CH IX : Algorithmique

Ce chapitre aborde la question l'analyse des algorithmes, à travers la preuve de leur correction et leur complexité. On développe un cadre formel permettant d'établir des éléments théoriques sur les algorithmes, illustré par de nombreux exemples.

I. Spécifier un algorithme

I.1. Un peu de vocabulaire

Définition (Signature, spécification)
Établir la **signature** d'un algorithme ou d'une fonction, c'est :

a) le nommer (lui donner un nom),

b) préciser ses arguments (ou paramètres) et leur type,

c) préciser ses sorties (valeurs renvoyées par l'algorithme ou la fonction) et leur type.

Spécifier un algorithme consiste à :

1) donner sa signature,

2) préciser les **pré-conditions** (conditions préalables) que ses arguments doivent satisfaire,

3) préciser les **post-conditions** portant sur les valeurs renvoyées.

Remarque

- Spécifier un algorithme ne donne aucune information sur son déroulement : il en fixe uniquement le cadre. Une spécification est comme une consigne donnée par un « client » à un développeur : « je vous demande de bien vouloir programmer une fonction qui prenne en entrée deux entiers a, b tels que $a > b \geqslant 1$ et qui renvoie un entier égal au PGCD des entiers a, b ».
- Une telle demande correspond exactement à la spécification suivante :
  1) signature : $(\texttt{a} : \texttt{int}, \texttt{b} : \texttt{int}) \longrightarrow (\texttt{d} : \texttt{int})$,
  2) pré-conditions : $\texttt{a} > \texttt{b}$ et $\texttt{b} \geqslant 1$,
  3) post-conditions : $\texttt{d} = \texttt{a} \wedge \texttt{b}$.

Exemples
Voici quelques exemples supplémentaires :

- une fonction prenant en argument un triplet d'entiers $(\texttt{a}, \texttt{b}, \texttt{c})$ et renvoyant le couple des racines du trinôme $aX^2 + bX + c$ si son discriminant est strictement positif :
  1) signature : $\texttt{Nom\_de\_fonction1}(\texttt{a} : \texttt{int}, \texttt{b} : \texttt{int}, \texttt{c} : \texttt{int}) \longrightarrow (\texttt{x1} : \texttt{float}, \texttt{x2} : \texttt{float})$,
  2) pré-conditions : $\texttt{b}^2 - 4\texttt{ac} > 0$,
  3) post-conditions : $\texttt{a}\,\texttt{x1}^2 + \texttt{b}\,\texttt{x1} + \texttt{c} = 0$ et $\texttt{a}\,\texttt{x2}^2 + \texttt{b}\,\texttt{x2} + \texttt{c} = 0$.
- une fonction prenant en argument deux entiers $\texttt{a}$, $\texttt{b}$ vérifiant $\texttt{a} \geqslant \texttt{b} \geqslant 1$ et renvoyant un triplet $(\texttt{d}, \texttt{u}, \texttt{v})$ où $\texttt{d}$ est le PGCD de $\texttt{a}$, $\texttt{b}$ et $\texttt{u}$, $\texttt{v}$ tels que $\texttt{a}\,\texttt{u} + \texttt{b}\,\texttt{v} = \texttt{d}$.
  1) signature : $\texttt{Nom\_de\_fonction2}(\texttt{a} : \texttt{int}, \texttt{b} : \texttt{int}) \longrightarrow (\texttt{d} : \texttt{int}, \texttt{u} : \texttt{int}, \texttt{v} : \texttt{int})$,
  2) pré-conditions : $\texttt{a} \geqslant \texttt{b}$ et $\texttt{b} \geqslant 1$,
  3) post-conditions : $\texttt{d} = \texttt{a} \wedge \texttt{b}$ et $\texttt{a}\,\texttt{u} + \texttt{b}\,\texttt{v} = \texttt{d}$.

Une bonne pratique
Il est vivement recommandé de préciser spécification (signature, pré-conditions et post-conditions) dans chaque programme, en commentaire. Cette tâche peut être remplacée par l'écriture d'un texte préalable au programme proposé.

I.2. Vérification des pré-conditions et post-conditions : instruction `assert`

La vérification d'une pré-condition peut s'effectuer en utilisant **l'instruction** `assert` **suivie d'une condition** :

```
assert condition
```

La condition est un booléen : si sa valeur est `True`, l'exécution du code se poursuit. Sinon, une erreur est levée, le programme s'interrompt avec `Assertion Error`.

Sur l'exemple des racines d'un trinôme à coefficients entiers, évoqué plus haut :

```
import numpy as np
def racines(a,b,c) :
    '''
    signature : (a : int, b : int, c : int)
    -> (x1 : float, x2 : float)
    pré-condition : b**2 - 4*a*c > 0
    post-conditions : x1 et x2 racines de aX**2 + bX + c
    '''
    # Vérification de la pré-condition
    assert(b**2 - 4*a*c > 0)
    delta = np.sqrt(b**2 - 4*a*c)
    x1 = (-b + delta) / (2*a)
    x2 = (-b - delta) / (2*a)
    return x1, x2
```

Vérifier le type d'une variable (hors programme)
En plus de vérifier les pré-conditions, il est possible de vérifier le type des variables passées en entrée, toujours à l'aide de l'instruction `assert`.
Python pratique un typage dynamique des variables, c'est-à-dire que leur type est déterminé au moment où on leur affecte une valeur. Toutefois, il est possible de vérifier le type d'une variable en utilisant la fonction `isinstance`.

Exemple
Reprenons le code ci-dessus :

```
def racines(a,b,c) :
    '''
    signature : (a : int, b : int, c : int)
    -> (x1 : float, x2 : float)
    pré-condition : b**2 - 4*a*c > 0
    post-conditions : x1 et x2 racines de aX**2 + bX + c
    '''
    # Vérification du type des variables
    assert isinstance(a, int)
    assert isinstance(b, int)
    assert isinstance(c, int)
    # Vérification de la pré-condition
    assert(b**2 - 4*a*c > 0)
    delta = np.sqrt(b**2 - 4*a*c)
    x1 = (-b + delta) / (2*a)
    x2 = (-b - delta) / (2*a)
    return x1, x2
```

L'instruction `assert` permet aussi de vérifier des post-conditions. **Attention toutefois à la comparaison de flottants :** dans le chapitre consacré à la représentation des nombres, nous verrons qu'il est possible qu'une somme mathématiquement égale à 1 n'apparaisse pas comme telle dans **Python**. Dans le code ci-dessus, il est possible (et même fort probable) que les grandeurs `a * x1**2 + b * x1 + c` et `a * x2**2 + b * x2 + c` ne renvoient pas exactement `0.0`.

II. Correction d'un algorithme

II.1. Problématique

Prouver qu'un algorithme se termine, et qu'il se termine bien, n'est pas chose toujours aisée.

Dans le cas de boucles `for` dont l'itérateur (indice, élément contenu dans une liste, etc.) n'est pas modifié, la terminaison arrive à coup sûr et au bout d'un nombre prévisible d'étapes.

Exemples
C'est le cas dans les exemples suivants.

- **Calcul d'une somme ou d'un produit** :

```
S = 0
P = 1
for k in range(n) :
    S = S + k
    P = P * k
print(S,P)
```

- **Construction d'une liste** :

```
# Cet algorithme construit la liste des n premiers carrés d'entiers
L = [0]*n
for k in range(n) :
    L[k] = k**2
print(L)
```

- **Conjonction de booléens** :

```
# Cet algorithme teste si tous les éléments de L sont nuls
t = True
for e in range(L) :
    t == t and e==0
print(t)
```

- **Disjonction de booléens** :

```
# Cet algorithme teste si la liste L contient au moins un multiple de 7
t = False
for e in range(L) :
    t == t or (e%7 == 0)
print(t)
```

- **Calcul des termes d'une suite récurrente** :

```
# Cet algorithme calcule les n premiers termes de la suite de Fibonacci
L = [0]*n
L[0] = 1
L[1] = 1
for k in range(n-2) :
    L[k+2] = L[k] + L[k+1]
print(L)
```

Malgré tout, il subsiste des exemples pour lesquels la terminaison reste non démontrée. Un des plus connus reste la suite de Syracuse, définie par le processus suivant :

$$\begin{cases} u_0 = a \in \mathbb{N}^* \\ \forall n \in \mathbb{N},\ u_{n+1} = \begin{cases} \dfrac{u_n}{2} & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{sinon} \end{cases} \end{cases}$$

Empiriquement, cette suite semble présenter une périodicité 4, 2, 1, 4, 2, 1, 4, 2, 1, ..., à partir d'un certain rang, mais personne n'a encore réussi à le montrer à ce jour.

II.2. Quelques définitions

Définition (Algorithme correct / partiellement correct)

1. Un algorithme est dit **partiellement correct** s'il est conforme à sa spécification *sous l'hypothèse qu'il termine.*
2. Un algorithme est **correct** s'il termine et s'il est conforme à sa spécification.

Remarque
Pour résumer :

partiellement correct + terminaison = correct

Définition (Variant de boucle)
Un **variant de boucle** d'un algorithme est une quantité :

1) positive lorsque la condition de la boucle est vérifiée,

2) qui dépend des variables en jeu dans l'algorithme,

3) qui décroît strictement à chaque itération.

Le suivi des variants de boucle permet parfois d'établir la terminaison d'un algorithme.

Théorème 1.
Si une boucle admet un variant alors elle se termine nécessairement.

Définition (Invariant de boucle)
Un **invariant de boucle** d'un algorithme est une proposition logique $\mathcal{P}$:

1) portant sur les variables de l'algorithme,

2) qui est vraie avant la boucle concernée,

3) qui reste vraie durant toute la boucle, après chaque itération.

Les invariants de boucle sont des outils utiles pour prouver le caractère correct d'un algorithme.

II.3. Exemples

II.3.a) Exemple 1 : Algorithme d'Euclide

Spécification de l'algorithme

On reprend la fonction, reprenant le calcul du PGCD de deux entiers. Sa spécification est la suivante :

1) signature : `Euclide(a : int, b : int)` $\longrightarrow$ `int`,

2) pré-condition : `a` et `b` sont des entiers positifs ou nuls,

3) post-conditions : la sortie est égale au PGCD des entiers `a`, `b`.

```
1  def Euclide(a, b) :
2      if a < b :
3          return Euclide(b, a)
4      else :
5          d, r = a, b
6          while r > 0 :
7              aux = d
8              d = r
9              r = aux % d
10         return d
```

Correction de l'algorithme

- On s'appuie sur :
  - × un premier invariant de boucle : l'entier `r` est un entier positif ou nul à travers tout l'algorithme, puisque c'est un reste dans une division euclidienne,
  - × un variant de boucle : l'entier `r` décroît strictement tant qu'il n'est pas nul, puisqu'on le remplace par un reste modulo lui-même,
  - × un second invariant de boucle : le PGCD entre `d` et `r` reste le même à travers la boucle `while`.
- La variable `r` définit une suite d'entiers naturels strictement décroissante. Elle est donc nulle à partir d'un certain rang. Cela assure la terminaison de l'algorithme.
- D'après le second invariant de boucle, à tout moment de la boucle, le PGCD de `d` et `r` est le PGCD de `a` et `b`. Or, lorsque la boucle s'arrête, `r` est nul et `d` contient donc bien le PGCD de `a` et `b`.

En définitive, l'algorithme est donc partiellement correct et termine. Il est donc correct.

II.3.b) Exemple 2 : tri fusion

On rappelle le principe de tri fusion.
On souhaite trier par ordre croissant les éléments d'une liste d'entiers. Étant donnée une liste L, on procède comme suit.

1) **Diviser** : la liste L est séparée en 2 (au milieu), à chaque appel.

2) **Régner** : chaque sous-liste est alors triée récursivement en utilisant le principe de découpage précédent. Cas initial : une liste qui ne contient qu'un élément est triée.

3) **Combiner** : les deux sous-listes triées sont alors combinées de sorte à ce que la liste résultat soit triée. Cette étape est appelée **fusion**.

On suppose implémentée une fonction `fusion` prenant en argument deux listes triées d'entiers `L1` et `L2`, et renvoyant la liste triée résultat de la fusion des deux listes.
On pourra se référer au TP6 pour plus de détails.

La fonction `tri_fusion` ci-dessous prend en paramètre une liste `L`, trie la liste selon le principe du tri fusion et renvoie la liste obtenue.

```
def tri_fusion(L) :
    n = len(L)
    # Terminaison : L vide ou à un élément
    if n <= 1 :
        return L
    else :
        m = n // 2
        return fusion(tri_fusion(L[:m]), tri_fusion(L[m:]))
```

- Il s'agit d'un algorithme récursif, il faut en tenir compte dans la preuve de sa correction.

Spécification de l'algorithme

La spécification de l'algorithme de tri fusion est la suivante :

1) signature : `tri_fusion(L : list(int))` $\longrightarrow$ `list(int)`,

2) pré-conditions : `L` est une liste d'entiers,

3) post-conditions : la liste renvoyée est une liste d'entiers triée par ordre croissant.

Correction de l'algorithme

- La boucle ici est invisible : il s'agit de l'enchaînement des appels récursifs censés mener à une liste triée. On identifie deux éléments caractéristique :
  - × un variant de boucle : la taille de la liste triée dans les appels récursifs est strictement décroissante tant qu'elle n'est pas égale à 1 ou 0,
  - × un invariant de boucle : la liste renvoyée par la ligne `fusionne(triFusion(L1),triFusion(L2))` est triée.
- La taille de la liste triée dans les appels récursifs définit une suite d'enters naturels strictement décroissante. En effet, la liste `L[:m]` est de taille `n//2` et la liste `L[m:]` de taille `n - n//2`, toutes deux de taille strictement inférieure à `n`. La taille de la liste triée atteint donc 0 ou 1 qui sont les cas de terminaison de l'algorithme. Cela assure la terminaison de l'algorithme.
- Par définition de la fonction `fusion`, les listes manipulées sont toujours triées. L'invariant de boucle assure donc la correction partielle de l'algorithme.

L'algorithme est donc partiellement correct et se termine. Il est donc correct.

II.3.c) Exemple 3 : Divisibilité par 7

On étudie l'algorithme suivant, à appliquer à un entier naturel non nul n.

- Tant que $n \geqslant 100$:
  - × on enlève le chiffre des unités u de n et on note q le résultat,
  - × on enlève deux fois ce chiffre des unités à n,
  - × on recommence avec $n = q - 2 \times u$.
- lorsque $n < 99$, on renvoie n.

L'utilisateur est alors à-même de voir si l'entier n entré initialement est multiple de 7 ou non : il l'est si et seulement si le résultat retourné est un multiple de 7.

On obtient le script suivant :

```
1  def divisibleParSept(n) :
2      N = n
3      while N >= 100 :
4          (q,u) = (N//10, N%10)
5          N = q - 2 * u
6      return N
```

Spécification de l'algorithme

La spécification de cet algorithme est la suivante :

1) signature : `divisibleParSept(n : int)` $\longrightarrow$ `int`,

2) pré-condition : `n` est un entier non nul,

3) post-condition : `N` est un entier multiple de 7 si et seulement si `n` est multiple de 7.

Correction de l'algorithme

- On définit les variant et invariant suivants :
  - × un variant de boucle égal à l'entier `N`, qui décroît strictement à chaque itération de la boucle `while`,
  - × un invariant de boucle qui est le booléen « `(N%7 == 0) == (n%7 == 0)` ».
- La variable `N` définit une suite d'entiers naturels strictement décroissante. Elle est donc nulle à partir d'un certain rang. Cela assure la terminaison de l'algorithme.
- L'invariant de boucle est vrai lorsque `N` est multiple de 7 si et seulement si `n` l'est (et faux sinon). Pour le prouver, il suffit de démontrer :

$$7 \mid \mathtt{N} \quad \Leftrightarrow \quad 7 \mid (\mathtt{q} - 2\,\mathtt{u})$$

Or, par définition de `q` et `u` : $\mathtt{N} = 10\,\mathtt{q} + \mathtt{u}$. Ainsi : $\mathtt{N} = 3\,\mathtt{q} + \mathtt{u} + 7\,\mathtt{q}$. Donc :

$$\mathtt{N} \equiv 3\,\mathtt{q} + \mathtt{u}\ [7]$$

On doit donc démontrer :

$$7 \mid (3\,\mathtt{q} + \mathtt{u}) \quad \Leftrightarrow \quad 7 \mid (\mathtt{q} - 2\,\mathtt{u})$$

On procède par double implication.

($\Leftarrow$) Supposons : $7 \mid (\mathtt{q} - 2\,\mathtt{u})$.
Alors il existe $k \in \mathbb{Z}$ tel que : $\mathtt{q} = 2\,\mathtt{u} + 7k$. D'où :

$$3\,\mathtt{q} + \mathtt{u} \ = \ 3\,(2\,\mathtt{u} + 7k) + \mathtt{u} \ = \ 7\,(\mathtt{u} + 3\,k)$$

Or : $\mathtt{u} + 3\,k \in \mathbb{Z}$. On en déduit : $7 \mid (3\,\mathtt{q} + \mathtt{u})$.

($\Rightarrow$) Supposons : $7 \mid (3\,\mathtt{q} + \mathtt{u})$.
Alors il existe $k \in \mathbb{Z}$ tel que : $\mathtt{u} = -3\,\mathtt{q} + 7k$. D'où :

$$\mathtt{q} - 2\,\mathtt{u} \ = \ \mathtt{q} - 2\,(-3\,\mathtt{q} + 7k) \ = \ 7\,(\mathtt{q} - 2\,k)$$

Or : $\mathtt{q} - 2k \in \mathbb{Z}$. On en déduit : $7 \mid (\mathtt{q} - 2\,\mathtt{u})$.

Ainsi, un entier `N` qui est multiple de 7 si et seulement si l'entier `n` l'était. L'algorithme est donc partiellement correct.

L'algorithme est donc partiellement correct et se termine. Cela démontre sa correction.

II.3.d) Exemple 4 : suite de Syracuse

Reprenons la suite de Syracuse $(u_n)_n$ et considérons la fonction ci-dessous, dont l'objectif est d'évaluer le *temps de vol* de la suite de Syracuse, c'est-à-dire le plus petit indice i tel que $u_i = 1$. Revenue à 1, on sait en effet que la suite de Syracuse se retrouve alors piégée dans un schéma périodique 1, 4, 2, 1, 4, 2, 1, 4, 2, ...

```
1  def Syracuse(u0) :
2      '''
3      signature : (u0 : int) -> (i : int)
4      pré-condition : u0 > 0
5      '''
6      assert(u0 > 0)
7
8      u = u0
9      i = 0
10     while u != 1 :
11         if u % 2 == 0 :
12             u = u//2
13         else :
14             u = 3 * u + 1
15         i += 1
16     return i
```

Spécification de l'algorithme

La spécification de l'algorithme est la suivante :

1) signature : `Syracuse(u0 : int)` $\longrightarrow$ `(i : int)`,

2) pré-conditions : `u0` doit être un entier strictement positif,

3) post-conditions : l'entier `i` est le plus petit entier i tel que $u_i = 1$.

Correction partielle de l'algorithme

- Soit $i \in \mathbb{N}$. On note $\mathtt{u}(i)$ la valeur de la variable `u` à l'itération i.
- On peut définir un invariant de boucle en posant :

$$\mathcal{P} : \mathtt{u}(i) \text{ est le } i^{\text{ème}} \text{ terme de la suite de Syracuse}$$

- Pour tout $i \in \mathbb{N}$, à l'itération i, la variable `u` contient le $i^{\text{ème}}$ terme de la suite de Syracuse. L'indice i s'incrémentant de 1 à chaque itération, lorsque `u` atteint 1, la boucle `while` s'interrompt et la variable i contient alors bien le premier indice i tel que $\mathtt{u}(i) = 1$.

On rappelle qu'on démontre ici seulement une correction partielle car on ne sait pas que l'algorithme se termine (d'ailleurs, personne ne sait encore démontrer qu'il se termine).

III. Complexité : enjeux, définition et calcul

III.1. Aux frontières des possibilités de la machine

Les opérations informatiques sont souvent limitées par trois facteurs indépendants : la mémoire, le temps et la précision.

III.1.a) Pas assez de mémoire

Un exemple.
Une mole d'atomes contient $6,02 \cdot 10^{23}$ atomes. Un volume d'un litre de gaz aux CSTP (conditions standards de température et de pression) contient environ $\dfrac{1}{24}$ mole.

Imaginons qu'on souhaite modéliser, en mettant éventuellement à profit la totalité des ordinateurs disponibles sur Terre, les mouvements de chaque particule d'un tel volume.

- Il faut alors commencer par stocker les positions (x_i, y_i, z_i) et les vecteurs vitesses initiaux (vx_i, vy_i, vz_i) de chaque particule i.
- Un flottant est codé sur 64 bits (en norme IEEE 754 mais ce point de détail sort du cadre du programme). Il faut donc un espace mémoire égal à :

$$\frac{1}{24} \times 6,02 \times 10^{23} \times 6 \times 64 \text{ bits}$$

soit environ $\boxed{1,2 \cdot 10^{24}}$ octets.

Sachant que nous sommes environ 7 milliards sur Terre, il faudrait donc que chacun soit équipé d'un ordinateur de $1.72 \cdot 10^{14}$ octets, soit environ 172 To (Téra-octet). Pourquoi pas. Mais ce n'est que pour les conditions initiales : désormais, il faut aussi de la mémoire vive pour effectuer les calculs. Et là, cela devient compliqué (et ce n'est que pour 1L, à conditions température-pression constantes).

III.1.b) Pas assez de temps

Prenons l'exemple de la constante d'Euler.
On note (u_n) la suite définie, pour tout $n \in \mathbb{N}^*$, par :

$$u_n \ = \ \sum_{k=1}^{n} \frac{1}{k} - \ln(n)$$

On démontrera, à l'aide d'une comparaison série-intégrale que la suite (u_n) converge vers une constante, notée γ, et appelée constante d'Euler-Mascheroni (ou constante d'Euler).
On sait peu de choses de cette constante. En particulier, on ignore si elle est rationnelle ou non (même si on en connaît désormais 400 milliards de décimales ! - record à battre).
Plus précisément, si on pose, pour tout $n \in \mathbb{N}^*$:

$$\begin{cases} u_n = \displaystyle\sum_{k=1}^{n} \frac{1}{k} - \ln(n+1) \\ v_n = \displaystyle\sum_{k=1}^{n} \frac{1}{k} - \ln(n) \end{cases}$$

Alors on obtient un encadrement de γ :

$$\forall n \in \mathbb{N}^*, \ u_n \leqslant \gamma \leqslant v_n$$

Ainsi, il suffit, étant donné un écart $\varepsilon > 0$ fixé, de déterminer le premier n tel que $v_n - u_n \leqslant \varepsilon$ pour disposer (à travers u_n ou v_n) d'une approximation de la constante γ à ε près.

Dans le code suivant, on utilise la méthode `time()` du package éponyme, pour chronométrer les calculs.

```
import time
def EulerApprox(eps) :
    u = 1 - np.log(2)
    v = 1
    n = 1
    while (v-u) > eps :
        n += 1
        u = u + 1/n + np.log(n) - np.log(n+1)
        v = v + 1/n + np.log(n-1) - np.log(n)
    return n

n = 8
L = [0]*n
T = [0]*n
for k in range(n) :
    t0 = time.time()
    L[k] = EulerApprox(10**(-k))
    t1 = time.time()
    T[k] = t1 - t0
print(T)
```

Ceci donne le tableau suivant :

Précision ε	n	u_n	Temps de calcul (s)
10^{-0}	1	$0,3068528$	$1,0 \cdot 10^{-5}$
10^{-1}	10	$0,5310730$	$1,45 \cdot 10^{-5}$
10^{-2}	100	$0,5722570$	$8,01 \cdot 10^{-5}$
10^{-3}	1000	$0,5767160$	$7,66 \cdot 10^{-4}$
10^{-4}	10000	$0,5771656$	$1,11 \cdot 10^{-2}$
10^{-5}	100000	$0,5772106$	$6,38 \cdot 10^{-2}$
10^{-6}	1000000	$0,5772152$	$5,74 \cdot 10^{-1}$
10^{-7}	10000000	$0,5772156$	$5,92$

Dans les grandes lignes, le temps de calcul est multiplié par 10 à chaque décimale supplémentaire exigée. Partant du temps mesuré pour 7 décimales exactes, si on souhaite 20 décimales de γ, il faut disposer de $6 \cdot 10^{13}$ secondes devant soi, c'est-à-dire... près de 2 millions d'années.
Celui qui a obtenu les 400 milliards de décimales n'est pas un dinosaure, il a simplement eu recours à un algorithme plus efficace, *de complexité inférieure*.

III.1.c) La précision

Chaque nombre a une place mémoire qui lui est allouée et le degré de précision dont on dispose est limité par cette mémoire allouée. En d'autres termes, si vous décidez d'allouer de quoi stocker 20 décimales pour chaque flottant, alors la précision dont vous disposerez pour les calculs n'ira pas plus loin que 10^{-20}.
Pour vous libérer de cette contrainte, on peut concevoir étendre la représentation d'un nombre, donc définir un nouveau format. Mais ceci a un coût en termes de mémoire. Ainsi, les limitations en termes de précision sont liées aux limitations en termes de mémoire.
Par curiosité, on pourra lancer le code suivant, censé ne jamais s'arrêter, mais qui termine tout de même.

```
i = 0
while 10**(-i) != 0 :
    i += 1
print(i)
```

La valeur renvoyée est d'ailleurs `i = 324`.

III.2. Définitions et relations de comparaison

III.2.a) Formalisation de la notion de complexité algorithmique

On considère un algorithme A, codé sous forme de fonction ou de procédure. Cet algorithme dépend de données d prises dans un ensemble D (par exemple, d peut être une liste d'entiers et D est alors l'ensemble des listes d'entiers). Ainsi, l'algorithme A est vu comme une fonction qui à chaque d associe un résultat d' appartenant à un ensemble D'.
Chaque jeu de données d est mesurée par une taille, qu'on représente par une fonction :

$$\texttt{taille} : D \longrightarrow \mathbb{N}$$

Par exemple, si d est une liste d'entiers, on prendra la fonction taille qui à d associe $\texttt{len}(d)$.
On peut alors partitionner D suivant la taille de ses objets en posant, pour tout $n \in \mathbb{N}$:

$$D_n \ = \ \{d \in D \mid \texttt{taille}(d) = n\}$$

Définition (Complexité par rapport à certaines opérations fondamentale)
On fixe un ou plusieurs types d'opérations appelées **opérations fondamentales**. Le **coût** de l'algorithme pour chaque $d \in D$ et pour les opérations fondamentales choisies est le nombre $\text{coût}(A(d))$ d'opérations fondamentales nécessaires pour mener à bien le calcul de $A(d)$.
Alors on définit :

- **la complexité de A dans le pire des cas** : $\max\limits_{d \in D_n} \Big(\texttt{coût}\big(A(d)\big)\Big)$.
- **la complexité de A dans le meilleur des cas** : $\max\limits_{d \in D_n} \Big(\texttt{coût}\big(A(d)\big)\Big)$.

Lorsque ces deux complexités ont des ordres de grandeur complètement différents, on introduit également la **complexité moyenne** comme, pour chaque $n \in \mathbb{N}$:

$$\frac{1}{\text{Card}(D_n)} \sum_{d \in D_n} \texttt{coût}\big(A(d)\big)$$

III.2.b) Relations de comparaison

On rappelle les définitions des relations de comparaison asymptotique de suites numériques.

Comparaison des suites
Soient $(u_n)_n$ et $(v_n)_n$ deux suites réelles ne s'annulant plus à partir d'un certain rang.

a) On dit que $(u_n)_n$ est dominée par $(v_n)_n$, et on note $u_n = \underset{n\to+\infty}{O}(v_n)$ si :

$$\boxed{\left(\frac{u_n}{v_n}\right) \text{ bornée}}$$

b) On dit que $(u_n)_n$ est négligeable devant $(v_n)_n$, et on note $u_n = \underset{n\to+\infty}{o}(v_n)$ si :

$$\boxed{\frac{u_n}{v_n} \underset{n\to+\infty}{\longrightarrow} 0}$$

c) On dit que $(u_n)_n$ est équivalent à $(v_n)_n$, et on note $u_n \underset{n\to+\infty}{\sim} v_n$ si :

$$\boxed{\frac{u_n}{v_n} \underset{n\to+\infty}{\longrightarrow} 1}$$

Les évaluations de complexité sont couramment faites en $\underset{n\to+\infty}{O}(\cdot)$, ce qui est une information de domination, donc imparfaite. Par exemple, un algorithme qui a un coût moyen de $4n$ pourra tout à fait être qualifié d'algorithme « en $\underset{n\to+\infty}{O}(n)$ » ou d'algorithme « en $\underset{n\to+\infty}{O}(n^2)$ », voire « en $\underset{n\to+\infty}{O}(n^7)$ », pourquoi pas. Mais, la meilleure estimation étant $\underset{n\to+\infty}{O}(n)$, on retiendra celle-ci.

III.3. Premiers exemples

III.3.a) Algorithme de division euclidienne

Considérons l'algorithme de division euclidienne d'un entier naturel par un entier naturel non nul, par soustractions successives.

```
1  def divEuclidienne(a, b) :
2      q = 0
3      r = a
4      while r >= b :
5          q = q + 1
6          r = a - q * b
7      return q, r
```

L'algorithme s'applique à deux entiers. La fonction de taille est donc la fonction identité de $\mathbb{N}$, c'est-à-dire :

$$\text{taille} : \mathtt{n} \mapsto \mathtt{n}$$

Complexité de l'algorithme
Soit $(\mathtt{a}, \mathtt{b}) \in \mathbb{N} \times \mathbb{N}^*$.
On considère que les opérations élémentaires sont la soustraction, l'addition et le produit.

- On compte, pour `a,b` entiers naturels, un nombre d'opérations élémentaires égal à $3\left\lfloor \frac{a}{b} \right\rfloor$.
- La complexité de cet algorithme est donc en $O\left(\left\lfloor \frac{a}{b} \right\rfloor\right)$.

III.3.b) Recherche dans une liste non triée

Considérons l'algorithme ci-dessous, de recherche d'un élément dans une liste d'éléments de même type (*cf TP4*).

```
def rechercheListeNonTriee(L, x) :
    n = len(L)
    for i in range(n) :
        if L[i] == x :
            return i
```

L'algorithme s'applique à une liste et un élément. La fonction de taille associe à une liste `L` l'entier `len(L)`, c'est-à-dire :

$$\text{taille} : \texttt{L} \mapsto \texttt{len(L)}$$

Complexité de l'algorithme
Soit `L` une liste à `n` éléments.
On considère que l'opération élémentaire est la comparaison (`=`).

- Dans le meilleur des cas, l'élément `x` est le 1$^{\text{er}}$ élément de la liste `L`. Dans ce cas, le boléen `L[0]==x` est vrai et l'algorithme effectue une seule comparaison.
  La complexité dans le meilleur des cas est donc en $\underset{n\to+\infty}{O}(1)$.
- Dans le pire des cas, l'élément `x` est le dernier élément de la liste `L` ou est absent de la liste `L`. Dans ce cas, les comparaisons `L[i] == x` sont effectuées pour tout `i` $\in [\![0, \texttt{n}-1]\!]$. Autrement dit, on effectue `n` comparaisons.
  La complexité dans le pire des cas est donc en $\underset{n\to+\infty}{O}(n)$.
- On peut démontrer que la complexité en moyenne de cet algorithme est également en $\underset{n\to+\infty}{O}(n)$.

III.3.c) Recherche du plus grand élément dans une liste

On rappelle que l'algorithme suivant permet d'obtenir le plus grand élément d'une liste `L` (*cf TP2*).

```
def maxi(L) :
    M = L[0]
    for i in range(1, len(L)) :
        if L[i] > M :
            M = L[i]
    return M
```

La fonction s'applique à une liste. La fonction de taille associe donc à une liste `L` l'entier `len(L)`.

$$\text{taille} : \texttt{L} \mapsto \texttt{len(L)}$$

Complexité de l'algorithme
Soit `L` une liste à `n` éléments.
On considère que l'opération élémentaire est la comparaison.

- Dans tous les cas, le parcours de tous les éléments de la liste `L` est nécessaire pour déterminer son plus grand élément. On effectue donc `n` comparaisons.
- La complexité de cet algorithme est donc en $\underset{n\to+\infty}{O}(n)$.

III.3.d) Recherche d'un entier p tel que $2^p \leqslant n < 2^{p+1}$

On se propose ici d'étudier la complexité de l'algorithme qui, à un entier naturel n, associe la plus grande puissance de 2 qui lui est inférieure ou égale.
C'est l'objet de la fonction `puissanceDeux` ci-dessous :

```
1  def puissanceDeux(n) :
2      p = 0
3      d = 1
4      while d <= n :
5          d = 2 * d
6          p = p + 1
7      return p - 1
```

L'algorithme s'applique à un entier. La fonction de taille est donc la fonction identité de $\mathbb{N}$.

$$\text{taille} : \mathtt{n} \mapsto \mathtt{n}$$

Complexité de l'algorithme
Soit $n \in \mathbb{N}$.
On considère que les opérations élémentaires sont l'addition et le produit.

- Le nombre de tours de boucle est égal au plus petit entier p tel que : $2^p > n$. Or :

$$\begin{aligned} 2^p > n &\Leftrightarrow p\ln(2) > \ln(n) && \textit{(par stricte croissance de } \ln \textit{ sur } \mathbb{R}_+^*\textit{)} \\ &\Leftrightarrow p > \frac{\ln(n)}{\ln(2)} \end{aligned}$$

  Ainsi l'entier p recherché est : $p = \left\lceil \dfrac{\ln(n)}{\ln(2)} \right\rceil$.

- On en déduit que cet algorithme réalise : $2 \times \left\lceil \dfrac{\ln(n)}{\ln(2)} \right\rceil + 1$ opérations fondamentales.
  La complexité de celui-ci est donc en $\underset{n\to+\infty}{O}\big(\ln(n)\big)$.

III.3.e) Tri par sélection

On étudie, dans le même but, le tri par sélection pour des listes d'entiers. On rappelle le principe du tri par sélection (*cf TP6*).
Cet algorithme consiste à :

- sélectionner un élément minimal de la liste,
- échanger cet élément avec le premier élément de la liste,
- recommencer avec les éléments suivants.

On en déduit le script suivant.

```
1  def indice_min(L, i) :
2      n = len(L)
3      i_min = i
4      for j in range(i, n) :
5          if L[j] < L[i_min] :
6              i_min = j
7      return i_min
8
9  def tri_selection(L) :
10     n = len(L)
11     for i in range(n) :
12         p = indice_min(L, i)
13         L[i], L[p] = L[p], L[i]
14     return None
```

La fonction s'applique à une liste. La fonction de taille est donc :

$$\text{taille} : \texttt{L} \mapsto \texttt{len(L)}$$

Complexité de l'algorithme
Soit `L` une liste à n éléments.
On considère que l'opération élémentaire est la comparaison.

- On remarque que :
  - × l'on effectue toujours n étapes (ce sont les n tours de boucle).
  - × À l'étape i on effectue toujours $n-i$ comparaisons pour déterminer le minimum de la liste `L[i:]`.
  - × On effetue alors en tout : $\sum_{i=0}^{n-1}(n-i) = \sum_{k=1}^{n} k = \frac{n(n+1)}{2}$ comparaisons.
- La complexité du tri par sélection est donc en $\underset{n\to+\infty}{O}(n^2)$.

III.4. Exemple : tri par insertion

L'analyse d'un algorithme s'appuie sur le plan suivant :

1. **Spécification de l'algorithme**
2. **Correction de l'algorithme**
3. **Évaluation de la complexité**

À titre d'exemple, on propose ici l'analyse algorithmique complète du tri par insertion (*cf* TP6). On rappelle le principe.

- Initialement, la partie gauche contient 1 élément (elle est donc triée) : la carte la plus à gauche.
- La carte suivante de la main est alors placée soit avant la carte précédente, soit après.
- On procède ainsi de suite de sorte qu'au début de la $i^{\text{ème}}$ étape, les i cartes les plus à gauche sont classées dans l'ordre croissant. On place alors la $i^{\text{ème}}$ carte au bon endroit dans cette partie de gauche classée.
- La dernière étape consiste à placer la carte la plus à droite parmi les $n-1$ cartes les plus à gauches qui sont alors classées correctement.

On obtient le script suivant.

```
1  def tri_insertion(L) :
2      n = len(L)
3      for i in range(1,n):
4          aux = L[i]
5          j = i
6          while (j > 0 and L[j-1] > aux) :
7              L[j] = L[j-1]
8              j = j - 1
9          L[j] = aux
```

Spécification de l'algorithme

La spécification de l'algorithme est la suivante :

1) signature : `tri_insertion(L : list)` $\longrightarrow$ `L : list`,

2) pré-condition : `L` est une liste d'objets pouvant être ordonnés,

3) post-condition : `L` est une liste triée.

Correction de l'algorithme

- La variable `j` définit une suite strictement décroissante d'entiers positifs. En effet, la variable `j` est initialisée à une valeur entière `i` comprise entre 1 et `n` -1 et décroît de 1 à chaque tour de boucle. C'est donc un variant de la boucle `while`. Cette dernière se termine donc.
  La boucle `for` se termine par construction.
  Ainsi, l'algorithme de tri par insertion se termine.
- La proposition suivante est un invariant de la boucle `for` :

  « la liste `[ L[0], ..., L[i] ]` est triée à la fin de l'étape `i` »

  Démontrons le. Soit $n \in \mathbb{N}^*$ et soit `L` une liste de taille n.
  Démontrons par récurrence : $\forall i \in [\![0, n-1]\!]$, $\mathscr{P}(i)$
  où $\mathscr{P}(i)$: la liste `[ L[0], ..., L[i] ]` est triée à la fin de l'étape i.
  - **Initialisation** :
    La liste `[ L[0] ]` contient un seul élément. Elle est donc triée. En particulier, elle est triée à la fin de l'étape 0 (c'est-à-dire avant d'entrer dans la boucle `for`).
    D'où $\mathscr{P}(0)$.
  - **Hérédité** : soit $i \in [\![0, n-2]\!]$.
    Supposons $\mathscr{P}(i)$ et démontrons $\mathscr{P}(i+1)$ (*i.e.* la liste `[ L[0], ..., L[i+1] ]` est triée à la fin de l'étape $i+1$).
    - Lors du $(\texttt{i}+1)^{\text{ème}}$ tour de boucle, à l'issue de la boucle `while`, la variable `j` contient le premier indice `k` (en partant de `i`) tel que :

      $$\texttt{L[k-1]} \leqslant \texttt{L[i+1]} < \texttt{L[k]}$$

    - Comme, par hypothèse de récurrence, les termes de la liste `[ L[0], ..., L[i] ]` sont triés, et que l'on vient de trier le terme `L[i+1]` au sein de cette sous-liste dans l'ordre croissant, la liste `[ L[0], ..., L[i+1] ]` est maintenant triée.

    D'où $\mathscr{P}(i+1)$.

  Par principe de récurrence : $\forall i \in [\![0, n-1]\!]$, $\mathscr{P}(i)$.
  Cet invariant de boucle assure la correction partielle de l'algorithme de tri par insertion.
- L'algorithme de tri par insertion est partiellement correct et se termine. Il est donc correct.

Complexité de l'algorithme
Soit $n \in \mathbb{N}$. Soit `L` une liste à n éléments.
On considère que l'opération élémentaire est la comparaison.

- On effectue toujours $n-1$ étapes (rien n'est fait lorsque la partie gauche ne contient qu'un élément).
- À l'étape i on effectue au maximum i comparaisons. Le pire cas se produit lorsque la liste apparaît initialement dans l'ordre décroissant.
- On effetue alors en tout : $\sum_{i=1}^{n-1} i = \frac{(n-1)\,n}{2}$ comparaisons.

La complexité du tri par insertion est donc en $\underset{n\to+\infty}{O}(n^2)$.

III.5. Exemple : recherche dichotomique dans une liste triée

Considérons l'algorithme suivant, qui effectue, de manière impérative, la recherche d'un élément `x` dans une liste d'entiers `L` *supposée triée* (par ordre croissant). Le principe est de situer, à chaque étape, `x` par rapport à l'élément médian de la liste `L`. La fonction renvoie l'indice où se trouve `x` si trouvé, un message sinon.

```
1  def recherche_dichotomique(L, x) :
2      n = len(L)
3      deb = 0
4      fin = n - 1
5      while fin - deb >= 0 :
6          m = (deb + fin) // 2
7          if L[m] == x :
8              return m
9          elif L[m] > x :
10             fin = m - 1
11         else :
12             deb = m + 1
13     return 'L'élément cherché ne se trouve pas dans la liste'
```

Spécification de l'algorithme
La spécification de l'algorithme est la suivante :

1) signature : `recherche_dichotomique(L : list, x : int)` $\to$ `L : list`,

2) pré-condition : `L` est une liste d'entiers,

3) post-condition : `L` est une liste triée

Correction de l'algorithme

- La variable `fin - deb` définit une suite strictement décroissante d'entiers positifs. En effet, cette variabe est initialisée à `n` (la longueur de la liste `L`) et décroît au moins de 1 à chaque tour de boucle, grâce aux mises à jour de `deb` et `fin`.
  C'est donc un variant de la boucle `while`. Celle-ci se termine donc.
  Ainsi l'algorithme de recherche dichotomique dans un tableau trié se termine.
- La proposition suivante est un invariant de la boucle `while` :

« si `x` est un élément de `L`, alors : `L[deb]` $\leqslant$ `x` $\leqslant$ `L[fin]` »

Démontrons le. Soit $n \in \mathbb{N}$. Soit `L` une liste de taille n.
Démontrons par récurrence : $\forall i \in \mathbb{N}$, $\mathscr{P}(i)$
où $\mathscr{P}(i)$: au $i^{\text{ème}}$ tour de boucle, si `x` est un élément de `L`, alors : `L[deb]` $\leqslant$ `x` $\leqslant$ `L[fin]`.

- **Initialisation** :
  Au $0^{\text{ème}}$ tour de boucle, c'est-à-dire avant la boucle `while`, la variable `deb` contient 0 et la variable `fin` contient $n-1$.
  Comme la liste `L` est triée, alors `L[0]` est le plus petit élément et `L[n-1]` est le plus grand élément de `L`. On en déduit que, si `x` est un élément de `L`, alors : `L[0]` $\leqslant$ `x` $\leqslant$ `L[n-1]`. Ainsi :

$$\texttt{L[deb]} \leqslant \texttt{x} \leqslant \texttt{L[fin]}$$

  D'où $\mathscr{P}(0)$.
- **Hérédité** : Soit $i \in \mathbb{N}^*$.
  Supposons $\mathscr{P}(i)$ et démontrons $\mathscr{P}(i+1)$ (*i.e.* au $(i+1)^{\text{ème}}$ tour de boucle, si `x` est un élément de `L`, alors : `L[deb]` $\leqslant$ `x` $\leqslant$ `L[fin]`)
  Par hypothèse de récurrence, à l'issue du $i^{\text{ème}}$ tour de boucle :

$$\texttt{L[deb]} \leqslant \texttt{x} \leqslant \texttt{L[fin]}$$

  On note : $m = \left\lfloor \dfrac{\texttt{deb} + \texttt{fin}}{2} \right\rfloor$. Trois cas se présentent alors :
  - si `L[m]` $=$ `x`, alors, comme `deb` $\leqslant$ `m` $\leqslant$ `fin` et que la liste `L` est triée dans l'ordre croissant :

$$\begin{array}{ccccc} \texttt{L[deb]} & \leqslant & \texttt{L[m]} & \leqslant & \texttt{L[fin]} \\ & & \| & & \\ & & \texttt{x} & & \end{array}$$

  - si `L[m]` $\geqslant$ `x`, alors, comme la liste `L` est triée dans l'ordre croissant :

$$\texttt{L[deb]} \leqslant \texttt{x} \leqslant \texttt{L[m-1]}$$

    On met ensuite à jour la variable `fin` qui contient maintenant `m - 1`. Ainsi :

$$\texttt{L[deb]} \leqslant \texttt{x} \leqslant \texttt{L[fin]}$$

  - si `L[m]` $\leqslant$ `x`, alors, comme la liste `L` est triée dans l'ordre croissant :

$$\texttt{L[deb+1]} \leqslant \texttt{x} \leqslant \texttt{L[fin]}$$

    On met ensuite à jour la variable `deb` qui contient maintenant `m + 1`. Ainsi :

$$\texttt{L[deb]} \leqslant \texttt{x} \leqslant \texttt{L[fin]}$$

  D'où $\mathscr{P}(i+1)$.

  Par principe de récurrence : $\forall i \in \mathbb{N}$, $\mathscr{P}(i)$.
  Cet invariant de boucle assure la correction partielle de l'algorithme de recherche dichotomique dans un tableau trié.

- L'algorithme de recherche dichotomique est partiellement correct et se termine. Il est donc correct.

Complexité de l'algorithme
Soit $n \in \mathbb{N}$. Soit `L` une liste à n éléments.
On considère que l'opération élémentaire est la comparaison.

- On sait que :
  - × l'intervalle de recherche initial (l'ensemble d'indices $[\![0, n-1]\!]$) est de longueur n.
  - × la longueur de l'intervalle de recherche est divisée par 2 à chaque tour de boucle.
    À la fin du $i^{\text{ème}}$ tour de boucle, l'intervalle de recherche est donc de largeur $\left\lfloor \frac{n}{2^i} \right\rfloor$.
  - × on effectue au maximum 3 comparaisons par tours de boucle.
  - × l'algorithme s'arrête lorsque l'intervalle devient de longueur strictement inférieur à 1.

  On obtient le nombre de tours de boucle effectués en procédant par équivalence :

$$\begin{aligned} \frac{n}{2^i} < 1 &\Leftrightarrow \frac{2^i}{n} > 1 && \textit{(par stricte décroissance de la fonction inverse sur } \mathbb{R}_+^*\textit{)} \\ &\Leftrightarrow 2^i > n \\ &\Leftrightarrow i\ \ln(2) > \ln(n) && \textit{(par stricte croissance de la fonction } \ln \textit{ sur } \mathbb{R}_+^*\textit{)} \end{aligned}$$

  Ainsi, on effectue au plus $\left\lceil \frac{\ln(n)}{\ln(2)} \right\rceil$ tours de boucle.

  Ainsi, on effectue dans le pire cas $3 \times \left\lceil \frac{\ln(n)}{\ln(2)} \right\rceil$ opérations élémentaires.

- La complexité de l'algorithme de recherche dichotomique dans un tableau trié est donc en $\underset{n\to+\infty}{O}\big(\ln(n)\big)$.