Informatique Pour Tous

Cours 20 – La complexité et un peu d'informatique théorique

Lorsque nous écrivons un algorithme utilisant une boucle **WHILE**, il convient de vérifier :

- Que l'algorithme se termine (pas de boucle infinie)
- Qu'il renvoie la valeur souhaitée (algorithme valide)
- Et qu'il se termine en un temps convenable, sans consommer trop de ressource (complexité)

A / *Validité* des algorithmes

A.1 / *VARIANT* de boucle : vérification de la *terminaison*

En CPGE, nous utilisons essentiellement des algorithmes **itératifs** (boucles `WHILE` et `FOR`). Par sa définition, la boucle `FOR` est toujours exécutée un nombre fini de fois, mais pas la boucle `WHILE` qui est exécutée tant qu'une condition est valide. Si la condition de sortie de boucle n'est jamais vraie, la boucle est infinie.

Pour qu'un algorithme avec une boucle `WHILE` soit valide, il faut donc vérifier sa terminaison.

Vérifier la terminaison revient à s'assurer que la condition de sortie de boucle est toujours atteinte, après un certain nombre d'itérations.
Plus rigoureusement, pour prouver qu'un programme s'arrête, il faut vérifier que la suite formée par les valeurs des variables au cours des itérations converge en un nombre fini d'étapes vers une valeur satisfaisant la condition d'arrêt.
Regardons cela sur un exemple.

```
def puissance(x,n):
    r = 1
    compteur = 0
    while compteur < n :
        r = r * x
        compteur = compteur + 1
    return r
```

```
def puissance(x,n):
    r = 1
    decompteur = n
    while decompteur > 0 :
        r = r * x
        decompteur = decompteur - 1
    return r
```

Ces 2 algorithmes sont strictement équivalents

On appelle **variant de boucle** une variable, généralement entière et positive, faisant office de compteur dans la boucle : il est modifié à chaque itération de la boucle (ici la variable `compteur`).

- si la valeur du **variant** augmente (strictement) à chaque itération, la **terminaison** sera possible si la condition de sortie de boucle est une valeur maximale de l'invariant
- si la valeur du **variant** diminue (strictement) à chaque itération, elle sera possible si la condition de sortie est une valeur minimale du variant, généralement 0.

A.2 / *INVARIANT* de boucle : vérification de la *validité*

Même si la terminaison de l'algorithme est correcte, il reste à vérifier sa **validité,** c'est-à-dire démontrer que le résultat qu'il affiche correspond à ce qu'on souhaite lui faire calculer. On utilisera pour cela un « **invariant de boucle** ».
Un **invariant d'une boucle** est une propriété mathématique P_j qui est vérifiée à chaque exécution (itération j) du corps de cette boucle.
Pour prouver qu'une propriété est un invariant de boucle, nous utilisons un raisonnement par **récurrence** :

- ***Initialisation :*** Montrer que la propriété est vérifiée avant le premier passage dans le corps de la boucle.
- ***Transmission :*** Montrer que si la propriété est vérifiée avant une exécution quelconque du corps de la boucle, il est encore vérifié avant l'exécution suivante.
- ***Sortie :*** Après la dernière itération, l'invariant doit permettre de démontrer que le résultat obtenu lorsque la condition de sortie de boucle est atteinte, est le résultat qui était attendu.

Remarque: à l'intérieur de la boucle WHILE, nous noterons dans ce cours, pour une variable x : x_{j-1} sa valeur à la précédente itération de la boucle, et x_j sa nouvelle valeur à la fin de la $j^{ième}$ itération.

Exemple : si dans une boucle WHILE, on a l'affectation $x = x + 1$

- A l'itération n : on commentera $x_j \leftarrow x_{j-1} + 1$ avec x_{j-1} valeur de la variable à l'issue de la boucle $j-1$

  x_j valeur de la variable à l'issue de cette boucle j

  À la fin de la boucle j, on écrase x_{j-1} (ancienne valeur) qui prend la nouvelle valeur x_j

- A l'itération suivante $j+1$: on commentera $x_{j+1} \leftarrow x_j + 1$ avec x_j valeur de la variable à l'issue de la boucle j

  x_{j+1} valeur de la variable à l'issue de cette boucle $j+1$

Prenons **deux exemples** de calcul de puissance (méthodes naïve et rapide).

```
1  def puissance(k,n):
2      c=n
3      p=1
4      while c>0:
5          p=k*p
6          c=c-1
7      return p
```

Fonction calculant k^n avec n entier > 0

c sera le variant de boucle

Supposons l'invariant de boucle suivant : $p_j = k^{n-c_j}$ et $c \geq 0$

Initialisation : $p_0 = 1 = k^0 = k^{n-n} = k^{n-c_0}$ et $c_0 = n > 0 \rightarrow$ OK

Transmission : à l'issue de la boucle précédente, on avait $p_{j-1} = k^{n-c_{j-1}}$ et $c_{j-1} \geq 0$ or $c_{j-1} > 0$ (car entrée dans la boucle)

$p_j = k \times p_{j-1} = k \times k^{n-c_{j-1}} = k^{n-c_{j-1}+1} = k^{n-(c_{j-1}-1)} = k^{n-c_j}$

Et $c_j = c_{j-1} - 1 \geq 0$ (car $c_{j-1} > 0$ et c_j entier, donc $c_{j-1} \geq 1$) donc → OK

Sortie : sortie de boucle ssi $c_j \leq 0$, or $c_j \geq 0$ (invariant), donc $c_j = 0$,

d'où $p_j = k^{n-0} = k^n \rightarrow$ OK, c'est bien ce qui est renvoyé

```
1  def puissance_rapide(k,n):
2      c=n
3      p=1
4      i=k
5      while c>0:
6          if c%2==0:
7              i=i**2
8              c=c//2
9          else:
10             p=p*i
11             c=c-1
12     return p
13
14
```

Supposons l'invariant de boucle : $p_j \times i_j^{c_j} = k^n$ et $c_j \geq 0$

Initialisation : $p_0 \times i_0^{c_0} = 1 \times k^n$ et $c_j = n > 0 \rightarrow$ OK

Si on entre dans la boucle, c'est que $c_{j-1} > 0$. Supposons $p_{j-1} \times i_{j-1}^{c_{j-1}} = k^n$

Transmission : cas 1 : c_{j-1} est pair

$p_j \times i_j^{c_j} = p_{j-1} \times \left(i_{j-1}^2\right)^{c_{j-1}/2} = p_{j-1} \times i_{j-1}^{c_{j-1}} = k^n$, et $c_j = c_{j-1}/2 > 0 \rightarrow$ OK

cas 2 : c_{j-1} impair (donc ≥ 1)

$p_j \times i_j^{c_j} = p_{j-1} \times i_{j-1} \times i_{j-1}^{c_{j-1}-1} = p_{j-1} \times i_{j-1}^{c_{j-1}} = k^n$,

et $c_j = c_{j-1} - 1 \geq 0$ car c_{j-1} (donc ≥ 1) → OK

Sortie : sortie boucle ssi $c_j \leq 0$, or $c_j \geq 0$ (invariant) donc $c_j = 0$

D'où $p_j \times i^{c_j} = p_j \times i^0 = p_j = k^n \rightarrow$ OK, c'est bien ce qui est renvoyé.

On donne ci-dessous une fonction calculant la somme $f(n) = \sum_{i=0}^{n} i$. Vérifions la terminaison et la validité de l'algorithme.

```
def somme(n) :   # n entier > 0
    s = 0
    a = 0
    while a < n :
        a = a + 1
        s = s + a
    return s
```

Initialisation : $s_0 = \sum_0^0 i = 0$ et $a_0 = 0 < n \rightarrow$ OK

Transmission : admettons qu'à l'étape précédente $s_{j-1} = \sum_{i=0}^{a_{j-1}} i$ et $a_{j-1} < n$

or $a_{j-1} < n$ (car entrée dans la boucle), càd $a_{j-1} \leq n-1$ (puisque a_{j-1} et n entiers)

$s_j = s_{j-1} + a_j = \sum_{i=0}^{a_{j-1}} i + (a_{j-1}+1) = \sum_0^{a_{j-1}+1} i = \sum_0^{a_j} i$

$a_j = a_{j-1} + 1 \leq n - 1 + 1$ →OK

Sortie : $a_j \geq n$, et $a_j \leq n$ (invariant) : donc $a_j = n$, donc renvoie $s_j = \sum_{i=0}^{n} i \rightarrow$ OK c'est ce qui est attendu

- **Terminaison :** variant de boucle a entier strictement croissant, $\forall n$ la limite $a \geq n$ est donc forcément atteinte.
- **Validité :** on propose l'invariant de boucle : $s_j = \sum_{i=0}^{a_j} i$ et $a_j \leq n$

Que font les deux fonctions suivantes ? Vérifions leur terminaison et leur validité.

```
def fonction(n) :   # n entier > 0
    p = 1
    i = 1
    while i < n :
        i = i + 1
        p = p * i
    return p
```

Initialisation : $p_0 = 1! = i_0!$ et $n \geq 1 = i_0 \rightarrow$ OK

Transmission : admettons $p_{j-1} = i_{j-1}!$ et $i_{j-1} \leq n$, or $i_{j-1} < n$ (entrée boucle)

donc $i_{j-1} \leq n-1$ (car i_{j-1} et n entiers)

$p_j = p_{j-1} \times i_j = i_{j-1}! \times (i_{j-1}+1) = (i_{j-1}+1)! = i_j!$ et $i_j = i_{j-1} + 1 \leq n \rightarrow$ OK

Sortie : $i_j \geq n$, or $i_j \leq n$ (invariant), d'où $i_j = n$, et $p_j = i_j! = n! \rightarrow$ OK c'est ce qui est renvoyé

- **Objectif :** calcul factoriel $n!$
- **Terminaison :** *i* variant entier strictement croissant, finit forcément par dépasser *n* : OK
- **Validité :** on propose l'invariant de boucle : $p_j = i_j!$ et $i_j \leq n$

```
def fonction2(a,b) : # a et b entiers > 0
    p = 0
    m = 0
    while m < a :
        m = m + 1
        p = p + b
    return(p)
```

Initialisation : $p_0 = 0 = 0 \times b = m_0 \times b$ et $m_0 = 0 \leq a \rightarrow$ OK

Transmission : admettons $p_{j-1} = m_{j-1} \times b$ et $m_{j-1} \leq a$, or $m_{j-1} < a$ (entrée boucle)

donc $m_{j-1} \leq a-1$ (car m_{j-1} et a entiers)

$p_j = p_{j-1} + b = m_{j-1} \times b + b = (m_{j-1}+1) \times b = m_j \times b$

et $m_j = m_{j-1} + 1 < a \rightarrow$ OK

Sortie : $m_j \geq a$, or $m_j \leq a$ (invariant), d'où $m_j = a$, et $p_j = a \times b \rightarrow$ OK c'est ce qui est renvoyé

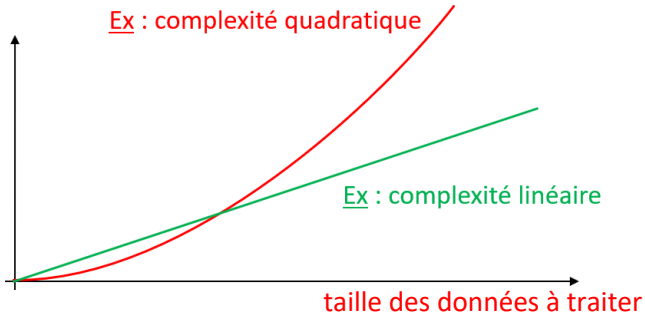
- **Objectif :** calcule $a \times b$
- **Terminaison :** *m* variant entier strictement croissant, finit forcément par dépasser *a* : OK
- **Validité :** on propose l'invariant de boucle : $p_j = m_j \times b$ et $m_j \leq a$

B / *Complexité* des algorithmes

Suite à l'étude da la validité d'un algorithme, nous sommes capables de déterminer si cet algorithme finira un jour par donner un résultat, et si ce résultat sera conforme à nos attentes. Toutefois, l'algorithme pourrait mettre plusieurs années avant de donner son résultat ! Il convient donc d'étudier à quel point l'algorithme que nous écrivons est ***complexe***, afin de choisir un nombre d'itérations convenable, en fonction du temps de simulation dont on dispose, et des ressources matérielles de notre ordinateur.

Plus un algorithme est **complexe,** plus le temps d'une itération va croître au cours du temps, et plus la quantité de données à stocker (variables) sera importante. On distingue donc deux types de complexités :

- Complexité en temps :
  Nombre total d'itérations nécessaires
  ou temps total algorithme
- Complexité en espace [de stockage] :
  taille des variables stockées
  dans la RAM (ou la swap)



Dans la plupart des cas usuels, la capacité de stockage en RAM des ordinateurs est moins limitante que le temps dont dispose son utilisateur. Ceci n'est pas vrai pour les grosses simulations destinées à de la recherche scientifique (calculs de plusieurs jours à plusieurs années !), mais généralement le cas pour de l'ingénierie (calculs de quelques secondes à quelques heures). On ne s'intéressera, en CPGE, qu'à la **complexité en temps** des algorithmes.

Pour évaluer la complexité en temps d'un algorithme, il nous faut évaluer son **temps d'exécution**. Pour cela, nous compterons le **nombre d'opérations** effectuées par l'algorithme indépendamment du type d'opérations réalisées.

Remarque *: En réalité, les différentes opérations élémentaires n'ont pas le même temps d'exécution.*

Rapide ⟶ ***Lent***

Affectation (=) — *Test (≤ , ==, ! =, etc.)* — *Addition (soustraction)* — *Multiplication* — *Division*

Attention ! Lorsqu'on utilise des fonctions externes (issues de bibliothèques ou de Python lui-même, comme *min* par exemple) celles-ci peuvent cacher un grand nombre d'opérations et on n'en connaît pas la complexité.
De plus, le temps exact d'exécution d'une opération dépend de nombreux paramètres

- Langage de programmation (particulièrement suivant qu'il est **compilé** comme le *C*, rapide, ou **interprété** comme Python, lent)
- De la fréquence du processeur (plus elle est élevée, plus on peut réaliser d'opérations par seconde) et de son architecture (certains processeurs sont plus performants pour certaines opérations)
- Du type de données, de leur stockage, des autres applications tournant en parallèle sur l'ordinateur, etc.

On ne cherchera donc pas à déterminer précisément le temps d'exécution, simplement à **approcher l'ordre de grandeur asymptotiquement**, on parlera alors ***d'ordre de complexité*** de l'algorithme. Si par exemple le temps d'exécution d'un script, déterminé précisément, est $4n^2 + 2n + 25$, n désignant ici la taille des données, alors :

- Alors quand $n \to \infty$ le temps d'exécution tend vers l'asymptote : $4n^2 + 2n + 25 \sim 4n^2$
- On cherche simplement une tendance, le chiffre significatif est donc ignoré : temps d'exécution $= \mathcal{O}(n^2)$

Rappel : Mathématiquement, $g(x) = \mathcal{O}(f(x))$ signifie que :

$$\exists x_0 \in \mathbb{R}^+, \exists k \in \mathbb{R}, \forall x > x_0, \quad g(x) \leq k.f(x) \ \to g \text{ est majorée par } k.f$$

c'est-à-dire que le rapport $g(x)/f(x)$ est borné pour x suffisamment grand.

En pratique, lorsqu'on réfléchit à un algorithme, ce qui nous intéressera n'est pas le nombre d'opérations exact, mais ***l'ordre de complexité*** (en temps), qui traduira la façon dont le temps de calcul est affecté par la taille des données.

Soit un algorithme **itératif** dont le **temps d'exécution** en fonction du nombre d'itérations *n* s'écrit t_n.
On dit que la **complexité en temps** est une **complexité en** $\mathcal{O}(f(n))$ ssi

$\frac{t_n}{f(n)}$ tend vers une constante quand $n \to \infty$

Exemples classiques :

Ordre de Complexité	Nom	Ordre de grandeur de temps pour $n = 10^6$	Remarques
$O(1)$	temps constant	1 ns	Le temps ne dépend pas de la taille des données. Très rare.
$O(\log n)$	logarithmique	10 ns	Exécution quasi instantanée, algorithme extrêmement rapide.
$O(n)$	linéaire	1 ms	Exécution, très rapide. Problème de mémoire avant que le temps d'exécution soit un problème.
$O(n^2)$	quadratique	15 min	Commence à devenir problématique lorsque la taille des données est grande.
$O(n^k)$	polynomiale	30 ans ($k = 3$)	
$O(2^n)$	exponentielle	$> 10^{300000}$ milliards d'années	inutilisable sauf pour de très très petites données.

Pour réaliser un même objectif, plusieurs algorithmes sont souvent possibles. L'algorithme le plus « évident », qui consistera souvent à scanner la totalité des données sans optimisation (recherche d'un élément dans une liste), à considérer la totalité des cas possibles (méthodes *bruteforce*), sera souvent appelé **algorithme naïf**, d'ordre de complexité sous-optimal. Dans le cadre du programme, nous mettrons en évidence que pour des tailles de données élevées, ces algorithmes sont limitants, et on envisagera des algorithmes optimisés, s'appuyant sur des méthodes de type ***« diviser pour régner »,*** ou sur de la décomposition d'un problème en plusieurs sous problèmes de dimensions moindres (ex : programmation dynamique), qui permettront de faire baisser l'ordre de complexité du code.

B.1 / Comment prévoir l'ordre de complexité d'un algorithme ?

Voici une série de règles simples pour le calcul de coût :

- Structure conditionnelle `IF` : on ne garde que le plus grand des coûts des ≠ branches
- Structure itérative `FOR` : on compte le nombre N_b de boucles réalisées, coût = N_b × coût d'une boucle
- Structure itérative `WHILE` : même chose, mais la détermination du nombre de boucles est souvent plus difficile

Remarque 1 : Lorsque plusieurs structures sont imbriquées, on multiplie leurs coûts.
Remarque 2 : Attention aux appels de fonctions : en une seule ligne un très grand nombre d'opérations peut être réalisé !

Exemples : Considérons les trois algorithmes suivants dont le résultat est identique.

```
#Algorithme 1 :
n = 100
Res = n*2*(1 + n)*n/2
print(Res)
```

$\mathcal{O}(1)$ opérations

```
#Algorithme 2 :
Res = 0
n = 100
for i in range(1,n+1) : # Somme des termes de 1 à n
    Res = Res + 2*n*i
print (Res)
```

soit $\mathcal{O}(n)$ opérations

```
#Algorithme 3 :
Res = 0
n = 100
for i in range(1,n+1) : # Somme des termes de 1 à n
    for j in range(1,n+1) : # Somme des termes de 1 à n
        Res = Res + i + j
print(Res)
```

$\mathcal{O}(n^2)$ opérations

Application : Donnons l'ordre de complexité des trois fonctions suivantes :

```
def table1(n) :
    for i in range (11) :
        print(i*n)
```

$\mathcal{O}(1)$

```
def table2(n) :
    for i in range (n) :
        print(i*i)
```

$\mathcal{O}(n)$

```
def table3(n) :
    for i in range (n) :
        for j in range (n) :
            print(i*j, end=" ")
        print()
```

$\mathcal{O}(n^2)$

B. 2 / La complexité peut-elle dépendre des données ?

Selon l'organisation des données en « entrée » d'un algorithme, le nombre d'itérations peut ne pas être le même. Si le nombre d'itérations n'est pas pré-défini en amont du code (typiquement, si on utilise un `while`), il pourra par exemple arriver que l'algorithme termine en une seule itération pour certaines données !
Par exemple, comme nous le verrons en TD, pour un algorithme de tri par ordre croissant, le temps de calcul peut être sensiblement différent selon que la liste en entrée est déjà triée dans le bon ordre, ou non (voire carrément triée par ordre décroissant).

On appellera ***ordre de complexité :***

- **dans le pire des cas** : le cas où la donnée initiale est arrangée dans le sens le plus défavorable

  Remarque 1 : par défaut, quand on vous demande l'ordre de complexité sans préciser lequel, c'est dans le pire des cas
  Remarque 2 : ce sera à nous d'intuiter quelle est la structure de donnée la plus défavorable

- **dans le meilleur des cas** : le cas où la donnée initiale est arrangée dans le sens le plus favorable

- **en moyenne** : statistiquement, si l'on tirait des données aux hasard, et qu'on répétait la mesure de temps d'exécution un grand nombre de fois pour chaque taille de donnée, c'est l'ordre de complexité qu'on mesurerait.

  Remarque : en TD, c'est la complexité moyenne que nous mesurerons. Il est important d'itérer chaque point de mesure de temps de calcul sur plusieurs données de même taille, pour qu'en moyenne on soit aussi loin du cas de l'organisation de cette donnée la plus favorable que du + défavorable. Et ceci pour chaque taille de donnée !