# TP n°6 - La complexité

## 1. Introduction

Étant donné un algorithme, on aimerait savoir si celui-ci est "efficace" : si on implémente cet algorithme sur une machine (si on crée un programme correspondant à cet algorithme donc), va-t-il s'exécuter rapidement ? Va-t-il consommer beaucoup de mémoire pendant son exécution ? On peut quantifier ces problèmes grâce à la complexité temporelle qui nous donnera un ordre de grandeur du temps d'exécution de l'algorithme et à la complexité spatiale qui nous donnera un ordre de grandeur de la quantité de mémoire requise à l'exécution de l'algorithme. L'objet du présent T.P. est la complexité temporelle.

L'intérêt de la complexité temporelle est de pouvoir évaluer la rapidité d'un algorithme indépendamment de la machine sur laquelle il est exécuté. En effet, la première idée qui pourrait venir pour savoir si un algorithme est efficace, serait de mesurer le temps d'exécution de celui-ci : le problème est que ce temps dépend non seulement de la machine mais aussi des processus déjà en cours sur la machine : cette mesure de temps n'est donc pas fiable pour comparer universellement deux algorithmes.

Il nous vient alors l'idée de calculer le nombre de d'opérations élémentaires effectuées dans l'algorithme pendant son exécution. Ainsi, si on part du principe que chaque opération élémentaire (comme on va la définir) prend un temps identique pour s'effectuer, en évaluant le nombre de ces opérations, on obtiendra le temps d'exécution de l'algorithme. Voyons comment calculer le nombre d'opérations élémentaires d'un algorithme et surtout, définissons ce qu'est une opération élémentaire !

## 2. Le coût d'un algorithme

**Définition 1.**

On appelle **coût** d'un algorithme le nombre d'opérations élémentaires effectuées par l'algorithme.

Mais comment définit-on une opération élémentaire ?

**Définition 2.**

On appelle **opération élémentaire** une des actions suivantes (la liste n'est pas exhaustive) :

- Faire une opération mathématique : addition, soustraction, multiplication, division, multiplication matricielle, puissance... ;
- Lire le contenu d'une variable ;
- Affecter une valeur à une variable ;
- Effectuer un test : `==`, `>=`, `!=`,... ;
- Faire une opération booléenne : `and`, `not`,...

Voici un exemple :

```
1 def sommeCarre(n):
```

```
    """ Somme des n premiers carres non nuls """
    S=0
    while n>0:
        S=S+n**2
        n=n-1
    return S
```

**Question.** Apparté

Au fait, quelle est la formule pour la somme des carrés de $n$ premiers entiers plus grand que 1 ?

**Question 1.**

Quel est le coût de cette algorithme ?

Si on compte toutes les opérations élémentaires citées plus haut, on obtient :
— $n + n + n$ opérations mathématiques ;
— $n + n + n + n + 1$ lectures ;
— $1 + n + n$ affectations ;
— $n$ comparaisons
Donc le coût total est de $10n + 2$ opérations élémentaires.

On remarque alors deux choses qui semblent peu cohérentes dans ces considérations :
- Parmi ces opérations, certaines prennent sûrement plus de temps que d'autres !
- Pour une même opération, le temps d'exécution est proportionnel à la taille des données !

Comment avoir un calcul de coût pertinent dans ces conditions ? Et bien on va faire les hypothèses suivantes afin d'une part, de rendre le calcul de coût plus facile, et d'autre part, de le rendre représentatif de l'efficacité de l'algorithme :

**Hypothèse de calcul du coût**

1. On fera toujours l'approximation que toutes les opérations élémentaires prennent le même temps ;
2. Dans le calcul du coût, on ne comptera que les opérations élémentaires les plus significatives de l'algorithme.

La deuxième hypothèse est peut sembler subjective : et bien elle l'est ! Mais dans la plupart des cas, il y aura bien une opération plus important que les autres, et qui est la plus pertinente à sélectionner en fonction du contexte de l'algorithme. Par exemple, pour l'algorithme précédent, ce sont les opérations mathématiques (on calcule une somme de carrés) qui constitue les opérations importantes pour cet algorithme. et ainsi, notre calcul de coût devient $3n$ pour ces opérations.

On voit bien dans notre exemple que le coût d'un algorithme dépend fortement de l'argument $n$. Ainsi, le coût dépend de la taille des données ; mais qu'entend-on exactement par "taille des données" ?

**Définition 3.**

**La taille des données** d'un algorithme est (en général) un entier $n$ qui mesure la données à traiter. Comme un algorithme est ici écrit comme une fonction, les données à traiter sont les paramètres d'appels de cette fonction.

Les cas les plus fréquents sont :

- donnée = entier $n \Rightarrow$ taille $= n$
- donnée = liste $\Rightarrow$ taille = longueur de la liste
- donnée = plusieurs entiers $\Rightarrow$ taille = le plus grand de ces entiers (par exemple).

**Exemple.** Calcul de coût

Considérons l'algorithme suivant qui tri dans l'ordre croissant une liste de nombres donnée en argument :

```
1 def tri(L):
2     n = len(L)
3     for i in range(n - 1):
4         for j in range(i, n):
5             if L[i] > L[j]:
6                 x = L[i]
7                 L[i] = L[j]
8                 L[j] = x
9     return L
```

Ici, les opérations qui semblent prédominantes sont les tests et les affections des variables. Calculons le nombres de ces opérations dans l'algorithme :

— Les tests : il y en a

$$\sum_{i=0}^{n-2} n-1-i = \sum_{j=1}^{n-1} j = \frac{n(n-1)}{2}$$

— Les affectations : il y en a 1 au départ et 1 à la fin et en fonction de la véracité de chaque test il y en a entre 0 et

$$3\sum_{i=0}^{n-2} n-1-i = 3\frac{n(n-1)}{2}$$

On remarque sur cet exemple simple que le coût dépend non seulement de la taille, mais aussi de la valeur des données !

**Exercice 1.**

Calculer le coût, en terme d'opérations mathématiques i.e. additions, produits et racines de l'algorithme suivant en fonction de la donnée $n$ :

```python
def mystere(n):
    d=0
    for k in range(1,n):
        if n%k==0:
            d=d+1
    return d
```

1. Que renvoie cet algorithme étant donné un argument `n`$\in \mathbb{N}^*$.
2. Quel est le coût de cet algorithme ?

## 3. Calcul pratique du coût d'un algorithme itératif

### a. Coût d'un algorithme itératif - boucle `for`

Grâce à l'exemple précédent, on observe que le coût d'un bloc d'une boucle `for` revient s'obtient de la manière suivante :

**Calcul du coût d'une boucle `for`**

```python
for i in range(N):
    instructions(i) #c_f(i) opérations
```

- Pour chaque `i` de la boucle `for`, on détermine le nombre $c_f(i)$ d'opérations élémentaires effectuées par `instructions(i)` dans le bloc de la boucle `for` ;
- Puis on somme ces nombres $c_f(i)$ d'opérations pour $i$ allant de l'indice de début de la boucle `for` (ici 0) à l'indice de fin de la boucle (ici $N-1$).
  Dans le cas précédent, cela donne :
$$\sum_{i=0}^{N-1} c_f(i).$$

**Exercice 2.**

Calculer le coût des algorithmes suivants en termes d'opérations arithmétiques :

1.
```python
def f1(n):
    a=1
    for i in range(n):
        a=a+i*3
    return a
```

2.
```
def f2(n):
    a=1
    for i in range(n):
        a=1+2*a
    for j in range(n):
        a=a//2
    return a
```

3.
```
def f3(n):
    x=1
    for i in range(n):
        for j in range(n):
            x=x+5
    return x
```

**Exercice 3.**

Calculer le coût, en terme d'opérations mathématiques i.e. additions, produits et racines de l'algorithme suivant en fonction de la donnée $n$ :

```
def mystere(a,b,n):
    for i in range(n):
        a,b=0.5*(a+b),sqrt(a*b)
    return b
```

Que calcule et à quoi correspond cet algorithme ???

### b. Coût d'un algorithme itératif - boucle `while`

Dans le cas d'une boucle `while`, le principe est le même qu'avec une boucle `for` mais une difficulté apparaît : alors qu'on connaît le nombre de tours de boucle d'un `for`, le nombre de tour d'un `while` n'est pas connu a priori !
Il faut donc le déterminer afin de pouvoir effectuer le même calcul que pour une boucle `for`.

> **Calcul du coût d'une boucle `while`**

```
u=valeur_initiale
while condition(u):
    instructions(u) #u n'est pas modifié ici
    u=modif(u)
```

- Pour chaque `i` de la boucle **`while`**, on détermine le nombre $c_w(u)$ d'opérations élémentaires effectuées par `instructions(u)` et `u=modif(u)` dans le bloc de la boucle **`while`** ;
- On modélise les valeurs de $u$ lors de chaque tour de la boucle **`while`** grâce à une suite récurrente $(u_k)_{k\in\mathbb{N}}$ :

$$\begin{cases} u_0 &= \texttt{valeur\_initiale} \\ u_k &= \texttt{modif}(u_{k-1}) \text{ pour } k \in \mathbb{N}^*. \end{cases}$$

On utilise alors nos talents mathématiques pour déterminer une expression explicite de cette suite pour $k \in \mathbb{N}$ :

$$u_k = f(k)$$

Grâce à cela, on peut déterminer le nombre de tours $N$ de la boucle **`while`**. En effet, ce nombre $N$ est déterminé par :

$$\texttt{condition}(u_{N-1}) \text{ est } \textbf{VRAI !} \text{ et } \texttt{condition}(u_N) \text{ est } \textbf{FAUX !}$$

Dans la pratique, on sera souvent amené à utiliser la fonction réciproque de la fonction $f$ telle que $u_k = f(k)$ pour déterminer $N$ (si tant est quelle soit bijective !).
De plus, il arrivera très souvent que l'on ne puisse pas déterminer exactement le nombre $N$ ! On essaiera alors d'obtenir un encadrement de $N$ en fonction de la taille de la donnée $n$.

On obtient ainsi toutes les valeurs de $u$, il s'agit de :

$$u_0, u_1, ..., u_{N-1}.$$

- Finalement, on somme les nombres $c_w(u)$ d'opérations pour chaque valeur de $u$.
Dans le cas précédent, cela donne :

$$\sum_{k=0}^{N-1} c_w(u_k).$$

**Exemple 1.**

Voici un exemple concret :

```python
def algo(n):
    a=1
    S=0
    while a<=n:
        S=S+a
        a=a*2
    return S
```

On calcule le coût de `algo(n)` en termes d'opérations arithmétiques :

- Il y a 2 opérations arithmétiques dans le bloc de la boucle `while`.
- Soit $(a_k)$ la suite des valeurs de la variables $a$. Alors, on a :

$$\begin{cases} a_0 &= 1 \\ a_k &= 2a_{k-1} \text{ pour } k \in \mathbb{N}^*. \end{cases}$$

Ainsi, en calculant de deux manières le produit $\prod_{i=1}^{k} \frac{a_i}{a_{i-1}}$, on obtient :

$$a_k = \frac{a_k}{a_0} = \prod_{i=1}^{k} \frac{a_i}{a_{i-1}} = 2^k.$$

Le nombre de tours $N$ de la boucle `while` vérifie alors

$$a_{N-1} \leq n \text{ i.e. } 2^{N-1} \leq n \text{ et } a_N > n \text{ i.e. } 2^N > n;$$

d'où l'encadrement :

$$\log_2(n) < N \leq \log_2(n) + 1.$$

*bu* On peut alors calculer le coût $c(n)$ de la boucle `while` et a fortiori de `algo(n)` puisqu'il n'y a pas opérations en dehors de cette boucle :

$$c(n) = \sum_{k=0}^{N-1} 2 = 2N$$

Or, d'après l'encadrement précédent, on a :

$$2\log_2(n) < c(n) \leq 2\log_2(n) + 2.$$

*Remarque :* Nous verrons dans la suite qu'un tel encadrement nous convient très bien !

**Exercice 4.**

Déterminer le coût des algorithmes suivant en termes d'opérations arithmétiques :

1.
```python
def f1(n):
    u=1
    while u<=2*n:
        u+=1
    return u
```

2.
```python
def f2(n):
    u=2**n
    P=1
    while u>1:
        P=P*u
        u=u//2
    return P
```

3.
```python
def f3(n):
    a=1
    S=0
    while a<=n:
          for i in range(a):
              S+=i
          a=a*2
    return S
```

## 4. La complexité

Malgré la simplicité de nos exemples, on peut voir que le calcul du coût peut très vite devenir impossible pour des algorithmes très grands. Et on remarque que si $n$ est très grand, le fait qu'il y ait $n^2$ opérations où $n^2+5n+6$ opérations revient quasiment au même puisque les autres termes sont négligeables devant $n^2$.

Ainsi on a l'idée de définir la complexité d'un algorithme de la façon suivante :

**Définition 4.** *Complexité*

La **complexité** d'un algorithme est une fonction simple telle que le coût de l'algorithme (pour des opérations élémentaires données) *dans le cas le plus défavorable* est dominée par quand la taille des données tend vers l'infini.

Ainsi, si on note $n$ la taille des données, et $c(n)$ le coût ; la complexité $C(n)$ de l'algorithme vérifie :

$$c(n) = O(C(n)).$$

**Remarque 1.**

Pour rappeler que la complexité est une approximation asymptotique, on notera souvent $C(n) = O(n^2)$ par exemple au lieu de $C(n) = n^2$.

Quelques fonctions de complexité

| | | |
|---|---|---|
| $C(n) = O(1)$ | complexité constante | extraordinaire ! |
| $C(n) = O(\ln(n))$ | complexité logarithmique | excellent |
| $C(n) = O(n)$ | complexité linéaire | super bien |
| $C(n) = O(n\ln(n))$ | complexité quasi-linéaire | très bien |
| $C(n) = O(n^2)$ | complexité quadratique | moyen |
| $C(n) = O(n^3)$ | complexité cubique | mauvais |
| $C(n) = O(n^p)$ | complexité polynomiale ($p > 3$) | très mauvais |
| $C(n) = O(a^n)$ | complexité exponentielle ($a > 1$) | horrible |
| $C(n) = O(n!)$ | complexité factorielle | à proscrire |

| | log(n) | n | n log(n) | $n^2$ | $n^3$ | $2^n$ | n ! |
|---|---|---|---|---|---|---|---|
| $10^2$ | 7 ns | 100 ns | 0,7 µs | 10 µs | 1 ms | $4.10^{13}$ ans | $3.10^{141}$ ans |
| $10^3$ | 10 ns | 1 µs | 10 µs | 1 ms | 1 s | $10^{292}$ ans | $10^{2560}$ ans |
| $10^4$ | 13 ns | 10 µs | 133 µs | 100 ms | 17 s | $6.10^{3002}$ ans | |
| $10^5$ | 17 ns | 100 µs | 2 ms | 10 s | 11,6 jours | | |
| $10^6$ | 20 ns | 1 ms | 20 ms | 17 min | 32 ans | | |

## 5. Calcul de Complexité

### a. Exemple :

Pour un algorithme composé de structures itératives (boucles `for`, boucles `while`), il s'agit donc simplement de calculer des sommes pour obtenir le coût dans le pire des cas, et de chercher une fonction simple qui domine ce coût pour obtenir la complexité :

Revenons à notre tri de l'exemple précédent :

```
1 def tri(L):
2     n = len(L)
3     for i in range(n - 1):
4         for j in range(i, n):
5             if L[i] > L[j]:
6                 x = L[i]
7                 L[i] = L[j]
8                 L[j] = x
9     return L
```

On a vu que dans le pire des cas, $c(n) = 2 + 4\frac{n(n-1)}{2}$. Donc $C(n) = O(n^2)$ ! La complexité est quadratique pour cet algorithme.

### b. Exercices basiques

**Exercice 5.** *Factorielle et binôme de Newton*

1. Écrire une fonction `facto(n)` qui renvoie la factorielle du nombre `n`
2. Calculer sa complexité.
3. Écrire une fonction `binom(n,p)` qui renvoie la valeur du binôme de Newton $\binom{n}{p}$ en vous servant de la fonction `facto` précédemment écrite.
4. Calculer sa complexité en fonction de $n$ (et du pire des cas pour $p$).

**Exercice 6.**

Calculer la complexité des 6 algorithmes suivants (en terme d'opérations arithmétiques) :

```python
#Algo f1
def f1(n):
    x = 0
    for i in range(n):
        for j in range(n):
            x = x+1
    return x

#Algo f2
def f2(n):
    x = 0
    for i in range(n):
        for j in range(i):
            x = x+1
    return x

#Algo f3
def f3(n):
    x = 0
    for i in range(n):
        j = 0
        while j * j < i:
            x += 1
            j += 1
    return x
```

```python
#Algo f4
def f4(n):
    x = 0
    i = n
    while i > 1:
        x += 1
        i //= 2
    return x

#Algo f5
def f5(n):
    x,i = 0,n
    while i > 1:
        for j in range(n):
            x+=1
        i//=2
    return x

#Algo f6
def f6(n):
    x,i = 0,n
    while i > 1:
        for j in range(i):
            x+=1
        i //=2
    return x
```

**Exercice 7.**

Considérons l'algorithme suivant : (remarque, si ce n'est pas déjà fait, n'oubliez pas le `from math import *` pour avoir accès au fonctions mathématiques de Python)

```python
def mystere(n):
    for d in range(2, floor(sqrt(n)) + 1):
        if n % d == 0:
            return False
    return True
```

1. Que détermine l'algorithme suivant ?
2. Calculer sa complexité en fonction de $n$ en prenant en compte les congruences (les `%`) comme opérations élémentaires.

## 6. Mise en pratique : Exercices sur la complexité temporelle

**Exercice 8.**

Déterminer la complexité des algorithmes suivants en terme d'opérations arithmétiques :

```python
#Algo1
def f1(n):
    p = 0
        for i in range(n):
            for j in range(n):
                for k in range(n):
                    s = 2 * s
```

```python
#Algo2
def f2(n):
    p = 0
        for i in range(n):
            s = s + 3
        for j in range(n):
            s = s + 3
```

```python
#Algo3
def f3(n):
    s = 0
        for i in range(n):
            for j in range(i,n):
                s = s + j**2
```

```python
#Algo4
def f4(n):
    p = 1
        for i in range(n):
            for j in range(i-5,i+5):
                p = 2*p
```

```python
#Algo5
def f5(n):
    s = 0
    for i in range(n):
        a = n
        while a > 1:
            a = a/2
        s += 1
```

```python
#Algo6
def f6(n):
    i = n
    s = 0
    while i > 1:
        for j in range(i):
            s = s + 1
        i = i/2
```

**Exercice 9.**

Soit $x \in \mathbb{R}$. On considère la suite récurrente $(u_n)_{n\in\mathbb{N}}$ telle que :

$$\begin{cases} u_0 = 1 \\ u_{n+1} = \frac{xu_n}{n+1} \quad \forall\, n \in \mathbb{N}^*. \end{cases}$$

1. Écrire une fonction `u(x,n)` qui renvoie le terme $u_n$ d'indice `n` de la suite $(u_n)_{n\in\mathbb{N}}$.
2. Que fait la fonction `mystere` suivante ?

```python
def mystere(x,n):
    S=0
    for i in range(n+1):
        S=S+u(x,i)
    return S
```

3. Montrer que pour tout $n \in \mathbb{N}$, $u_n = \frac{x^n}{n!}$. En déduire une formule mathématique pour `mystere(x,n)`.
4. Importer la fonction ln du module `math` grâce à l'instruction `from math import log` (Attention `log` désigne ici le logarithme népérien et pas le logarithme en base 10 !).
   — Calculer `mystere(log(10),n)` pour $n = 1, 5, 10, 1000$. Vers quelle valeur `mystere(log(10),n)` semble-t-elle converger ?

— Calculer `mystere(2*log(10),n)` pour $n = 1, 5, 10, 1000$. Vers quelle valeur `mystere(2*log(10),n)` semble-t-elle converger ?

— Calculer `mystere(5*log(10),n)` pour $n = 1, 5, 10, 1000$. Vers quelle valeur `mystere(5*log(10),n)` semble-t-elle converger ?

— Calculer `mystere(10*log(10),n)` pour $n = 1, 5, 10, 1000$. Vers quelle valeur `mystere(10*log(10),n)` semble-t-elle converger ?

Conjecturer une formule pour la limite de `mystere(x*log(10),n)` quand $n$ tend vers l'infini, puis pour `mystere(x,n)`. En "déduire" une formule mathématique reliant la fonction exponentielle et une somme infinie.

**Exercice 10.**

On considère les algorithmes suivants :

— Se terminent-t-ils ?

— Si oui, déterminer leurs coûts puis leurs complexités.

```python
#Algo1
def f1(n):
    a = 1
    while a < n:
        a = a + 5
```

```python
#Algo2
def f2(n):
    a = 1
    while a < n:
        a = 2 * a
```

```python
#Algo3
def f3(n):
    while n >= 0:
        n = n//2
```

```python
#Algo4
def f4(n):
    a=1
    a = 1
    while a < n**2:
        a = 3*a+1
```

**Exercice 11.** *Nombres de Bell*

Pour $n \in \mathbb{N}$, on définit le nombre de Bell $B_n$ par récurrence :

$$\begin{cases} B_0 = 1 \\ B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_k \end{cases}$$

1. Écrire une fonction `Bell(n)` qui prend pour argument un entier naturel `n` et qui renvoie la valeur de $B_n$. On pourra réutiliser la fonction `binom` codée la semaine dernière.
2. Quelle est la complexité temporelle de votre algorithme en terme d'opérations arithmétiques
3. Soit $E$ un ensemble. Une famille $(A_i)_{i\in I}$ de parties de $E$ est appelée **partition** de $E$ si, pour tout $i, j \in I$ avec $i \neq j$, $A_i \cap A_j = \emptyset$ et $\bigcup_{i\in I} A_i = E$. Par exemple,

   $$(\{1\}, \{2,5\}, \{3,4,6\}) \text{ est une partition de } \{1,2,3,4,5,6\}.$$

   Soit $n \in \mathbb{N}$. On considère l'ensemble :

   $$[\![1, n]\!] = \{1, \ldots, n\}.$$

   On convient que pour $n = 0$, $[\![1, 0]\!] = \emptyset$.

   Montrer que le nombre de partitions de $[\![1, n]\!]$ est égal à $B_n$.
4. *Question Bonus très dure !* Écrire un algorithme `partition(n)` qui renvoie la liste des partitions de $[\![1, n]\!]$. Une partition sera représentée par une liste de listes : la partition $(\{1\}, \{2,5\}, \{3,4,6\})$ de l'exemple plus haut sera donc codée `[[1],[2,5],[3,4,6]]`

**Exercice 12.** *Algorithme de calcul de puissances*

1. *Algorithme naïf* :

   Écrire, en utilisant un algorithme simple, une fonction `puissance(x,n):` qui retourne le nombre `x` à la puissance `n` où `n` est un entier naturel. Quelle est la complexité de cet algorithme ?

2. *Algorithme d'exponentiation rapide* :

   On cherche à déterminer un algorithme avec une meilleure complexité pour calculer une puissance. On part du constat suivant :
   soit $x \in \mathbb{R}$. Alors pour $n \in \mathbb{N}$,

$$x^n = \begin{cases} x^{\frac{n}{2}}.x^{\frac{n}{2}} & \text{si } n \text{ est pair} \\ x^{\frac{n-1}{2}}.x^{\frac{n-1}{2}}.x & \text{si } n \text{ est impair} \end{cases}$$

   En utilisant cette remarque et la boucle suivante :

```
while n>0:
    ... #à vous de jouer
    n=n//2
```

   écrire une fonction `puissance_rapide(x,n):` qui retourne le nombre `x` à la puissance `n` où `n` est un entier naturel. Quelle est la complexité de cet algorithme ?