Informatique

9

Algorithmique

Cours

Algorithmique

1.I. Introduction

Face à un algorithme, on doit se poser 3 questions :

- L'algorithme donne-t-il un résultat ?
- Le résultat est-il le bon ?
- Est-ce que le temps mis par l'algorithme est raisonnable / optimisé ?

A ces 3 notions sont respectivement associés 3 termes :

- Terminaison
- Correction
- Complexité

1.II. Terminaison

1.II.1 Introduction

Vérifier la terminaison d'un algorithme consiste à vérifier qu'il a bien une fin. Autrement dit, il faut que le nombre d'itérations soit fini.

Lorsqu'une boucle for est utilisée avec un **compteur non perturbé** (pas de modification dans la boucle de ce qui est après le for, nous traiterons un exemple), on connaît à priori le nombre d'itérations qui sont réalisées, on sait donc que l'algorithme se finit.

Lorsqu'une boucle « while » est utilisée, l'algorithme se termine si la condition d'entrée n'est plus vérifiée. Par construction, la condition est vérifiée lors de l'entrée dans l'algorithme afin d'en exécuter le contenu. Il est donc nécessaire de faire évoluer la condition dans la boucle et d'être sûr qu'elle finira par prendre la valeur « False ».

La non terminaison d'un algorithme conduit le logiciel utilisé à répéter indéfiniment des actions sans fin.

1.II.2 Outil d'étude de la terminaison

On définit un « **variant** » ou « convergent » de boucle. C'est un **entier** qui prend des valeurs dans un ensemble de dimension finie (imposé par la condition) et qui évolue strictement et de manière monotone (il ne peut rester identique entre deux itérations) à chaque itération dans l'intervalle. Il est donc sûr que la boucle se terminera. Si son élaboration est impossible, c'est qu'il y a un problème et qu'il faut revoir l'algorithme.

En pratique, on introduit une suite qui sera considéré comme le variant. Par exemple, le variant peut être la somme de la taille de deux listes $len(L1) + len(L2)$…

1.II.3 Exemples

	Programme	Terminaison

for	`n = 100`
	`L = [i for i in range(n)]`
	`Somme = 0`
	`for i in range(len(L)):`
	`    Somme += L[i]`
	`print(Somme)`
		***i* est un variant de boucle**. C'est un entier positif défini dans l'intervalle $[0, n-1]$ dont la valeur croit strictement à chaque itération (i+=1).
		La terminaison est vérifiée

	`s = 0`
	`while s<10:`
	`    s+=0.1`
		10s est un variant de boucle. C'est un entier variant dans l'intervalle $[0,100]$

	`L = [0]`
	`for Terme in L:`
	`    L.append(Terme)`
		Danger : On modifie la taille de L dans la boucle !
		La taille initiale de L vaut $T_0 = 1$.
		A chaque itération, la taille de L est incrémentée de 1 : $\forall i, T_i = T_0 + i$ lors du début de lecture de la boucle.
		La boucle basée sur la liste L appelle l'indice i tant que $i < T_i$
		Soit : $i < T_0 + i \Leftrightarrow 0 < T_0$
		Cette condition est toujours vraie si $T_0 > 0$!
		La terminaison n'est pas vérifiée
		Rq : si ce n'est pas une liste mais un réel après « in », pas de problèmes même si ce réel est modifié dans la boucle

while	`# Division euclidienne a = bq+r`
	
	`a = 25`
	`b = 7`
	
	`q = 0`
	`r = a`
	`while r >= b:`
	`    r = r - b`
	`    q += 1`
	
	`print(q,r)`
		Le reste ***r* est un variant de boucle**. C'est un entier positif (imposé par la condition `r > b` et `r = r - b`) défini dans l'intervalle $[b, +\infty]$ dont la valeur décroit strictement à chaque itération (`r = r - b`).
		La terminaison est vérifiée

	`# Nombre pair -> 0 impaire -> 1`
	
	`N = 23`
	`while N != 1:`
	`    N += -2`
	`print(N)`
		N décroit à chaque itération mais n'évolue pas dans un intervalle fini. La condition `N != 1` définie l'intervalle $N \backslash 1$.
		On ne répond donc pas à la condition de terminaison.
		Si N est impair au départ, la boucle a une fin.
		Si N est pair, la boucle n'a pas de fin.
		La terminaison n'est pas vérifiée

1.III. Correction

1.III.1 Introduction

La correction d'un algorithme, qu'il soit composé ou non d'itérations, consiste à vérifier qu'il donne la valeur attendue. Cela revient à en faire la démonstration.

Vérifier un algorithme présentant une boucle consiste à réaliser une démonstration comme la récurrence en mathématiques.

1.III.2 Outil d'étude de la correction

Appelons $\mathcal{P}$ une propriété quelconque, qui doit être vérifiés tout au long du déroulement de l'algorithme. La propriété $\mathcal{P}$ s'appelle un invariant de boucle.

Trouver la propriété est la phase « difficile ». Il faut de l'intuition, ou écrire l'algorithme pour quelques exemples.

Exemple : Soit l'algorithme suivant :

```
# Factoriel n, n>0

p = n
a = n
for i in range(n-1):
    a = a - 1
    p = p * a
print(p)
```

Trouvons la propriété $\mathcal{P}(i)$ qui définit les valeurs $\begin{cases} a_i = \cdots \\ p_i = \cdots \end{cases}$

$n=1$	RAS	Valeurs de a	Valeurs de p
$n=2$	$i=0: \begin{cases} a_0 = n-1 = 2-1 = 1 \\ p_0 = n * a_0 = 2*1 = 2 \end{cases}$	$a_0 = 1 = 2-1-0$	$p_0 = 2 = 2*1 = \frac{2!}{0!}$
$n=3$	$i=0: \begin{cases} a_0 = a-1 = 3-1 = 2 \\ p_0 = p * a_0 = 3*2 = 6 \end{cases}$ $i=1: \begin{cases} a_1 = a_0 - 1 = 2-1 = 1 \\ p_1 = p_0 * a_1 = 6*1 = 6 \end{cases}$	$a_0 = 2 = 3-1-0$ $a_1 = 1 = 3-1-1$	$p_0 = 6 = 3*2 = \frac{3!}{1!}$ $p_0 = 6 = 3*2*1 = \frac{3!}{0!}$
$n=4$	$i=0: \begin{cases} a_0 = a-1 = 4-1 = 3 \\ p_0 = p * a_0 = 4*3 = 12 \end{cases}$ $i=1: \begin{cases} a_1 = a_0 - 1 = 3-1 = 2 \\ p_1 = p_0 * a_1 = 12*2 = 24 \end{cases}$ $i=2: \begin{cases} a_2 = a_1 - 1 = 2-1 = 1 \\ p_2 = p_1 * a_2 = 24*1 \end{cases}$	$a_0 = 3 = 4-1-0$ $a_1 = 2 = 4-1-1$ $a_2 = 1 = 4-1-2$	$p_0 = 12 = 4*3 = \frac{4!}{2!}$ $p_1 = 24 = 4*3*2 = \frac{4!}{1!}$ $p_2 = 24 = 4*3*2*1 = \frac{4!}{0!}$

Avec notre intuition, on peut proposer la propriété suivante :

$$\mathcal{P}(i): \begin{cases} a_i = n - i - 1 \\ p_i = \dfrac{n!}{(n-2-i)!} \end{cases}$$

1.III.3 Correction d'une boucle « for »

Notons $\mathcal{P}(k)$ la propriété après l'itération pour $i = k$

Corriger une boucle « for » consiste à réaliser 3 étapes :

- Initialisation : $\mathcal{P}$ vraie à la première itération : $\mathcal{P}(0)$
- Transmission : Si $\mathcal{P}$ vraie à l'itération i, $\mathcal{P}$ vraie à l'itération $i+1$: $\mathcal{P}(i) \Rightarrow \mathcal{P}(i+1)$
- Sortie : A la dernière itération n, $\mathcal{P}$ doit permettre de démontrer que le résultat obtenu est le résultat attendu : $\mathcal{P}(n) \Rightarrow Résultat$

Remarque : la première itération correspond à $i = 0$

1.III.4 Correction d'une boucle « while »

Notons $\mathcal{P}(i)$ la propriété après la i eme itération.

Corriger une boucle « while » consiste à réaliser 3 étapes :

- Initialisation : $\mathcal{P}$ vraie **avant** la première itération : $\mathcal{P}(0)$
- Transmission : Si $\mathcal{P}$ vraie avant l'itération i, $\mathcal{P}$ vraie après: $\mathcal{P}(i) \Rightarrow \mathcal{P}(i+1)$
- Sortie : Après la dernière itération n (lorsque la condition devient fausse pour la première fois), $\mathcal{P}(n)$ doit permettre de démontrer que le résultat obtenu est le résultat attendu : $\mathcal{P}(n) \Rightarrow Résultat$

Remarque : $i = 0$ correspond à l'état initial, avant la première itération. $i = 1$ correspond donc à la fin de la première itération, contrairement à la correction de la boucle for où $i = 1$ correspond à la fin de la seconde itération, $i = 0$ étant la fin de la première itération

1.III.5 Exemples

<table>
<tr><th></th><th>Programme</th><th>Correction</th></tr>
<tr><td>Programme normal</td><td>

```
# Partie
positive

X = 10
Y = (X +
abs(X))/ 2
print(Y)
```

</td><td>Il suffit de vérifier que le programme fait ce qu'il annonce
$$\frac{X+\text{abs}\,X}{2}=\begin{cases}\frac{X+X}{2}=X\ si\ X>0\\ \frac{X-X}{2}=0\ si\ X>0\end{cases}$$
Ce programme est correct</td></tr>
<tr><td rowspan="2">for</td><td>

```
# Factoriel n,
n>0

p = n
a = n
for i in
range(n-1):
    a = a - 1
    p = p * a
print(p)
```

</td><td>Vérifions si cette fonction calcule bien $n!$ Si $n>0$
Supposons $n\geq 1$. Notons a_i et p_i les valeurs de a et p à la fin de l'itération i.
Algorithme : à tout moment, on a : $\begin{cases}a_{i+1}=a_i-1\\ p_{i+1}=p_i*a_{i+1}\end{cases}$
Propriété : soit la propriété $\mathcal{P}(i)$: $\begin{cases}a_i=n-i-1\\ p_i=\frac{n!}{(n-2-i)!}\end{cases}$
Initialisation : $i=0$, $\mathcal{P}(0)$: $\begin{cases}a_0=n-1=n-0-1=n-i-1\\ p_0=n*(n-1)=\frac{n!}{(n-2-0)!}=\frac{n!}{(n-2-i)!}\end{cases}$
Transmission : $i\in[1,n-2]$, $\mathcal{P}(i)$ supposée vraie :
$$\begin{cases}a_i=n-i-1\\ p_i=\frac{n!}{(n-2-i)!}\end{cases}$$
$$\begin{cases}a_{i+1}=a_i-1=n-1-i-1=n-(i+1)-1\\ p_{i+1}=p_i*a_{i+1}=p_i*(a_i-1)=p_i*(n-2-i)=\frac{n!\,(n-2-i)}{(n-2-i)!}\end{cases}$$
$$\begin{cases}a_{i+1}=n-(i+1)-1\\ p_{i+1}=\frac{n!}{(n-3-i)!}=\frac{n!}{(n-2-(i+1))!}\end{cases}$$
$\mathcal{P}(i+1)$ est donc vraie
Sortie : quand $i=n-2$: $\mathcal{P}(n-2)$ vraie : $p_{n-2}=\frac{n!}{(n-2-(n-2))!}=\frac{n!}{0!}=n!$
Ce programme est correct</td></tr>
<tr><td>

```
# Factoriel n,
n>0

p = 1
for i in
range(n+1):
    p = p *
(i+1)
print(p)
```

</td><td>Vérifions si cette fonction calcule bien $n!$ Si $n>0$
Supposons $n\geq 1$. Notons p_i la valeur de p à la fin de l'itération i.
Algorithme : à tout moment, on a : $p_{i+1}=p_i*(i+2)$
Propriété : soit la propriété : $\mathcal{P}(i)$: $p_i=(i+1)!$
Initialisation : $i=0$, $\mathcal{P}(0)$: $p_0=1=(0+1)!=(i+1)!$
Transmission : $i\in[1,n]$, $\mathcal{P}(i)$ supposée vraie $p_i=i!$
$$p_{i+1}=(i+1)!*(i+2)=(i+2)!$$
$\mathcal{P}(i+1)$ est donc vraie
Sortie : quand $i=n$: $\mathcal{P}(n)$ vraie : $p_n=(n+1)!$
Ce programme est incorrect – Il faudrait changer n+1 en n</td></tr>
</table>

<table>
<tr><th></th><th>Programme</th><th>Correction</th></tr>
<tr><td rowspan="2">while</td><td>

```
# Somme des n 1° entiers

s = 0
a = 0
while a <= n:
    a += 1
    s += a
print(s)
```

</td><td>
Vérifions si cette fonction calcule bien $\sum_{i=1}^{n} i$
Supposons $n \geq 0$. Notons a_i et s_i les valeurs de a et s à la fin de l'itération i.
Algorithme : à tout moment, on a : $\begin{cases} a_{i+1} = a_i + 1 \\ s_{i+1} = s_i + a_{i+1} \end{cases}$
Propriété : soit la propriété : $\mathcal{P}(i)$: $\begin{cases} a_i = i \\ s_i = \sum_{j=1}^{i} j \end{cases}$
Initialisation : $i = 0$ (avant boucle), $\mathcal{P}(0)$: $\begin{cases} a_0 = 0 = i \\ s_0 = 0 = \sum_{j=1}^{0} j \end{cases}$
Transmission : $i \in [1, n+1]$, $\mathcal{P}(i)$ supposée vraie :
$$\begin{cases} a_i = i \\ s_i = \sum_{j=1}^{i} j \end{cases}$$
$$\begin{cases} a_{i+1} = a_i + 1 = i + 1 \\ s_{i+1} = s_i + a_{i+1} = \sum_{j=1}^{i} j + (i+1) = \sum_{j=1}^{i+1} j \end{cases}$$
$\mathcal{P}(i+1)$ est donc vraie
Sortie : quand $a_i = n+1$, soit $i = n+1$: $\mathcal{P}(n+1)$ vraie : $p_{n+1} = \sum_{i=1}^{n+1} i$
Ce programme est incorrect – Il faudrait changer <=n en <n
</td></tr>
<tr><td>

```
# Factoriel n, n>0

p = 1
a = 0
while a < n:
    a = a + 1
    p = p * a
print(p)
```

</td><td>
Vérifions si cette fonction calcule bien $n!$ Si $n > 0$
Supposons $n \geq 1$. Notons a_i et p_i les valeurs de a et p à la fin de l'itération i.
Algorithme : à tout moment, on a : $\begin{cases} a_{i+1} = a_i + 1 \\ p_{i+1} = p_i * a_{i+1} \end{cases}$
Propriété : soit la propriété : $\mathcal{P}(i)$: $\begin{cases} a_i = i \\ p_i = i! \end{cases}$
Initialisation : $i = 0$ (avant boucle), $\mathcal{P}(0)$: $\begin{cases} a_0 = 0 = i \\ p_0 = 1 = 0! = i! \end{cases}$
Transmission : $i \in [1, n]$, $\mathcal{P}(i)$ supposée vraie :
$$\begin{cases} a_i = i \\ p_i = i! \end{cases}$$
$$\begin{cases} a_{i+1} = a_i + 1 = i + 1 \\ p_{i+1} = p_i * a_{i+1} = i!\,(i+1) = (i+1)! \end{cases}$$
$\mathcal{P}(i+1)$ est donc vraie
Sortie : quand $a_i = n$, soit $i = n$: $\mathcal{P}(n)$ vraie : $p_n = n\,!$
Ce programme est correct
</td></tr>
</table>

1.III.6 Outils de vérification

Il existe quelques outils qui peuvent permettre de détecter en cours d'exécution d'un code des erreurs qui conduiraient à un mauvais résultat. Parmi eux, parlons de la commande « assert ».

1.III.6.a Commande « assert »

Soit la fonction suivante :

```
def f(L):
    print(L[0])
```

```
>>> f([1,2,3])
1

>>> f([])
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "<tmp 1>", line 2, in f
    print(L[0])
IndexError: list index out of range
```

Lorsque la liste est vide, il y a une erreur. On peut faire en sorte de vérifier si la liste est vide avant d'appeler L[0] et de renvoyer un message souhaité si elle est vide, ce qui arrêtera l'exécution du code. Pour cela, on utilise « assert » :

```
assert booleen, "message si False"
```

```
>>> assert True,"Message si False"

>>> assert False,"Message si False"
Traceback (most recent call last):
  File "<console>", line 1, in <module>
AssertionError: Message si False
```

On modifie alors la fonction ainsi :

```
def f(L):
    assert len(L)>0,"Liste vide"
    print(L[0])
```

```
>>> f([1,2,3])
1

>>> f([])
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "<tmp 1>", line 2, in f
    assert len(L)>0,"Liste vide"
AssertionError: Liste vide
```

L'avantage est de savoir où est le problème, car de toute manière, le code présentait un problème.

Dans certains cas, cela peut déceler une erreur qui n'aurait pas fait bogué le code. Prenons l'exemple suivant :

```
def f(L):
    s = 0
    for t in L:
        s += t
    L.append(s)

def g(L):
    Res = f(L)
    return Res

L = [1,2,3]
Res = g(L)
```

Ce code ne présente aucune erreur, et pourtant il ne renvoie pas le résultat attendu !

```
>>> Res

>>> Res == None
True
```

J'aurais pu prendre un exemple plus flagrant, où on pourrait croire au résultat. Ici, le résultat est un None… On s'en rendra vite compte, ou alors viendra vite un message d'erreur si on l'utilise… Mais j'ai pris cet exemple car plusieurs d'élèves font ce type d'erreur chaque année et cela permet quand même d'illustrer « assert ».

Allons plus loin en faisant apparaître une erreur :

```
def f(L):
    s = 0
    for t in L:
        s += t
    L.append(s)

def g(L):
    Res = f(L)
    return Res

L = [1,2,3]
Res = g(L)
Res.append(1)
```

```
>>> (executing file "<tmp 1>")
Traceback (most recent call last):
  File "<tmp 1>", line 13, in <module>
    Res.append(1)
AttributeError: 'NoneType' object has no a
ttribute 'append'
```

Res serait un Nonetype… Ajoutons un test avec « assert » afin de vérifier que le résultat renvoyé par f est bien une liste :

```
def f(L):
    s = 0
    for t in L:
        s += t
    L.append(s)

def g(L):
    Res = f(L)
    assert type(Res)==list,"Res n'est pas
une liste mais un " + str(type(Res))
    return Res

L = [1,2,3]
Res = g(L)
Res.append(1)
```

```
>>> (executing file "<tmp 1>")
Traceback (most recent call last):
  File "<tmp 1>", line 13, in <module>
    Res = g(L)
  File "<tmp 1>", line 9, in g
    assert type(Res)==list,"Res n'est pas
une liste mais un " + str(type(Res))
AssertionError: Res n'est pas une liste ma
is un <class 'NoneType'>
```

Eh bien non, c'est un None…

Effectivement, il fallait soit renvoyer L dans f, soit ne pas créer Res dans g puisque L est modifiée en place avec le « append ». Voici deux code justes faisant ce qui était attendu :

```
def f(L):
    s = 0
    for t in L:
        s += t
    L.append(s)

def g(L):
    f(L)
    return L
```

```
def f(L):
    s = 0
    for t in L:
        s += t
    L.append(s)
    return L

def g(L):
    Res = f(L)
    return Res
```

Voici maintenant un exemple ou « assert » prend beaucoup de sens, et illustrant un problème que j'ai rencontré pendant ma thèse. En effet, je simulais l'écoulement d'un polymère fondu soumis à un rayonnement laser. Il fallait à la fois réaliser une résolution d'écoulement fluide afin de mettre à jour la position des points de fluide à cheque instant, et une résolution thermique. Pour des questions de convergence des solutions, j'étais limité par la valeurs des pas de temps pour chaque résolution. Il m'a donc fallu, entre deux pas de temps fluides, réaliser plusieurs pas de temps thermiques en découpant le pas de temps fluide en n pas de temps thermiques.

Pour simplifier les choses, disons qu'à chaque pas de temps fluide, on réalise 10 pas de temps thermiques. On défini le nombre de points nf de la résolution Euler en fluide, nt ne nombre de points thermiques, le temps final de 10 secondes pour chaque résolution, et les pas de temps associés :

```
T = 10
```

```
nf = 1000
dtf = T/(nf-1)
```

```
nt = nf*10
dtt = T/(nt-1)
```

```
tf = 0
tt = 0
while tf < T:
    tf += dtf
    while tt < tf:
        tt += dtt
```

A la fin de cette simulation, on doit avoir tf=tt. Je ne le vérifiais pas avec une assertion, mais voici ce que j'aurais obtenu si j'avais mis une assertion :

```
assert tt==tf,"Erreur sur les temps"
```

```
AssertionError: Erreur sur les temps
```

```
>>> tf
10.010010010010008
```

En effet :

```
>>> tt
10.011001100109636
```

Sans assertion, on passe littéralement à côté de cette erreur. Et cela a eu de l'influence dans mes simulation, j'ai dû chercher et trouver d'où venait le problème. C'est en réalité un cumul d'erreurs d'arrondis (cf cours codage des nombres), et pour s'en convaincre, essayer ce code avec T=10000, nf=1001 et nt=10001, on travaille avec des entiers et tf=tt à la fin.

Si vous vous dites « Normalement, a et b sont égaux ici », ayez le réflexe `assert a==b`

1.III.6.b Typage des données

Le « typage » consiste à préciser les types des données en entrée et sortie des fonctions afin de s'assurer de la cohérence des données manipulées. On précise le type des arguments avec un « **:** » et le type de sortie avec « **->** » :

```
def f(x:float)->float:
```

Exemples :

```
def f(x:float)->float:
    return x**2

test = f(2)
print(test)

def g(h:callable,x)->float:
    return h(x)

test = g(f,2)
print(test)
```

```
4
4
```

```
def h(i:int):
    return i+1

test = h(3)
print(test)
```

```
4
```

Attention, ces « annotations » ne sont là qu'à titre indicatif :

- Les données d'entrée peuvent avoir des types différents
- La donnée de sortie ne sera pas typée comme précisé après « **->** »

Je n'irai pas dans plus de détails, ce paragraphe a seulement pour but de faire en sorte que vous ne soyez pas déroutés si, dans les épreuves, le typage est utilisé.

1.III.6.c Commande « try »

Il est possible de tester des choses sous python, et de faire en sorte que le code ne bug pas pour autant en cas d'erreur. Il suffit d'utiliser la commande « try ». Voici un exemple :

```
while True:
    try:
        x = int(input("Entrer un nombre:"))
        break # Passe ici si pas d'erreur
    except :
        print("Oups")
```

Attend obligatoirement un nombre

```
L = []
try:
    x = L.pop()
except:
    print("L est vide")
```

Ne bug pas si L est vide

On peut aller plus loin en précisant le type d'erreur après **except**, cf aide python !

1.IV. Complexité

1.IV.1 Introduction

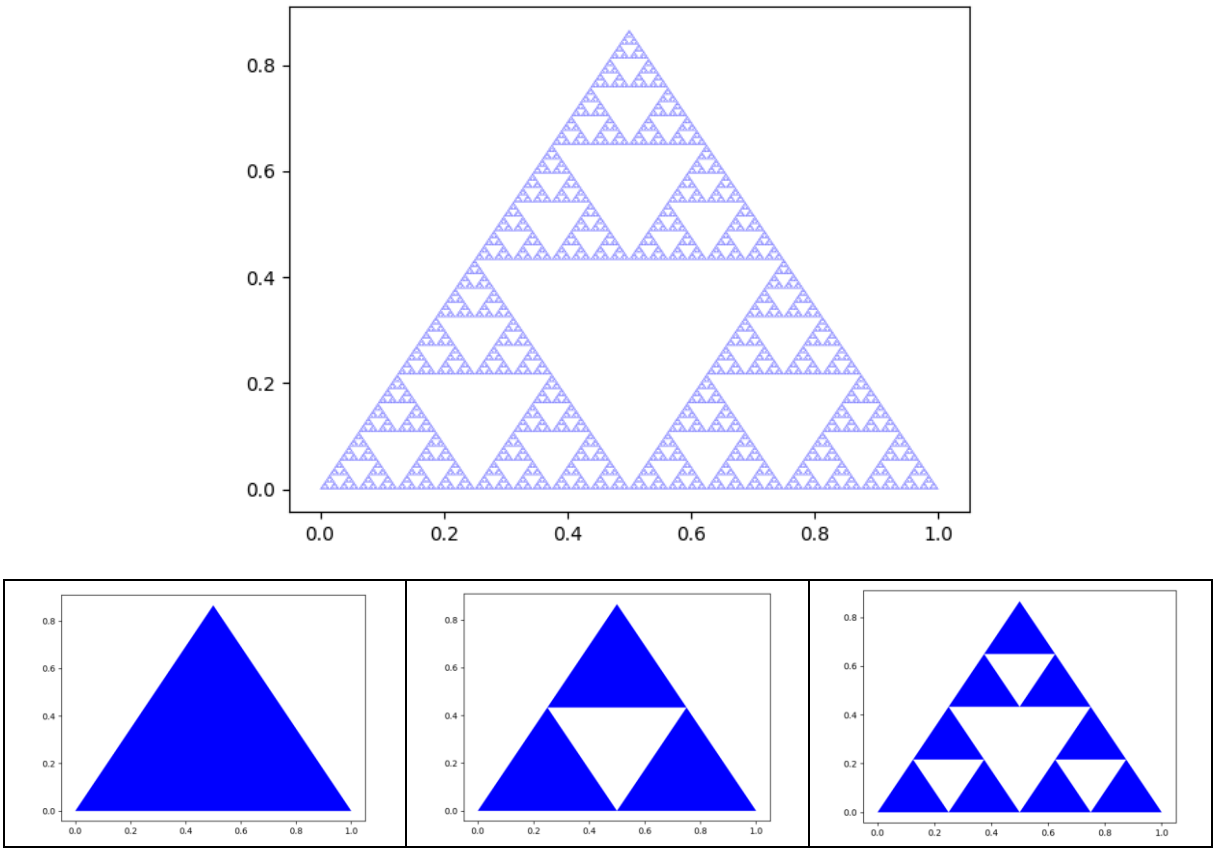
Un algorithme exécute une série d'opérations. Celles-ci mettent un certain temps à être exécutées et peuvent, selon les variables stockées, occuper un espace mémoire important. **On parle de complexité en temps et en espace des algorithmes**.

Il ne faut pas confondre complexité et capacités des ordinateurs. Un programme complexe en mémoire tournant sur un ordinateur ayant une capacité mémoire importante sera exécuté sans problèmes. Un programme de même complexité en temps ne prendra pas le même temps d'exécution sur deux ordinateurs différents (fréquence de processeur par exemple).

Aujourd'hui, la mémoire n'est généralement pas un problème. On s'intéresse souvent plus à la complexité en temps des algorithmes.

Remarque : Certains simulations numériques peuvent prendre plusieurs mois à plusieurs années de calculs.

Exemple de réalisation récursive qui consomme un temps de calcul en 3^n, n étan t le nombre de sous triangles demandés. Réalisation de l'image : 5 minutes !



Le domaine météorologique est un bon exemple dans lequel lorsque l'on lance un calcul de prévision pour dans 2h, il est évidemment important d'obtenir le calcul le plus fiable, en moins de 2h…

1.IV.2 Complexité en temps

On définit un paramètre de complexité qui correspond à un nombre qui va définir en quelques sorte le nombre d'opérations à réaliser.

Prenons l'exemple suivant :

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

Appelons t_a le temps mis pour effectuer une affectation de variable (`Res = 0`, `n = 100`, `Res = Res`).

Appelons t_o le temps mis pour effectuer une opération sur un entier (augmenter i, calculer `Res + i`)

On peut alors estimer le temps total T d'exécution de l'algorithme ci-dessus :

$$T = 2t_a + n(2t_o + t_a)$$

Les temps t_a et t_o dépendent du langage de programmation et de l'ordinateur utilisés.

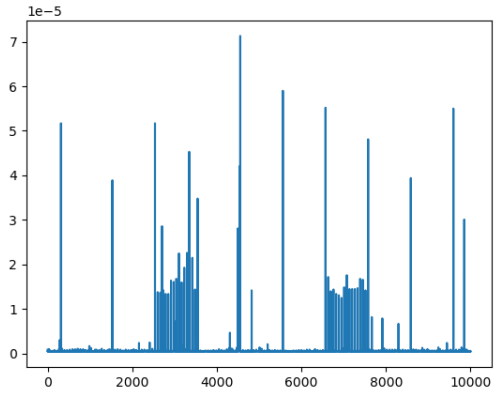
On dira que cet algorithme est de complexité $O(n)$, où que son temps d'exécution augmente linéairement avec le paramètre n.

1.IV.3 Ordres de grandeurs de temps de calcul

L'ordre de grandeur d'un calcul d'ordre 1 est $t = 1 * s = 10^{-6}\ s$.

Voici un code qui permet de mesurer le temps d'exécution minimum de l'ordinateur :

```
from time import perf_counter as tps
n = 1000
Ln = []
Lt = []
for i in range(n):
    t1 = tps()
    # a=1
    # 1+1
    t2 = tps()
    Temps = t2-t1
    Ln.append(i)
    Lt.append(Temps)
T_min = min(Lt)
print(T_min)
from matplotlib import pyplot as plt
plt.close('all')
def Affiche(fig,X,Y):
     plt.figure(fig)
     plt.plot(X,Y)
     plt.show()
Affiche(1,Ln,Lt)
```




```
>>> (executing file "Temp
s mini.py")
9.999996564147295e-08
```

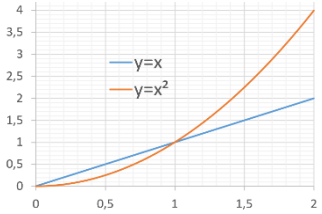
Avec mon ordinateur de la région IDF, on voit que le temps minimum entre la création de t1 et t2 est d'environ 10^{-7}. Il faut bien étudier le temps minimum, l'ordinateur faisant des choses en parallèle lorsque ce temps n'est pas le plus petit possible. J'ai ainsi étudié le temps pour réaliser une opération 1+1 et une affectation a=1, et j'obtiens ces ordres de grandeur : $\begin{cases} t_o \approx 10^{-7} s \\ t_a \approx 2.10^{-7} s \end{cases}$. Si vous le souhaitez, exécutez ce code chez vous.

A partir de là, il est assez simple de calculer le temps d'exécution T d'un algorithme de complexité $Of(n)$ en calculant :

$$T = t * f(n)$$

	$n = 1$	$n = 10^5$	$n = 10^{10}$
$O(1)$	$1\ \mu s$	$1\ \mu s$	$1\ \mu s$
$O(\log_2 n)$	$1\ \mu s$	$17\ \mu s$	$33\ \mu s$
$O(n)$	$1\ \mu s$	$0{,}1\ s$	$2{,}7\ h$
$O(n^2)$	$1\ \mu s$	$2{,}7\ h$	$31\ 700\ millénaires$
$O(n^n)$	$1\ \mu s$	Démesuré	

Attention, ce sont des ordres de grandeur. Pour de faibles valeurs de n, un algorithme $O(an)$ peut être moins rapide qu'un algorithme $Of(bn^2)$.



1.IV.4 Notation de Landau – Grand O

En mathématiques, vous allez rapidement utiliser la notation $o(n)$ ou « petit o de n », et $O(n)$ ou « grand o de n ». Ils correspondent à une comparaison asymptotique.

- $o(n)$ s'associe à quelque chose de petit devant n au voisinage de l'infini
- $O(n)$ s'associe à quelque chose de dominé par n au voisinage de l'infini

Dans le cadre de l'analyse de la complexité des algorithmes, nous n'utiliserons que la notation $O(f)$ pour dire que cette complexité est de l'ordre de grandeur de la fonction f dans la parenthèse.

Mathématiquement, dire : $f(x) = O\big(g(x)\big)$ signifie $\exists N \in \mathbb{R}^+ \ \& \ A \in \mathbb{R} \ / \forall x > N, \left|\frac{f(x)}{g(x)}\right| < A$

Lorsque qu'une complexité est indépendante de n, c'est donc une constante. On donne alors le résultat $O(1)$.

Si une complexité est en $O(n)$ et si par exemple $n = 100$, NE SURTOUT PAS dire $O(100)$, ce qui est équivalent à $O(1)$.

Evidemment, lorsque l'on demande une complexité, on attend la fonction de croissance « la plus petite » possible pour décrire le comportement temporel du temps de calcul maximum. Ainsi, dire que (presque) tout code est de complexité $O(n^n)$ n'est pas faux……

Ainsi :

$f(n)$	$C(n)$	Preuve
10	$O(1)$	$\frac{10}{1} = 10$
$2n$	$O(n)$	$\frac{2n}{n} = 2$
$2n^2$	$O(n^2)$	$\frac{2n^2}{n^2} = 2$
$2n + 3n^2$	$O(n^2)$	$\frac{2n + 3n^2}{n^2} \underset{+\infty}{\sim} \frac{3n^2}{n^2} = 3$
n	$O(n^3)$	$\frac{n}{n^3} \underset{+\infty}{\sim} \frac{1}{n^2} \underset{+\infty}{\to} 0$ A éviter toutefois…
$2^n + 3^n$	$O(3^n)$	$\frac{2^n + 3^n}{3^n} = \left(\frac{2}{3}\right)^n + 1 \underset{+\infty}{\sim} 1$
2^{n+1}	$O(2^n)$	$\frac{2^{n+1}}{2^n} = 2$
$\log n$	$O(\ln n)$	$\frac{\log n}{\ln n} = \frac{\frac{\ln n}{\ln 10}}{\ln n} = \frac{1}{\ln 10}$
$\log_2 n$	$O(\ln n)$	$\frac{\log_2 n}{\ln n} = \frac{\frac{\ln n}{\ln 2}}{\ln n} = \frac{1}{\ln 2}$

1.IV.5 Exemples

1.IV.5.a Algorithmes simples

Soient les algorithmes suivants, dont le résultat est identique :

Code Python	Complexité temporelle

`n = 100`
`Res = n*2*(1 + n)*n/2`
`print(Res)`
	$O(1)$

`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)`
	$O(n)$

`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):`
`        Res = Res + i + j`
`print(Res)`
	$O(n^2)$

`Res = 0`
`n = 100`
`for i in range(1,n+1):`
`    Res += i`
`for j in range(1,n+1):`
`    Res += j`
`print(Res)`
	$O(n)$

`Res = 0`
`n = 100`
`m = 100`
`for i in range(1,n+1): # Somme des termes de 1 à n`
`    for j in range(1,m+1):`
`        Res = Res + i + j`
`print(Res)`
	$O(n * m)$
	
	Exemple : traitement d'images

Si l'on fait tourner ces 3 algorithmes à la suite pour $n = 10^4$ par exemple, on verra immédiatement le temps mis par l'ordinateur pour donner les différentes solutions (cf paragraphe « Outil d'analyse de complexité) quelques pages plus loin.

Un second exemple parlant car vous risquez fort de le coder ainsi :

Code en O(n)	Code identique, en $O(n^2)$

`def Moyenne(Liste):`
`  Taille = len(Liste)`
`  Somme = 0`
`  for i in range(Taille):`
`  Somme += Liste[i]`
`  Moyenne = Somme / Taille`
`  return Moyenne`
 	
`def Variance(Liste):`
`  Taille = len(Liste)`
`  Somme = 0`
`  Moy = Moyenne(Liste)`
`  for Terme in Liste:`
`  Val = (Terme - Moy)**2`
`  Somme+= Val`
`  Variance = Somme / Taille`
`  return Variance`
	`def Moyenne(Liste):`
	`  Taille = len(Liste)`
	`  Somme = 0`
	`  for i in range(Taille):`
	`  Somme += Liste[i]`
	`  Moyenne = Somme / Taille`
	`  return Moyenne`
	
	`def Variance(Liste):`
	`  Taille = len(Liste)`
	`  Somme = 0`
	`  for Terme in Liste:`
	`  Moy = Moyenne(Liste)`
	`  Val = (Terme - Moy)**2`
	`  Somme+= Val`
	`  Variance = Somme / Taille`
	`  return Variance`

1.IV.5.b Fonctions récursives

$$u_0 = 1, n \geq 1, u_{n+1} = \begin{cases} u_n + 1 \; si \; u_n < 1 \\ \frac{u_n}{2} \; sinon \end{cases}$$

Code Python	Complexité temporelle

`def rec(n):`
`  if n==0:`
`  return 1`
`  else:`
`  Un_m1 = rec(n-1)`
`  if Un_m1 < 1:`
`  Un = Un_m1 + 1`
`  else:`
`  Un = Un_m1 / 2`
`  return Un`
	Soit a_n le nombre d'appels de rec à l'appel n:
	$a_0 = 0$
	$a_{n+1} = a_n$
	$a_n = a_0 + n = n$
	(Suite arithmétique)
	
	$\Rightarrow O(n)$

`def rec(n):`
`  if n==0:`
`  return 1`
`  else:`
`  if rec(n-1) < 1:`
`  Un = rec(n-1) + 1`
`  else:`
`  Un = rec(n-1) / 2`
`  return Un`
	Soit a_n le nombre d'appels de rec à l'appel n:
	$a_0 = 0$
	$a_1 = 2$
	$\forall n \geq 1, a_{n+1} = 2a_n$
	$a_n = 2^{n-1}a_1 = 2^n$
	(Suite géométrique)
	
	$\Rightarrow O(2^n)$

1.IV.6 Outil d'analyse de complexité : time.perf_counter()

Importons un outil qui permet de définir le temps actuel afin de calculer le temps mis par l'ordinateur pour effectuer chacun des algorithmes vus plus haut :

Code exécuté	Résultat (temps en secondes)

`import time`
 	
`tic = time.perf_counter()`
`n = 10000`
`Res = n*2*(1 + n)*n/2`
`print(Res)`
`toc = time.perf_counter()`
`print("Temps 1: ",toc - tic)`
 	
`tic = time.perf_counter()`
`Res = 0`
`n = 10000`
`for i in range(1,n+1): # Somme des termes de 1 à n`
`    Res = Res + 2*n*i`
`print(Res)`
`toc = time.perf_counter()`
`print("Temps 2: ",toc - tic)`
 	
`tic = time.perf_counter()`
`Res = 0`
`n = 10000`
`for i in range(1,n+1): # Somme des termes de 1 à n`
`    for j in range(1,n+1):`
`        Res = Res + i + j`
`print(Res)`
`toc = time.perf_counter()`
`print("Temps 3: ",toc - tic)`
	`1000100000000.0`
	`Temps 1:  3.499999999689862e-05`
	`1000100000000`
	`Temps 2:  0.0017922999999981926`
	`1000100000000`
	`Temps 3:  22.626590499999995`

Attention, ces temps dépendent de l'ordinateur utilisé. On voit clairement que pour le même calcul, les temps évoluent fortement en fonction de la méthode de programmation choisie. Il conviendra donc toujours de réfléchir à la meilleure manière de programmer, dès que les temps de calculs deviennent importants.

Voici deux codes permettant de trouver :

- Un ordre de grandeur du temps minimum d'un calcul ou d'une affectation en python (déjà partagé plus haut) : LIEN
- La complexité d'une instruction (à vous de comprendre et faire évoluer le code : LIEN

1.IV.7 Complexité en log

Prenons un dernier exemple d'algorithme dont la complexité est en $O(\log_2 n)$. Soit une liste L délimitant des intervalles du type : $L = [0,10,20,30,40,50,60,70,80,90,100]$ ou d'une manière générale, des intervalles de largeur l, dont la limite basse est 0 et la limite haute $l * n$:

$$L = [l * i \; for \; i \; in \; range(n+1)]$$

On cherche alors dans quel intervalle se trouve une valeur x quelconque. On souhaite donc déterminer si $L[i] \leq x < L[i+1]$ afin de remonter à i et $i+1$. Deux méthodes s'offrent à nous :

- Parcourir les intervalles jusqu'à trouver le bon, ce qui donne une complexité en $O(1)$ dans le meilleur des cas, et $O(n)$ dans le pire des cas
- Procéder par dichotomie, c'est-à-dire diviser la plage d'étude en 2, voir si x est supérieur ou inférieur à la valeur médiane, définir alors le nouvel intervalle du bon côté continuer jusqu'à n'avoir plus qu'un intervalle. Calculons la complexité de cette méthode :

```
n = 10
l = 10
L = [l*i for i in range(n+1)]
x = 50
import copy
L_Mod = L.copy()
while len(L_Mod) >= 3:
    print(L_Mod)
    Separateur_Med =
int(len(L_Mod)/2)
    Lg = L_Mod[0:Separateur_Med+1]
    Ld = L_Mod[Separateur_Med:]
    L_Med = L_Mod[Separateur_Med]
    if x >= L_Med:
        L_Mod = Ld
    elif x < L_Med:
        L_Mod = Lg
print(L_Mod)
```

On part de n intervalles, on divise t fois la plage d'intervalles jusqu'à ce que l'intervalle final ait une largeur égale à 1. On a donc :

$$\frac{n}{2^t} = 1$$

On cherche le nombre de divisions à réaliser, t :

$$2^t = n$$
$$e^{t \ln 2} = n$$
$$t \ln 2 = \ln n$$
$$t = \frac{\ln n}{\ln 2} = \log_2 n$$

Soit finalement :

$$O(\log_2 n)$$

On peut aussi dire :

$$O(\ln n)$$

Quelques explications et exemples :

Tant que la liste L contient au moins 3 valeurs, c'est qu'il y a au moins deux intervalles, $[10,20,30]$ par exemple contient les intervalles $[10,20]$ et $[20,30]$

Ensuite, voici deux exemples d'exécution d'une étape du while pour deux listes différentes :

$L = [10,20,30]$	$L = [10,20,30,40]$
Len(L)=3 Separateur_Med = 1 Lg = L[0:2]=[10,20] Ld = L[1:]=[20,30] Lmed=L[1]=20	Len(L)=4 Separateur_Med = 2 Lg = L[0:3]=[10,20,30] Ld = L[2:]=[30,40] Lmed=L[2]=30
On compare alors x à Lmed pour retenir l'intervalle du bon côté	

On notera qu'il est important que l'inégalité du test (`x >= L_Med`) soit en accord avec les choix réalisés dans la définition des intervalles… Il faut que Lmed soit dans Ld