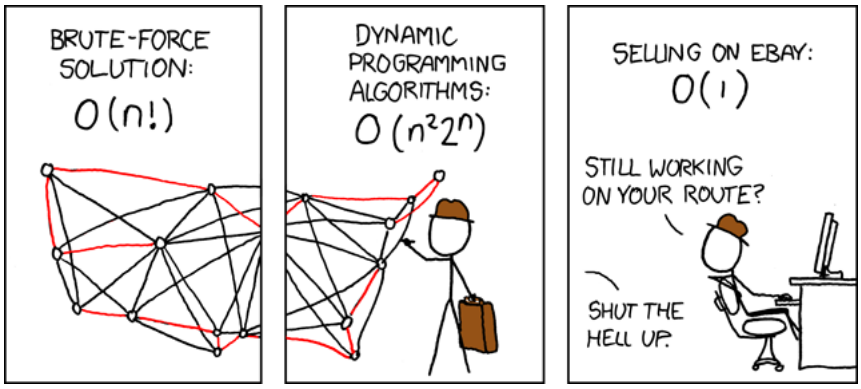
Lycée Loges

Chapitre 6
Informatique théorique

Simon Dauguet
simon.dauguet@gmail.com

9 janvier 2025



Jusque là, les questions de corrections, d'exactitudes des algorithmes créés ont été largement mis de côté. Mais rien ne nous garantit que les algorithmes créés pour répondre à un problème répondent effectivement bien au problème posé. Les algorithmes ont beau avoir été testés sur des exemples, il est possible que l'algorithme en question ne donne pas dans tous les cas le bon résultat. Une série de tests n'est pas une preuve mathématiques (donc certaine) du bon fonctionnement de l'algorithme. On aurait pu jouer de malchance et ne tester que les quelques rares situations où le résultat correspond aux attentes par erreurs.

Aussi, pour s'assurer du bon fonctionnement d'un algorithme, une étude théorique (en fait, mathématique) va être nécessaire. Cette étude va s'axer sur 3 grand pôles.

- Il faut commencer par montrer que l'algorithme "tourne bien", c'est-à-dire qu'il démarre correctement et fini par s'arrêter pour renvoyer un résultat. Clairement, s'il ne s'arrête jamais, il ne renverra rien et il n'aura aucune utilité.
- Il faudra ensuite montre évidemment que ce qui est renvoyé par l'algorithme correspond bien à la solution du problème posé. Si on demande une couleur à un algorithme et qu'il renvoie une recette de cuisine, toute délicieuse soit-elle, ça n'a pas d'intérêt.
- Il faudra ensuite s'enquérir de problème de limite physique. C'est-à-dire à ce que les deux études précédentes sont théoriques. Il est possible de faire un algorithme qui s'arrête et donne la bonne

solution, mais dans un temps déraisonnablement long. S'il faut attendre plusieurs centaines d'années pour obtenir la réponse, l'algorithme est utile en théorie, mais pas en pratique.

Table des matières

1 Terminaison et correction

Définition 1.1 (Terminaison et correction d'un algorithme) :
L'étude de la *terminaison* d'un algorithme correspond à la preuve que l'algorithme s'arrêtera (un jour, à un moment donné).

L'étude de la *correction* d'un algorithme correspond à la preuve que l'algorithme renverra bien ce qui est attendu, ce pour quoi il est fait.

La terminaison et la correction d'un algorithme correspondent donc à des études théoriques, mathématiques, du "bon" déroulement de l'algorithme. La terminaison et la correction forment donc la preuve scientifique du fait que l'algorithme apporte bien une réponse au problème posé.

1.1 Jeu de tests d'un algorithme

Définition 1.2 (Jeu de tests d'un algorithme) :
Proposer un jeu de tests d'un algorithme correspond à prendre des valeurs particulières des entrées de l'algorithme et de suivre les valeurs prises successivement par chacune des variables locales de l'algorithme, pour "voir" ce qu'il se passe et comment fonctionne l'algorithme.

Les jeux de tests d'un algorithme permettent de mieux se rendre du fonctionnement interne de l'algorithme (sorte d'autopsie, on met les tripes de l'algorithme à l'air et on regarde comment elles se comportent). Ce qui permet en particulier de pouvoir se rendre d'éventuelles difficultés ou problèmes. C'est un moyen de débogage.

Exemple 1.1 :
On propose le code suivant :

```
1 def SyraLike(n:int) -> int :
2     u = 6
3     for k in range(n) :
4         u = 2*u-5
5     return(u)
```

Un jeu de test avec $n = 3$ est alors :

ligne	`u`	`k`	Sortie

Exercice 1 :
On reprend la suite de l'exemple précédent ($u_0 = 6$ et $\forall n \in \mathbb{N}$, $u_{n+1} = 2u_n - 5$). Il est facile de montrer que cette suite tend vers $+\infty$. Écrire un algorithme `SeuilU(A:float) -> int` qui renvoie le premier indice $n \in \mathbb{N}$ tel que $u_n > A$. Proposer un jeu de tests de cet algorithme pour $A = 30$.

1.2 Terminaison d'un algorithme

Définition 1.3 (Variant de boucle) :
Un variant de boucle est une variable de la boucle strictement décroissante à chaque itération de la boucle. Les itérations s'arrêtant quand le variant de boucle devient négatif.

Exemple 1.2 :
Dans une boucle `for`, le variant de boucle est la distance entre la variable d'incrémentation et la valeur finale de la boucle. Plus précisément, dans un boucle `for k in range(a,b)`, le variant de boucle est la valeur $b - 1 - k$.

Remarque :
Dans le cadre d'une boucle `for`, le variant de boucle n'est pas très mystérieux et la terminaison de l'algorithme est automatique. C'est moins clair avec une boucle `while` (c'est le problème des boucles infinies).

L'algirthme s'arrête donc, dès qu'un variant de boucle atteint 0. Cet argument repose sur le résultat mathématique suivant :

Proposition 1.1 :
Toute suite d'entiers naturels strictement décroissante ne peut prendre qu'un nombre fini de valeurs.

Autrement dit, avec un variant de boucle, il finira toujours par devenir négatif. Et donc l'algorithme s'arrêtera.

Exercice 2 :
Étudier les terminaisons des deux algorithmes suivant :

```
c=1
p=a
while c <= b :
    p = p+a
```

```
c=1
p=a
while c <= b :
    p=p+a
    c=c+1
```

Exercice 3 :
On considère l'algorithme suivant :

```
a=A
b=B
while a != b :
    if a>b :
        a = a-b
    else :
        b = b-a
print(a)
```

1. Proposer un jeu de tests pour $a = 15$ et $b = 6$.
2. Que semble faire cet algorithme ?
3. Faire une preuve de la terminaison de cet algorithme.

Remarque :
Tout algorithme ne se termine pas forcément. Même sans problème de boucle infinie. Par exemple, la conjecture de Syracuse est toujours un problème ouvert aujourd'hui. Autrement dit, en terme informatique, l'algorithme qui démarre à $u_0 \in \mathbb{N}$ et qui calcule les termes de la suite de Syracuse successif ($u_{n+1} = u_n/2$ si u_n pair et $u_{n+1} = 3u_n + 1$ sinon) et s'arrêtant quand on tombe sur la valeur 1, est un algorithme dont

la preuve de la terminaison n'est pas connue aujourd'hui. Tous les jeux de tests se sont terminés, jusque là. Mais on a pas la preuve que c'est toujours le cas.

La preuve de cette terminaison rapporterait 1 million de dollars à son auteur.

1.3 Correction d'une algorithme

Faire la preuve de la correction correspond à montrer qu'une propriété reste valable tout au long de l'algirthme. Cette propriété devant être en lien avec ce que l'on cherche. De sorte que, si l'algorithme se termine bien, on aura à la fin, la même propriété qu'au début. Et la propriété du début étant c'elle que l'on veut, on aura aussi ce que l'on veut à la fin.

Ceci se fait au moyen d'un invariant de boucle.

Définition 1.4 (Invariant de boucle) :
Un invariant est une propriété initialement vraie à l'entrée de la boucle et qui reste vraie à chaque itération de la boucle.

Exemple 1.3 :
Considérons l'algorithme

```
1 c=0
2 p=0
3 while c < b :
4     p = p+a
5     c = c+1
6 print(p)
```

Posons p_0 la valeur initiale de la variable `p` (donc $p_0 = 0$). Soit $n \in \mathbb{N}$ le nombre d'itération de la boucle `while`. On pose $\forall k \in \{0, \dots, n\}$, p_k la valeur de la variable `p` à la fin de la k-ème itération de la boucle `while`.

Montrons que la propriété $\forall k \in \{0, \dots, n\}$, $p_k = ka$ (c'est notre invariant de boucle).

Pour $k = 0$, on a $p_0 = 0 = 0 \times a$. Donc la propriété est initialement vraie.

Soit $k \in \{0, \dots, n-1\}$. Supposons que $p_k = ka$. Donc à la fin de la k-ème itération, la variable `p` contient la valeur ka. Alors, à la fin de la $(k+1)$-ème itération, on aura $p_{k+1} = p_k + a = (k+1)a$.

Donc, par principe de récurrence, $\forall k \in \{0, \dots, n\}$, $p_k = ka$.

Donc la propriété "$p_k = ka$" est bien un invariant de boucle. Or l'algorithme se termine après la b-ème itération. Donc l'algorithme renverra ba.

Remarque :
Les preuves de correction se font souvent avec des raisonnements par récurrence (récurrence simple, double, voir forte, en fonction de la complexification de l'algorithme, du nombres de boucles imbriquées, etc...).

Exercice 4 :
On reprend l'algorithme

```
a=A
b=B
while a != b :
    if a>b :
        a = a-b
    else :
        b = b-a
print(a)
```

On pose a_0 la valeur initiale de `a` (donc $a_0 = A$) et b_0 la valeur initiale de `b` (donc $b_0 = B$). On note N le nombre d'itérations de la boucle `while`. On note $\forall k \in \{0, \dots, N\}$, a_k et b_k les valeurs des variables `a` et `b` à la fin de la k-ème itération respectivement.

Montrer que la propriété : "$a_k \wedge b_k = a \wedge b$" est in invariant de boucle. En déduire ce que fait cet algorithme.

2 Complexité

La complexité est une forme de réponse à la dernière question relative au bon fonctionnement d'un algorithme : le temps.

Aucune réponse idéal ne peut être donné car le temps pris par un algorithme pour compiler dépend de beaucoup trop de paramètres hors de contrôle. Par exemple, si l'ordinateur fait des tâches en fond (des mises à jours, par exemple), une partie de ses ressources sont alors détournées vers ces tâches de fond et vont donc ralentir *de facto* la vitesse d'exécution de l'algorithme.

Mais la notion de vitesse de calcul est en plus directement liée à la puissance brute de la machine, et donc pas à l'algorithme qu'on exécute. Les smartphones de nos jours sont beaucoup plus puissants que les super-calculateurs de la taille d'un hangar des années 1970. Pourtant, certains algorithmes sont les mêmes aujourd'hui qu'à l'époque. Il s'exécute donc beaucoup plus vite aujourd'hui qu'en 1970.

Or on voudrait une information relative à l'algorithme en lui même, indépendant des caractéristiques techniques de la machine sur lequel on le lance. Pour se faire, on va donner des ordres de grandeur du temps pris par l'algorithme en fonction de ses paramètres. Ces ordres de grandeurs donnent alors une idée de la vitesse *relative* d'exécution de ces algorithmes, vitesse à moduler en fonction des caractéristiques au moment de l'exécution.

2.1 Principe général

Prenons un exemple pour présenter la complexité. On va essayer de déterminer la liste des diviseurs d'un entier naturel. On considère l'algorithme

```
def DiviseursNaif(n) :
    for i in range(1,n+1) :
        if n%i==0 :
            print(i)
```

Comptons le nombre de calculs faits. Pour chaque tour dans la boucle `for`, l'ordinateur effectue une division euclidienne, une comparaison et éventuellement un affichage. Comme il y a n passages pour la boucle `for`, l'algorithme fait donc, environ, $3n$ tâches.

Considérons maintenant l'algorithme suivant

```
1 import numpy as np
2 def DiviseursRapide(n) :
3     for i in range(1,int(np.sqrt(n)+1)) :
4         if n%i==0 :
5             print(i)
6             if n//i!=i :     # rappel : n//i quotient de la division euclidienne
7                 print(n//i)
```

Cet algorithme fait donc $\lfloor \sqrt{n} \rfloor$ boucles et dans chacune, une division euclidienne et une comparaison suivie d'au plus un affichage, une division euclidienne, une comparaison et éventuellement encore un affichage. On a donc au pire, $6 \lfloor \sqrt{n} \rfloor$ calculs lors de l'exécution de cet algorithme.

En prenant $n = 100000$, on a donc 300000 calculs pour le premier algorithme et 1896 calculs seulement pour le second. Ce qui sera beaucoup plus rapide.

Remarque :
En général, un algorithme avec une bonne complexité (donc faible) est souvent beaucoup plus difficile à coder. Ça marche en sens inverse. Puisqu'il faut faire attention aux nombres de calculs que l'on fait et essayer d'en faire le moins possible pour avoir une bonne complexité, il faut donc faire plus attention et c'est donc plus difficile à coder.

On mesure la complexité d'un algorithme par rapport à la taille de l'entrée, *i.e.* par rapport au nombre de chiffres qui le composent.

2.2 Complexité et temps de calcul

La complexité mesure de combien l'algorithme est compliqué en terme de nombre d'opérations. Et bien sûr, le nombre d'opérations à faire est lié au temps de calcul. La complexité est donc, en quelque sorte, une mesure du temps de calculs de l'ordinateur pour un algorithme donné.

2.2.1 Détermination du coût d'un algorithme

Définition 2.1 (Complexité d'un algorithme) :
La complexité d'un algorithme correspond à l'ordre de grandeur du nombre maximum de calculs fait dans l'algorithme. On ne considère donc que le pire des cas de l'algorithme, en terme de nombre de calculs. C'est un ordre de grandeur et est donc invariant par multiplication par une constante. On utilise la notation de Landau O.

Définition 2.2 (Suite dominée) :
Si (u_n) et (v_n) sont deux suites et si $\exists n_0 \in \mathbb{N}$ tel que $\forall n \geq n_0$, $v_n \neq 0$, on dit que (u_n) est dominée par (v_n) si

$$\exists M \geq 0,\ \forall n \geq n_0,\ \left|\frac{u_n}{v_n}\right| \leq M.$$

On notera alors $u_n \underset{n\to\infty}{=} O(v_n)$ et se liera "(u_n) est un grand O de (v_n)".
(Pour plus de détails, voir le chapitre sur les suites en Maths)

Exemple 2.1 :
Montrer que $7n^2 - 25n^3 + 3n + 100 \underset{n\to+\infty}{=} O(n^3)$.

Remarque (Problème de définition d'une tâche élémentaire) :
La notion fondamentale ici est donc la définition d'une *opération élémentaire informatique*. Cette définition n'est pas très claire et dépend un peu de la précision avec laquelle on veut donner une complexité. Plus précisément, Python cache un peu la façon dont il est codé. Par exemple, faire x^4 peut prendre un nombre distinct d'opération, selon comment on regarde cette opération.

De plus, selon comment le langage a été créé, il pourrait faire des sauvegarde momentané de calculs effectué pour pouvoir les réutiliser plus tard et donc réduire le nombre de calculs à faire, donc gagner du temps.

Toutes ces manipulations là sont invisibles. Par conséquent, pour calculer la complexité d'un algorithme, on a besoin de connaître un ordre de grandeur du nombre de tâches effectués, que l'on ne peut pas connaître avec exactitude.

L'une des conséquences est que la méthode utilisé pour compter le nombre de tâches peut différer un peu d'un informaticien à un autre, ou d'un livre à un autre, selon la précision et l'exactitude cherchée par l'auteur.

Comme nous ne sommes dans le cadre d'étude poussée en informatique, on gardera un point de vue un peu vague pour se simplifier la vie. Le système de décompte des tâches effectuées par un algorithme présenté ici et tout à fait discutable et soumis à critiques. Il me semble (c'est une analyse personnelle) qu'il suffit pour les besoins demandés.

Proposition 2.1 (Règles de calcul de la complexité) :
La mesure de la complexité est régie par les règles suivantes :

- **Unité de mesure de base** : comparaisons, affectations, évaluations, affichage, sélection d'un élément (dans un itérable, un n-uplet, une liste, une chaîne ...), lecture, écriture (par caractère). Toute tâche qui ne peut être redécoupée en sous-tâche sera considérée comme une tâche élémentaire.
- **Structure `if`** : le coût d'une structure `if` est inférieur ou égal à la somme des coûts de la condition et du maximum des coûts des deux instructions, *i.e.* pour le code

```
if b :
    p
else :
    q
```

  la complexité est inférieure ou égale à la somme de la complexité de `b` et du maximum de la complexité de `p` et de `q`, *i.e.* la complexité est $C(b) + \max(C(p), C(q))$, en notant C la complexité d'une instruction.
- **Boucle `for`** : Le coût total d'une boucle `for` est la somme des coûts des itérations du corps de la boucle. Dans le cas où le coût du corps ne dépend pas du numéro de l'itération, le coût de la boucle est simplement le coût du corps de la boucle multiplié par le nombre d'itérations, *i.e.*

```
for i in range(n,m) :
    p
```

  le coût de cette boucle est en général $m - n - 1$ fois le coût de `p`. Sinon, si le coût de`p` dépend de `i`, c'est la somme des coûts de `p`, *i.e.* la complexité est $(m - n + 1)C(p)$ (où il faut étudier ce qu'est $C(p)$, qui peut être une structure if/else).
- **Boucle `while`** : Le coût d'une boucle `while` est la somme du coût de la condition et du coût du corps de la boucle, le tout multiplié par le nombre de fois où la boucle a été répétée. La difficulté ici est donc de trouver le nombre de fois où la boucle est parcourue. Mais on peut se contenter d'un majorant du nombre d'itérations. Attention à ne pas oublier la vérification de la condition lors de la sorte de la boucle while. Autrement dit, pour une instruction de la forme

```
while b :
    p
```

  la complexité sera de $N(C(b) + C(p)) + C(b)$, où N est le nombre de fois où l'on va parcourir la boucle.

Exemple 2.2 :
Pour l'algorithme suivant de calcul de factorielle,

```
def factorielle(n) :
    fact=1
    i=1
    while i<=n :
        fact=fact*i
        i=i+1
    return(fact)
```

le coût total est de $5n + 4$, et donc la complexité est en $O(n)$.

Remarque (Notations de Landau) :
En utilisant la notation de Landau, on perd beaucoup d'informations. Avec cette notation, un algorithme mettant $n \times 10^8$ calculs est considéré aussi rapide qu'un algorithme mettant $n \times 10^{-8}$ calculs. Bien sûr, ce n'est pas le cas, si on est parfaitement rigoureux. Mais la différence de temps de calcul entre les deux algorithmes pour de grandes valeurs de n est négligeable devant la différence de temps que met un algorithme en $O(n^2)$ par exemple.

Il faut se rappeler que l'intérêt de la complexité est de donner un ordre de grandeur du temps mis pour de grandes valeurs des paramètres. Il n'est pas nécessaire d'avoir un temps précis.

Exercice 5 :
Déterminer les complexités des algorithmes suivants :

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

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

def table3(n) :
    for i in range(n) :
        for j in range(n) :
            print(i*j)
```

Remarque (Problèmes avec Python) :
Pour connaître la complexité d'un algorithme, il faut connaître le coût temporel de chaque opération. Ici, on se fixe un coût arbitrairement de 1 pour les opérations de bases listées ci-dessus. Mais cette convention est erronée. En réalité, faire un multiplication ne nécessite pas les mêmes ressources que faire une simple addition. Et une multiplication diffère beaucoup aussi de la sélectionner d'un élément dans une liste. C'est un des problèmes de Python. Si nous voulions être tout à fait rigoureux, il faudrait étudier comment chacune des opérations de bases ont été codées pour comparer leur complexités réelles. Mais ce serait trop compliqué. On va donc se contenter de cette approximation, consistant à niveler toutes les opérations et

les mettre au même niveau de complexité (ce qui ferait bondir certaines informaticiens théoricien un peu méticuleux).

Toutefois, une approche de ce problème est proposé en exercice en étudiant un peu plus profondément l'opération puissance. On voit alors qu'elle ne correspond pas tout à fait à des multiplications successives. Elle a été codé pour améliorer sa complexité. Idem pour la multiplication par rapport à l'addition.

2.3 Lien entre complexité et temps de calcul

A titre indicatif, on peut donner un tableau donnant le temps de calculs à un ordinateur pour réaliser un algorithme de différents ordres de grandeur. On va considérer un ordinateur faisant une opération toutes les 10 ns.

Temps	Type de complexité	$n = 10$	$n = 10^3$	$n = 10^4$	$n = 10^6$	Exemple de problème
$O(1)$	Constant	10ns	10ns	10ns	10ns	Accès tableaux
$O(\ln(n))$	Logarithmique	10ns	30ns	40ns	60ns	Recherche dichotomique
$O(n)$	Linéaire	32ns	$10\mu\text{s}$	$100\mu\text{s}$	10ms	Parcours de liste
$O(n^2)$	Quadratique	$1\mu\text{s}$	10ms	1s	2.8h	Parcours de 2 tableaux
$O(n^3)$	Cubique	$10\mu\text{s}$	10s	2.7h	316ans	Multiplication matricielle non optimisée
$O(2^{\ln(n)})$	Sous-exponentielle	100ns	10s	3.2ans	10^{20}ans	Décomposition en facteur premier

Le but étant donc de faire des algorithmes raisonnables en terme de temps de calculs. Il est clair qu'un algorithme nécessitant 316 ans de temps de calcul n'est pas un algorithme raisonnable...

Bien sûr, ces temps de calculs ne sont qu'à titre indicatif. Ils dépendent très fortement de la puissance de calcul de l'ordinateur. Plus la technologie se développe, plus les ordinateurs calculent vite et plus ces temps sont amenés à réduire. Mais on voit bien que si un algorithme met 10^{20} ans à être calculé, la puissance de l'ordinateur aura du mal à le rendre raisonnable.

Remarque (Difficulté d'améliorer la complexité) :
Un bon algorithme est donc un algorithme avec une faible complexité. Pour qu'il soit efficace et aille vite. Mais il y aura une sorte d'équilibre. On pourra soit faire un algorithme simple, naïf et facile coder mais une très mauvaise complexité ; soit améliorer la complexité mais l'algorithme sera beaucoup plus difficile à coder.

Pour améliorer la complexité d'un algorithme, il faut être conscient de toutes les tâches que l'on demande à l'ordinateur et étudier la nécessité de chacune de ces tâches pour éliminer les superflues. Il faut essayer

aussi d'être malin pour imbriquer le moins possibles de boucles. Améliorer la complexité va donc nécessité une étude plus approfondie de ce que l'on demande à l'ordinateur. Ce sera donc très coûteux en terme énergétique pour nous.

Le but étant surtout de trouver une sorte d'équilibre entre un algorithme naïf facile à coder mais qui ne tourne pas en des temps raisonnable et un algorithme difficile à coder nécessitant de longues heures d'études, redoutablement efficace. Il faudra, dorénavant, avoir un peu de recul sur ce qui est codé pour essayer de garder une complexité acceptable, autrement dit d'améliorer légèrement les algorithmes trop naïfs.

2.4 Autres types de complexités

Il existe d'autres types de complexité. Nous avons utilisé la complexité temporelle dans le pire des cas, en prenant en compte toujours le pire des scénarios en terme de nombre de tâches effectués. On a donc, en fait, un ordre de grandeur majorant le nombre de tâches faites. Mais il peut arriver que le pire des scénarios ne soient pas facile à déterminer. Il peut être alors plus facile de déterminer le nombre minimum de tâches. On obtiendrait alors un ordre de grandeur minorant le nombre de tâches de l'aglrithme.

Il existe aussi des complexités en mémoire (on en espace). Chaque algorithme utilise des variables locales. Ces variables prennent de l'espace de stockage. On peut imaginer une situation où l'alglorithem est relativement simple, avec peu de tâche directe à faire. Mais la taille des données sur lesquelles on va appliquer l'algorithme serait énorme (c'est le cas pour les calculs de trajectoires d'un voyage dans l'espace). La problématique alors qui se pose n'est plus le nombre de tâche à faire, mais la taille des données à manipuler. Le temps pris ne repose plus tellement sur le nombre de tâches à faire, mais plutôt sur l'accession aux données, voir la place prises par ces données dans un espace de stockage qui serait limité. On parle alors de complexité en mémoire. Le principe est le même, mais au lieu de compter une tâche élémentaire, on compte le nombre de "place élémentaire" prise par chaque variable.

2.5 Profiling : détermination empirique de la complexité en temps

Il existe différents moyens intégrés à Python pour connaître le temps de calcul d'un algorithme. Le plus simple est la fonction `time()` du module `time`. C'est un outil rudimentaire mais efficace. La fonction`time()` permet d'avoir l'heure de l'horloge interne du processeur de l'ordinateur en secondes. Pour mesurer le temps mis pour effectuer un algorithme, il faut donc faire la différence entre le temps au début de l'algorithme et à la fin.

```python
import math as mt
from time import time
def Premier(n) :
    t1=time()    # Heure de l'horloge interne de l'ordinateur au début de l'algo
    prem=True
    for i in range(2,int(mt.qrt(n)+1)) :
        if n%i==0 :
            prem=False
            t2=time()    # Heure de l'ordinateur à la fin de l'algorithme
            print("Temps de calcul : ", t2-t1)
            return(prem)
    t2=time()    # Heure à la fin de l'algorithme
    print("Temps de calul : ", t2-t1)
    return(prem)
```