Algorithmes et programmes

Année 2007–2008

L1, Université Bordeaux 1

23 janvier 2008

Modalités

- 15 cours :
  - 2 cours/semaine les 5 1ères semaines,
  - 1 cours/semaine les 5 dernières semaines.
- 15 TD :
  - 1 TD/semaine les 5 1ères semaines,
  - 2 TD/semaine les 5 dernières semaines.
- 10 TP.
- Contrôle continu 50% DS + 50% projet ou test.
- Note finale : 75% Examen + 25% Max(Examen,CC).

Bibliographie

A. Aho, J.E. Hopcroft, J. D. Ullman.
Structures de données et algorithmes.
InterÉditions, 1989.

D. Beauquier, J. Berstel, Ph. Chrétienne.
Éléments d'algorithmique.
Masson, 1992.
Épuisé, mais disponible en bibliothèque. Version électronique (gratuite) :
http://www-igm.univ-mlv.fr/~berstel/Elements/Elements.html

Documentation python
http://www.python.org/doc/tut/

Plan du cours

1. Algorithmes et programmes : généralités
2. Rappels de programmation

1 – Algorithmes et programmes : généralités

Généralités sur les algorithmes

Mohamed *Al-Khwarizmi*, Perse, vers 780-850

- Un algorithme est une méthode systématique, une recette, pour résoudre un problème donné.
- Il se compose d'un suite d'opérations simples à effectuer pour résoudre le problème. Cette méthode peut donc être appliquée par une machine, un ordinateur.
- L'algorithme doit donner une réponse après un nombre fini d'opérations.

Exemple d'algorithmes

Problème : Étant donné un entier $x > 2$, est-ce que x est premier ?

Algorithme `Prem1` :

1. si x est pair, alors répondre NON.
2. tester, pour tous les entiers impairs $i \in [3, \sqrt{x}]$, si i ne divise pas x.
3. si oui, répondre OUI, sinon répondre NON.

- On exprime un algorithme en langage naturel. L'objectif est que l'algorithme soit compréhensible par un humain, pour l'appliquer ou le prouver par exemple.
- La suite des opérations doit être suffisamment claire et explicite pour que la méthode soit systématique.

Programme

- L'ordinateur ne peut exécuter qu'un programme. Il ne comprend pas un algorithme.
- Un programme est un texte écrit dans un langage obéissant à des règles très précises, traduisant les opérations simples de l'algorithme en instructions.
- Il existe de nombreux langages : C, C++, Java, Lisp, Caml, Prolog, Smart Talk, ...
- Dans cette UE on utilisera `Python`.

Programme Python correspondant à l'algorithme Prem1

```
from math import *
def est_premier1(x):
    if x%2==0:
        return "NON"
    else:
        i=3
        divise=False
        while i<=sqrt(x):
            if x%i==0:
                divise=True
            i=i+2
        if divise:
            return "NON"
        else:
            return "OUI"
```


- Il y a plusieurs programmes pour le même algorithme.
- Les opérations dictées par l'algorithme sont les mêmes, leur traduction dans le langage est différente.
- On parle d'implantation d'un algorithme.

```
def est_premier2(x):
    i=2
    divise=False
    while i*i<=x:
        if x%i==0:
            divise=True
        i=i+2
        if i==4:
        i=3
    if divise:
        return "NON"
    else:
        return "OUI"
```

Nous reviendrons plus tard sur Python.

Algorithmes et programmes
└Algorithmes et programmes : généralités

Qu'est-ce qu'un problème ?

C'est la description d'une entrée et d'une sortie. On utilise parfois les termes *donnée* (ou *instance*), et *résultat*.

PREMIER :

Entrée : un entier $x > 2$

Sortie : OUI, si x est premier, NON sinon

CONNEXE :

Entrée : un graphe G

Sortie : OUI, si G est connexe, NON sinon

Algorithmes et programmes
└Algorithmes et programmes : généralités

Exemples de problèmes

VALEUR_POLYNÔME

Entrée : un polynôme $P(X) = a_0 + a_1X + \cdots + a_pX^p$ donné par ses coefficients entiers $a_0, \ldots, a_p$, et une valeur entière x_0.

Sortie : la valeur $P(x_0)$.

PGCD

Entrée : deux entiers positifs x et y.

Sortie : la valeur du plus grand diviseur commun de x et y.

COLORATION

Entrée : un graphe fini.

Sortie : une coloration du graphe avec le minimum de couleurs.

Exemples de problèmes

POSITION_DANS_PERMUTATION

Entrée : une permutation σ des entiers entre 1 et n.

Sortie : la position de l'entier 1.

INTERSECTION

Entrée : deux suites finies d'entiers.

Sortie : OUI si les deux suites ont un élément commun, NON sinon.

INTERSECTION_BIS

Entrée : deux suites finies d'entiers compris entre 0 et 20.

Sortie : OUI si les deux suites ont un élément commun, NON sinon.

- On a vu plusieurs programmes implémentant un même algorithme.
- Il y a aussi plusieurs algorithmes pour un même problème.

Algorithme `Prem1bis` :

1. pour tous les entiers $i \in [2, \sqrt{x}]$
2. si i divise x, alors répondre NON.
3. si tous les tests précédents ont échoué, répondre OUI.

```
i=2
while i*i<=x:
    if x%i==0:
        return "NON"
    i=i+1
return "OUI"
```


Des questions naturelles

- Quelles sont les propriétés souhaitées pour un algorithme ?
  - correct, et
  - efficace.
- Comment s'assurer qu'un algorithme est correct ?
  On verra que c'est une question difficile.
- Lorsqu'on a plusieurs algorithmes pour résoudre un problème, comment choisir entre l'un ou l'autre ?
  On peut comparer l'efficacité des algorithmes sur différentes instances de grande taille.
- Comment évaluer l'efficacité d'un algorithme ?

Représentation des données

Pour évaluer l'efficacité d'un algorithme, on a besoin de spécifier la taille des entrées et des sorties, et donc leur représentation.

ADD :

Entrée : deux entiers naturels x et y

Sortie : l'entier $x + y$

Un algorithme qui résoud le problème ADD sera différent si x et y sont écrits en unaire, en binaire, en écriture décimale ou en chiffres romains !

Par défaut, il s'agira toujours d'une représentation en binaire, celle utilisée par l'ordinateur.

Taille des entrées

ADD2 :

Entrée : deux entiers x et y écrit en binaire

Sortie : l'entier $x + y$ écrit en binaire

La taille d'une entrée pour un problème donné est la longueur de sa représentation. [unité=*bit*=*binary digit*, poids des fichiers de données].

Pour ADD2 c'est le nombre total de chiffres binaires pour écrire x et y.

La taille est généralement exprimée en bits, mais pas toujours ! Parfois c'est un nombre de sommets ou d'arêtes par exemple si l'on suppose que les opérations sur les sommets ou les arêtes sont élémentaires.

Base 2

- Les entiers sont représentés par une suite de chiffres (digit).
- Par exemple en base 10 : $180 = 1 \times 10^2 + 8 \times 10^1 + 0 \times 10^0$.
- En base 2, on utilise le même principe.
  - $5 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$ est représenté par 101.
  - $180 = 2^7 + 2^5 + 2^4 + 2^2$ est représenté par 10110100.

180 :

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	0	1	1	0	1	0	0

- En base 1 : 180 représenté par $\underbrace{111\cdots111}_{\text{180 fois}}$.
- La taille d'un entier est le nombre de chiffres qui le compose.

Comparaisons des bases

- En informatique, on utilise souvent la représentation en base 2 : c'est la façon dont les entiers sont représentés en machine.
- Un entier $x > 0$ est représenté
  - en base 2 avec $\lceil \log_2(x+1) \rceil$ bits.
  - en base 10 avec $\lceil \log_{10}(x+1) \rceil$ bits.
- Ces deux quantités sont proportionnelles.
- En base 1 par contre, il faut x chiffres, ce qui est exponentiellement plus élevé.

Qu'est-ce qu'une opération simple (1/2) ?

- Malheureusement, dans un algorithme, cela dépend du niveau d'abstraction auquel on se situe, et des objets que l'on manipule.
- Parfois il s'agit d'opérations arithmétiques $(+, -, *, /, \%)$ sur des entiers, d'opérations ensemblistes $(\cup, \cap, \setminus)$, d'opérations sur des graphes...
- Par contre, quand on s'intéresse à l'algorithme d'addition de 2 entiers, (opération pourtant plus simple que la division), on ne considère pas que l'addition est « simple » : on se place à un niveau plus réaliste.

Algorithme PGCD1 et PGCD2 (d'Euclide)

Algorithme PGCD1(x, y)

1. pour tout entier d allant de $\min(x, y)$ à 1 par pas de -1 :
2. si d divise x et si d divise y
3. répondre d

Algorithme PGCD2(x, y) (d'Euclide)

1. $r \leftarrow$ reste de la division de y par x
2. Tant que $r \neq 0$, faire :
3. $y \leftarrow x$
4. $x \leftarrow r$
5. $r \leftarrow$ reste de la division de y par x
6. retourner x.

Qu'est-ce qu'une opération simple (2/2) ?

- Algorithme `Est-Connexe`(G) :
  - Tester s'il existe une chaîne entre toute paire de sommets de G
  - Si oui, répondre OUI, sinon NON
- Ici l'opération simple sous-entendue est une fonction `chaîne(G,s,t)` qui répond `True` s'il existe une chaîne entre s et t dans G, et `False` sinon.
- Dans tous les cas il doit s'agir d'opérations que l'on peut décomposer en une suite d'opérations **élémentaires**, "comprises" et exécutables par l'ordinateur.
- Dans l'exemple précédent, `chaîne(G,s,t)` peut utiliser une fonction `est_voisin(G,s,t)`.

Instructions élémentaires

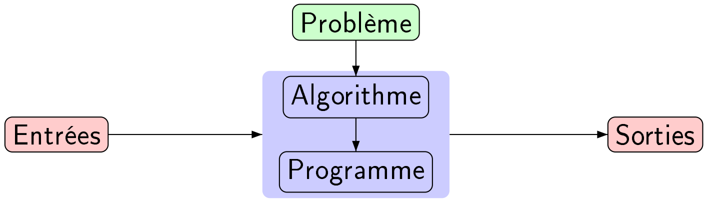
L'ordinateur ne peut exécuter que des instructions dites élémentaires, qui prennent un temps constant, indépendant des opérandes.

Ex : addition, multiplication ou comparaison ($<, >, ==$) de variables de type `int`.

Remarques

- Une variable de type `int` dans un ordinateur ne peut pas contenir n'importe quel entier. Seulement ceux représentables en binaire avec 32 ou 64 bits (dépend de l'architecture du processeur). Cela ne fait qu'au plus 2^{64} entiers possibles, ce qui fait 18 milliards de milliards (20 chiffres). C'est déjà pas mal.
- L'addition (type `int`) et peut être légèrement plus rapide que la multiplication, mais dans tous les cas cela prend un temps constant, indépendant des opérandes.
- `Python` possède un type d'entiers long (type `long`) qui permet de manipuler des entiers très long, de longueur (presque) arbitraire. Dans ce cas, l'addition et la multiplication sur ce type de variable ne peuvent plus être considérées comme instructions élémentaires.

Résumé



Taille des entrées

Opération simple
Instruction élémentaire

Notion de complexité

- **Rappel.** Il existe plusieurs algorithmes pour résoudre un même problème.
- Pour le problème PREMIER par exemple, encore un algorithme :
  Algorithme `Prem2` :
  - tester si x est pair
  - tester si tous les entiers **premiers** $\leqslant \sqrt{x}$ ne divisent pas x.
- Certains sont plus faciles que d'autres à programmer ...
- Certains sont plus efficaces que d'autres ...

Complexité : définition

- La complexité d'un algorithme A résolvant le problème P est le nombre maximum $C(n)$ d'instructions élémentaires utilisées par A pour résoudre P sur une entrée de taille n.

$$C(n) = \max_{|x|=n} c(x), \quad \text{où } |x| \text{ est la taille de } x$$

- On peut définir de manière similaire la complexité d'un programme.
- On parle aussi de complexité dans le **pire des cas**.
- $C(n)$ est donc une fonction de n.
- Souvent on s'intéresse à $C(n)$ lorsque n est "grand". On parle de complexité asymptotique.

Complexité : exemple

- Algorithme Trouver_1
  pour POSITION_DANS_PERMUTATION :
  - Pour chaque i entre 1 et n.
  - si $\sigma(i)$ vaut 1, retourner i.
- On compte le nombre de comparaisons de $\sigma(i)$ à 1.
- On considère que la taille de la permutation est n (ce qui n'est pas tout à fait exact).
- Complexité dans le cas le pire : n comparaisons.
- On peut aussi définir la complexité dans le meilleur cas, ici 1. Elle est peu utilisée en raison de son manque d'intérêt.
- On peut aussi définir la complexité en moyenne, en se donnant une distribution sur les permutations.

Recherche de la valeur 1 dans une permutation : complexité en moyenne

- On suppose une distribution de probabilité uniforme sur l'ensemble $\mathfrak{S}_n$ des permutations de $\{1, \ldots, n\}$.
- Comme il y a $n!$ permutations, le coût est

$$\frac{1}{n!} \sum_{\sigma \in \mathfrak{S}_n} c(\sigma)$$

- Le coût pour σ est $c(\sigma) = k$ si 1 apparaît en position k dans σ.
- Il y a $(n-1)!$ permutations telles que $\sigma(k) = 1$.
- La complexité en moyenne Trouver_1 est donc

$$\frac{1}{n!} \sum_{k=1}^{n} k \times (n-1)! = \frac{1}{n} \sum_{k=1}^{n} k = \frac{n+1}{2}.$$

Incrémentation en binaire

INCRÉMENTER

Entrée : un entier $x = x_{n-1} \cdots x_0$, de n bits (x_i est soit 0 soit 1).

Sortie : l'entier $x+1$ si $x < 2^n - 1$, l'entier 0 si $x = 2^n - 1$.

Algorithme Incrémenter

- $k \leftarrow 0$
- Tant que $x_k = 1$ et $k \neq n$ faire
-     $x_k \leftarrow 0$
-     $k \leftarrow k+1$
- Si $k \neq n$, alors faire $x_k \leftarrow 1$

Complexité de l'incrémentation (1/3)

- On compte le nombre $c(x)$ d'affectations à un x_k.
- Dans le cas le pire, on trouve n, si $x = x_{n-1}\underbrace{111\cdots111}_{n-1 \text{ fois}}$.

  La retenue va propager le calcul jusqu'à x_{n-1}
- Que vaut $c(x)$ en moyenne, en supposant les entrées équiprobables ?
- On remarque que l'écriture en base 2 d'un nombre pair finit par 0.
- Donc, quand x représente un nombre pair, le coût $c(x)$ vaut 1 (seul le dernier 0 est changé en 1 et l'algorithme s'arrête).

Complexité de l'incrémentation (2/3)

- Soit $E(n)$ l'ensemble des entrées de taille n.
- Par exemple $E(2) = \{00, 01, 10, 11\}$.
- On note $|E(n)|$ le nombre d'éléments de $E(n)$. On a $|E(n)| = 2^n$.
- Par définition, le nombre moyen d'affectations à x_k pour $x \in E(n)$ est

$$C_{\text{moy}}(n) = \frac{1}{|E(n)|} \sum_{x \in E(n)} c(x).$$

- On compte le nombre d'entrées pour lesquelles $c(x) = k$.
- $c(x) = k$ pour les entrées de la forme $x = x_{n-1} \cdots x_k 0\underbrace{111\cdots111}_{k-1 \text{ fois}}$.

  (sauf pour $k = n$, ou cela arrive pour 2 entrées).

Complexité de l'incrémentation (3/3)

- Pour $1 \leqslant k < n$, il y a 2^{n-k} entrées x telles que $c(x) = k$.
  ... sauf pour $k = n$ où il y a 2 telles entrées. Donc

$$
\begin{aligned}
C_{\text{moy}}(n) &= \frac{1}{|E(n)|} \sum_{x \in E(n)} c(x) \\
&= \frac{1}{2^n} \sum_{x \in E(n)} c(x) \\
&= \frac{1}{2^n} \left[2n + \sum_{k=1}^{n-1} k \cdot 2^{n-k} \right]
\end{aligned}
$$

- On calcule facilement $\left[2n + \sum_{k=1}^{n-1} k \cdot 2^{n-k}\right] = 2^{n+1} - 2$.
- On a donc

$$
C_{\text{moy}}(n) = \frac{2^{n+1} - 2}{2^n} = 2 - \frac{1}{2^{n-1}}
$$

- La complexité en moyenne de INCRÉMENTATION est inférieure à 2 !

Complexité d'un problème

- Dans certains cas, on parle de complexité d'un problème P pour parler de la plus petite complexité possible d'un algorithme résolvant P.
- Déterminer la complexité d'un problème, ou même son ordre de grandeur, est en général très difficile : il faut trouver le meilleur algorithme possible !

Complexité inconnue (1/3)

- La multiplication de deux entiers de n bits ?
- L'algorithme que l'on apprend à l'école a une complexité proportionnelle à n^2.
- Le meilleur algorithme date de 2007, et a une complexité de

$$
(n \log n) \cdot 2^{\log^* n}
$$

  où $\log^* n$ est le nombre de fois que l'on doit successivement appliquer la fonction $\log_2$ pour obtenir un nombre $\leqslant 1$. Si

$$
n = 2^{2^{2^{\cdot^{\cdot^{\cdot^{2^2}}}}}}
$$

  alors $\log^* n$ vaut le nombre de "2". Cette fonction a une croissance très très faible.

Complexité inconnue (2/3)

- Peut-on colorier les sommets d'un graphe G en 3 couleurs ? (sans avoir deux voisins adjacents de la même couleurs)
- Cette question fait partie des 7 questions mathématiques sélectionnées par le CLAY Mathematical Institute pour leur difficulté.
- Elle vaut 1 million \$.

Complexité inconnue (3/3)

- Est-ce qu'un graphe G possède un triangle ?
  (3 sommets x, y, z deux à deux voisins)

Le meilleur algorithme a une complexité proportionnelle à $n^{2,376}$ où n est le nombre de sommets de G, l'algorithme naïf ayant une complexité proportionnelle à n^3 :

Algorithme `Triangle-Naïf`(G) :

1. tester pour tous les triplets $\{x, y, z\}$ de sommets si x, y, z forment un triangle
2. si oui, OUI, sinon NON

Complexité : linéaire, exponentielle, *etc.*

- L'algorithme A a une complexité *linéaire* si, lorsque n est grand, $C(n)$ est proportionnelle à n. Mathématiquement, il faut qu'il existe une constante $a \neq 0$ telle que :

$$\limsup_{n\to+\infty} C(n)/n \ = \ a$$

- Terminologie (a, b sont des constantes, $a > 0$ et $b > 1$) : Pour une fonction $f(n)$ telle que $\limsup_{n\to+\infty} C(n)/f(n) = a$:

$f(n)$	type de complexité
n	linéaire
n^2	quadratique
b^n	exponentielle
$\log_b n$	logarithmique
n^b	polynomiale

- Les fonctions telles que $\log_b n$ ou $\sqrt[b]{n}$ sont **sous-linéaires** en n.

Temps d'exécution

Comme une instruction élémentaire prend un temps constant (disons a secondes), indépendant des opérandes et donc de la taille des entrées, la complexité peut donner une idée du temps d'exécution du programme T qui s'exécuterait sur une entrée de taille n.

$$\text{temps}(T, n) \ \leqslant \ a \cdot C(n)$$

Complexité de `Prem1`

La complexité de `Prem1` est proportionnelle à $\sqrt{x}$, qui est sous-linéaire à x.

1. si x est pair, alors répondre NON.
2. tester, pour tous les entiers impairs $i \in [3, \sqrt{x}]$, si i ne divise pas x.
3. si oui, répondre OUI, sinon répondre NON.

La complexité de `Prem1` n'est pas sous-linéaire ! car la taille de x n'est pas x mais $n = \lceil \log_2(x+1) \rceil$, qui représente la longueur de l'écriture binaire d'un entier $x > 0$.

$$\sqrt{x} = 2^{\log_2 \sqrt{x}} = 2^{\frac{1}{2}\log_2 x} = 2^{n/2}$$

(en négligeant la partie entière supérieure $\lceil \cdot \rceil$ et le $+1$)

La complexité de `Prem1` est donc **exponentielle !**

Complexité de PREMIER

En 2004, il a été trouvé un autre algorithme résolvant PREMIER de complexité $a \cdot (\log_2 x)^6$, soit $a \cdot n^6$ pour un nombre de n bits. La complexité de PREMIER est donc **polynomiale**.

Différence entre $2^{n/2}$ et n^6

La différence est énorme ! Si l'on suppose qu'un ordinateur peut exécuter une instruction élémentaire en 10^{-9} s (1 GHz), soit 1 milliard d'instructions par seconde, alors :

n (bits)	temps pour $2^{n/2}$		temps pour n^6	
32	$2^{32/2} \times 10^{-9} \approx$	65 μs	$32^6 \times 10^{-9} \approx$	1 s
64	$2^{64/2} \times 10^{-9} \approx$	4 s	$64^6 \times 10^{-9} \approx$	1 min 8 s
128	$2^{128/2} \times 10^{-9} \approx$	**584 ans**	$128^6 \times 10^{-9} \approx$	**1 h 13**
256	$2^{256/2} \times 10^{-9} \approx$	10^{11} ma.	$256^6 \times 10^{-9} \approx$	3,5 j
512	$2^{512/2} \times 10^{-9} \approx$	10^{52} ma.	$512^6 \times 10^{-9} \approx$	6,8 mois
1024	$2^{1024/2} \times 10^{-9} \approx$	10^{128} ma.	$1024^6 \times 10^{-9} \approx$	36 ans

1 μs = 1 millionième de seconde
1 ma. = 1 milliard d'années
512 bits = 320 chiffres décimals

- Dans la pratique, on cherche bien sûr des algorithmes de complexité la plus faible possible.
- Les complexités exponentielles sont donc à éviter.
- Parfois, ce n'est pas possible ! Voici un problème de complexité doublement exponentielle, où tout algorithme a une complexité $C(n) \geqslant 10^{10^n}$.
  - Entrée : un entier x (écrit en base 10).
  - Sortie : tous les entiers de x chiffres.
- y a 10^x entiers de x chiffres, donc la complexité est au moins 10^x. Or, la taille de l'entrée est ici de $n \approx \log_{10} x$. Autrement dit $x \approx 10^n$.
- La complexité de ce problème est au moins $10^x = 10^{10^n}$.

Problèmes très très difficiles

- Contrairement à ce qu'on pourrait croire, il existe des problèmes qui ne possèdent pas d'algorithme pour les résoudre, quelque soit leur complexité !
- On parle de problème **indécidable**.

Exemple de problèmes indécidables (1/2)

HALTE :

Entrée : un programme `Python` (fichier de caractères)

Sortie : OUI si le programme s'arrête au bout d'un moment, NON sinon (s'il boucle)

Notez qu'à chaque programme `Python`, une réponse existe belle et bien : c'est OUI ou NON. Il n'y a simplement pas de méthode systématique pour **calculer** la réponse. Une machine ne pourra jamais résoudre ce problème.

Exemple de problèmes indécidables (2/2)

DIOPHANTIEN [Hilbert, 1900] :

Entrée : une équation polynomiale à coefficients entiers.

Sortie : OUI s'il existe une solution en entiers > 0, NON sinon.

Ex :

$x^2 + y^2 = z^2$ (OUI, $3^2 + 4^2 = 5^2$)

$x^5 + y^5 = z^5$ (NON, Théorème de Fermat-Wiles)

$x^2y + 2xyw - y^3w^4 = 3z^5$ (? ? ?)

Théorème de Matiyasevich (1970)

Il n'existe aucun algorithme répondant au problème DIOPHANTIEN.

Description des algorithmes

- Un algorithme peut être décrit en utilisant
  - une séquence d'instructions comprenant :
  - l'affectation de valeurs à des variables : $x \leftarrow$ valeur,
  - des instructions conditionnelles si C alors S,
  - des instructions conditionnelles si C alors S sinon T,
  - des instructions répétitives tant que C faire S,
  - des instructions répétitives pour tout élément x de L faire S.
  - Une instruction de retour.

Notation grand $O(\cdot)$

Pour simplifier l'expression de certaines complexités, on utilise la notation $O(f(n))$ où $f(n)$ est une fonction de n.

Par exemple si la complexité de l'algorithme `A` est $C_A(n) = 3n - 2$ et celle de l'algorithme `B` est $C_B(n) = \frac{1}{2}n^2 + \sqrt{n} - 1$, alors on notera $C_A(n) = O(n)$ et $C_B(n) = O(n^2)$.

C'est une façon de dire que la complexité de `A` est linéaire et celle de `B` quadratique.

Dans la notation $O(\cdot)$ on ne retient que le terme dominant (lorsque $n \to +\infty$) et on supprime le facteur multiplicatif. On parle d'ordre de grandeur et de complexité asymptotique.

L'idée est qu'il ne sert à rien de calculer la complexité d'un algorithme à l'unité près. Par exemple, sur une machine à 1 GHz et pour $n = 10^9$, l'exécution de n ou $n+1$ instructions élémentaires prendrons 1 s dans les deux cas, la différence étant en pratique indiscernable !

On supprime le facteur multiplicatif, car il est courant qu'une suite de plusieurs instructions élémentaires sur une machine A puissent être regroupées en une seule sur une machine B plus puissante.

Ex : calcul de "A=B+4"

```
Z80:   LD A,4      ; charger 4 dans l'accumulateur
       ADD B       ; A=A+B

i86:   ADD A,B,4   ; A=B+4
```

Sur les machines récentes (processeurs Intel DuoCore), on excécute plusieurs instructions élémentaires en parallèle ... Cela fait donc n ou $\frac{1}{2}n$ instructions élémentaires suivant la machine.

La complexité d'un algorithme est une mesure inhérente à l'algorithme (la méthode), elle ne doit pas dépendre de la machine sur laquelle on l'implante.

Définition formelle

Soient C et f deux fonctions $\mathbb{N} \to \mathbb{R}$.

On note "$C(n) = O(f(n))$", et l'on dit "$C(n)$ est en grand-O de n", pour dire qu'ils existent $n_0 \in \mathbb{N}$ et $a \in \mathbb{R}^*$ tels que

$$\forall\, n \geqslant n_0, \quad C(n) \ \leqslant\ a \cdot f(n) \ .$$

Attention !

$C(n) = O(f(n))$ est une **notation** !

Si $C_1(n) = O(n)$ et $C_2(n) = O(n)$, alors il est faux de dire que $C_1(n) = C_2(n)$ ou encore que $C_1(n) - C_2(n) = 0$!
En effet, peut-être que $C_1(n) = 3n$ et $C_2(n) = n + 2\sqrt{n}$.

Mais il est correcte d'écrire : $C_1(n) + C_2(n) = O(n)$.

En effet, $C_1(n) + C_2(n) = 4n + 2\sqrt{n} \leqslant 5n$ pour tout $n \geqslant 4$ (car $2\sqrt{n} \leqslant n \Leftrightarrow 2 \leqslant \sqrt{n} \Leftrightarrow 4 \leqslant n$).
Ici $n_0 = 4$ et $a = 5$.

Complexité constante

La notation $C(n) = O(1)$ signifie (par définition) qu'ils existent $n_0 \in \mathbb{N}$ et une constante $a \in \mathbb{R}$ telles que $C(n) \leqslant a$ $(= a \cdot 1)$ pour tout $n \geqslant n_0$.

On dit que la fonction C est bornée par une constante, sous-entendu lorsque n est assez grand.

Ex : $10/n + 2(\sin n)^2 = O(1)$ [Pourquoi ?].

Dans certains ouvrages on note (plus justement ?) "$C(n) \in O(f(n))$" au lieu de $C(n) = O(f(n))$.

Parfois, dans des calculs de complexité, on écrit (à lire de gauche à droite) :

$$C(n) = 2n - 4\sqrt{n} + 1 = 2n - O(\sqrt{n}) = O(n)$$

ce qui pose des problèmes avec la notation $\in O(\cdot)$.

Calcul précis de complexité

On utilise aussi la notation $O(\cdot)$ car dans la pratique il est souvent difficile de calculer exactement la complexité d'un algorithme (ou d'un programme), c'est-à-dire le nombre d'instructions élémentaires.

```
for i in range(n):
    s=s+i
```

Combien d'instructions élémentaires ?
n ? $5n - 1$?
$\Rightarrow$ La complexité de cette boucle est $O(n)$.

Retenons qu'une instruction élémentaire est une instruction qui s'exécute en un temps (constant) indépendant de la taille de ses opérandes. Donc `s=s+i` peut être considérée comme une instruction élémentaire.

2 – Rappels de programmation

1 - Le langage `Python`
2 - Variables
3 - Itérations, boucles et tests
4 - Fonctions