^{d} a_{d-i} n^{d-i} = O(n^{d+1})$$

- $log_a(n) = \Theta(log(n)), \forall a > 1$

Un algorithme A est un :

- $O(g)$: si A effectue au plus de l'ordre de g opérations, borne supérieure
- $\Omega(g)$: si A effectue au moins de l'ordre de g opérations, borne inférieure
- $\Theta(g)$: si A effectue exactement de l'ordre de g opérations

Complexités les plus courantes

- $O(1)$ algorithme constant (test de parité, ...)
- $O(loglog(n))$ algorithme log logarithmique
- $O(log(n))$ algorithme logarithmique (recherche binaire, ...)
- $O(log^k(n))$ algorithme polylogarithmique
- $O(n)$ algorithme linéaire (recherche simple, ...)
- $O(nlog(n))$ algorithme pseudo linéaire (tri fusion, quicksort, FFT, ...)
- $O(n^2)$ algorithme quadratique (tri par insertion, multiplication naïve, ...)
- $O(n^3)$ algorithme cubique (produit matrice vecteur, tous les plus courts chemins - Floyd-Warshall -, ...)
- ...

Complexités les plus courantes

$O(2^n)$ algorithme exponentiel (force brute sur un problème de décision - ex sac à dos - , ...)
Si la complexité s'exprime sous forme de factorielle (TSP - voyageur de commerce), alors on peut appliquer la formule suivante :

Formule de Stirling : $\lim_{n \to +\infty} \frac{n!}{\sqrt{2\pi n}(n/e)^n}$ ou $n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$

Exemple : calcul de 40! (avec bc sous bash) :

```
$  echo 'define fact(x) {if (x>1){return x*fact(x-1)};return 1} fact(40)' | bc
815915283247897734345611269596115894272000000000
```

Et de la formule de Stirling (sous R) :

```
>sqrt(2*40*pi)*(40/exp(1))^40
[1] 8.142173e+47

Integer.MAX_VALUE:          2147483647
Long.MAX_VALUE:    9223372036854775807
20 !            =  2432902008176640000
21 !            = 51090942171709440000
```

Factorielle ça croît comment ?

QUIZZ ...

Factorielle ça croît comment ?

Correction de l'affirmation faite en cours : si on suppose qu'un chiffre écrit en police Arial 9 a une taille moyenne de 1.724 mm, alors 393 500 ! qui s'écrit avec 2 030 720 chiffres (calcul donné par la fonction ci-dessous) représente une longueur de 3.5 km, soit la distance de l'IUT au terminal 1 de l'aéroport. Inutile de lancer le calcul dans une boucle `for()` !

```
1 static int findDigits(int n) {
2   if (n < 0) return 0;
3   if (n <= 1) return 1;
4   double digits = 0;
5   for (int i = 2; i <= n; i++)
6     digits += Math.log10(i);
7   return (int) (Math.floor(digits)) + 1;
8 }
```

Temps d'exécution avec un Intel Core i7 5960X (Haswell) 238 310 MIPS (Millions Instructions Par Seconde) à 3.0 GHz (Dhrystone)

n	$\Theta(n)$	$\Theta(log(n))$	$\Theta(nlog(n))$	$\Theta(n^2)$	$\Theta(2^n)$
5	2.1×10^{-11}s	9.7×10^{-12}s	4.9×10^{-11}s	1.0×10^{-10}s	1.3×10^{-10}s
10	4.2×10^{-11}s	1.4×10^{-11}s	1.4×10^{-10}s	4.2×10^{-10}s	4.3×10^{-9}s
1000	4.2×10^{-9}s	4.2×10^{-11}s	4.2×10^{-8}s	4.2×10^{-6}s	$4.5 \times 10^{+289}$s
10^6	4.2×10^{-6}s	8.4×10^{-11}s	8.4×10^{-5}s	4.2s	4.2×10^{301020}s , 1 siècle $\approx 3.2 \times 10^9$s

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

n	$log(n)$	$nlog(n)$	n^2	2^n
2.1×10^{16}	$2^{205899840000000000}$ ou $10^{0.6 \times 10^{16}}$	4.2×10^{14}	1.4×10^8	54

Complexité en fonction de quoi ?

De la taille des données auxquelles s'applique l'algorithme

- sur un entier : l'entier (ou la taille nécessaire pour le coder en binaire)
- sur une chaîne de caractères : longueur de la chaine
- 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 d'arêtes

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 $2 \times n$.

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/Python 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) $= min\{t(I), I \subset D_n\}$
- Complexité dans le pire des cas
  pire(n) $= \max\{t(I), I \subset D_n\}$
- Complexité en moyenne
  moyenne(n) $= \sum_{I \subseteq D_n} P(I) \times t(I)$ où $P(I)$ est la proba de I.

Complexité temporelle des algorithmes itératifs

- Les instructions élémentaires (affectation, comparaison) sont en temps constant.
- Séquence de blocs.

```
1        I1 ;
2        I2 ;
```

est de complexité $\Theta(max(f1(n), f2(n)))$ où $\Theta(f1(n))$ est la complexité de `I1` et $\Theta(f2(n))$ la complexité de `I2`.

- If then else

```
1          if (C) I1 ; else I2;
```

est de complexité $O(max(f(n), f1(n), f2(n)))$ où $\Theta(f(n))$ est la complexité de C, $\Theta(f1(n))$ la complexité de I1 et $\Theta(f2(n))$ la complexité de I2.

- Itération for

```
1          for (int i=0; i< n; i++)
2              {I;}
```

est de complexité $\Theta(n(f(n)))$
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 $\sum_{i=0}^{n-1} \Theta(f(i))$

- Itération while

```
1   While (C)
2     {I;}
```

est de complexité $\Theta(g(n) \times max(f1(n), f2(n)))$ si C est en $\Theta(f1(n))$, I en $\Theta(f2(n))$ et que la boucle while est exécutée $\Theta(g(n))$.

Exemple 1 : Recherche du maximum d'un tableau (Java)

```
1 public static int chercheMax(int[] tab) {
2   int maxCourant = tab[0];
3   for (int i=1;i<tab.length;i++)
4       if (maxCourant < tab[i])
5       maxCourant = tab[i];
6   return maxCourant;
7 }
```

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

Exemple 2 : Recherche d'un élément dans un tableau

```
1  public static int recherche(int[] tab, int x) {
2    for (int i=0;i<tab.length;i++)
3      if (x==tab[i]) return i;
4    return -1;
5  }
```

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

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.

On trouve :

$$\text{moyenne trouvée}(n) = \sum_I P(I)t(I) = \sum_{i=0}^{n-1} \frac{1}{n}(i+1)$$

On simplifie :

$$\begin{array}{rcl} \sum_{i=0}^{n-1} \frac{1}{n}(i+1) & = & \frac{1}{n}\sum_{i=0}^{n-1}(i+1) \\ & = & \frac{1}{n}\sum_{i=1}^{n} i \\ & = & \frac{1}{n}\frac{n(n+1)}{2} \end{array}$$

Complexité en moyenne quand x **n'est pas** dans le tableau : quelle que soit la donnée, il y a n comparaisons. Donc $\text{moyennepasTrouvé}(n) = n$.

La complexité en moyenne est :

$$\text{moyenne}(n) = p \times \text{moyennetrouvé}(n) + (1-p) \times \text{moyennepasTrouvé}(n)$$

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

$$\text{moyenne}(n) = p\frac{(n+1)}{2} + (1-p) \times n$$

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.

Algorithme *Trier*(n) pour un tableau de taille n

1) Chercher le min et l'extraire.
2) Trier le tableau restant de taille $n-1$

Analyse

- (1) va couter $\Theta(n)$ pour la recherche et $\Theta(1)$ pour l extraction.
- (2) va couter $T(n-1)$.

Donc nous avons

$$T(n) = T(n-1) + \Theta(n)$$

Soit $T(n) = T(n-k) + \Theta(n + (n-1) + ...(n-k+1))$ et donc $T(n) = \Theta(\sum_{i=0}^{i=n} i) = \Theta(n^2)$

Algorithme *Trier*(n) pour un tableau de taille n

0) Si tableau est petit on le trie en temps constant.
1) Scinder le tableau en 2 tableaux T_0, T_1 de taille $n/2$
2) Trier les tableaux T_1, T_2.
3) Fusionner les deux tableaux.

Analyse

- (1) va couter $\Theta(n)$ au plus *Theta*(n) une bonne implémentation peut fontionner en $\Theta(1)$.
- (2) va couter $2 \times T(n/2)$.
- (3) coûte $\Theta(n)$.

Donc nous avons

$$T(n) = 2T(n/2) + \Theta(n)$$

Soit $T(n) = 4T(n/4) + \Theta(n + 2\frac{n}{2}) = 8T(n/8) + \Theta(n + 2\frac{n}{2} + 4\frac{n}{4})$
Pour k récursions :

$$T(n) = \Theta(2^k T(n/2^k)) + \Theta(\sum_{i=0}^{i=k} 2^i n/2^i)$$

$$T(n) = \Theta(2^k T(n/2^k)) + \Theta(\sum_{i=0}^{i=k} n$$

$$T(n) = \Theta(2^k T(n/2^k)) + k\Theta(n)$$

So k et le nombre de recursions nous avons :

$$T(n) = \Theta(2^k T(n/2^k)) + k\Theta(n)$$

- La récursion se termine quand $n/2^k = \Theta(1)$ (pex $k \in [0, 2]$)
- le k final k_0 est tel que $2^{k_0} = \Theta(n), k_0 = \Theta(log2(n))$.
- on a aussi $n/2^{k_0} = \Theta(1)$
- $T(n/2^{k_0}) = \Theta(1)$ car on tire un tableau borné.

Donc :

$$T(n) = \Theta(2^{k_0} T(n/2^{k_0})) + k_0\Theta(n) = \Theta(n)\Theta(1) + k_0\Theta(n)$$

$$T(n) = \Theta(2^{k_0} T(n/2^{k_0})) + k_0\Theta(n) = \Theta(n)\Theta(1) + \log 2(n)\Theta(n)$$

Bibliography