II. Terminaison et preuve des boucles

II.1. Introduction

Lorsque l'on écrit une boucle `while`, il est possible que dans certains cas cette boucle ***ne se termine pas*** ce qui a pour conséquence que la boucle bloque non seulement l'exécution du programme auquel elle appartient, mais aussi l'exécution des autres programmes car elle consomme l'essentiel des ressources de la machine.
Pour éviter cela, on souhaite pouvoir ***prouver qu'une boucle se termine et produit effectivement l'effet attendu***.

II.2. Terminaison d'une boucle `while`

Définition 7.1 (Variant de boucle)

On appelle ***variant de boucle*** une grandeur vérifiant les propriétés suivantes :
- les valeurs prises par cette grandeur sont des ***entiers positifs*** ;
- cette grandeur ***décroît strictement à chaque pas de boucle***.

Proposition 7.2

Si une boucle possède un variant v de boucle, alors elle se termine.

Démonstration

Exemple :

```
>>> def euclide(a,b):
...     """Renvoie le PGCD des entiers a et b."""
...     u,v=a,b
...     while v!=0:
...         u,v=v,u%v
...     return u
...
>>> euclide(210,462)
42
```

Montrer que la variable `v` est un ***variant de boucle***.
En déduire un majorant du nombre de pas de boucles effectués.

II.3. Preuve d'un algorithme

Définition 7.3 (Invariant de boucle)

On appelle ***invariant de boucle*** une grandeur dont la valeur reste constante à l'intérieur d'une même boucle.
En pratique, pour montrer qu'une grandeur est un invariant de boucle, ***on montre seulement qu'elle ne change pas lorsqu'on effectue un pas de boucle***.

La notion d'***invariant de boucle*** permet de prouver qu'un algorithme effectue bien la tâche qu'on attend de lui.
Exemple :
Montrer que dans l'exemple du paragraphe précédent, PGCD(u, v) est un invariant de la boucle `while`.
En déduire que la fonction `euclide(a,b)` renvoie bien le PGCD de a et de b.

III. Complexité d'un algorithme

III.1. Introduction

La ***complexité*** d'un algorithme est une notion visant à donner ***un ordre de grandeur des ressources machine*** qui sont nécessaires à la mise en œuvre de l'algorithme.
La complexité d'un algorithme dépend évidemment, en général, de la taille N de ses paramètres d'entrée. Par exemple, rechercher un élément dans une liste risque de prendre plus de temps si la liste est longue que si la liste est courte.
Par ailleurs, on peut distinguer au moins deux types de complexité :

- la ***complexité en mémoire*** mesure l'espace mémoire utilisé par l'algorithme pour son fonctionnement propre, indépendamment de l'espace mémoire occupé par les paramètres d'entrée ;
- la ***complexité en temps*** mesure le nombre d'opérations élémentaires effectuées par l'algorithme.

III.2. Difficultés à définir la notion de complexité

a) Opération élémentaire

La notion de complexité la plus utilisée est celle de complexité en temps. Comme nous venons de le voir elle s'exprime en ***nombre d'opérations élémentaires***.
La définition de la complexité en temps soulève donc une première question : qu'est-ce qu'une opération élémentaire ?
Une opération élémentaire peut être une addition, une soustraction, une affectation, etc... Concernant la multiplication ou la division, elle devient plus difficile à définir : en effet, on peut expérimenter soit même qu'une multiplication est plus longue à effectuer à la main qu'une addition.
C'est, au moins en partie, également valable en informatique : l'exponentiation (c'est-à-dire le fait d'élever un nombre à une certaine puissance) est plus coûteuse en temps que la multiplication.
Il est possible de mesurer précisément le temps mis par un code python à l'aide des instructions ***IPython*** `time` et `timeit` ou à l'aide de la fonction `clock` du module `time` :

```
>>> import time as t
>>> import math as m
>>> def temps(op):
...     T=t.clock()
...     eval(op)
...     dT=t.clock()-T
...     mdT=dT
...     N=m.floor(0.5/dT)
...     i=0
...     while i<N:
...         T=t.clock()
...         eval(op)
...         dT1=t.clock()-T
...         mdT=min(mdT,dT1)
...         dT+=dT1
```

```
...             i+=1
...         s1=' boucle(s) effectuee(s).\nMeilleur temps (en seconde) : '
...         s2='\nTemps moyen (en seconde) : '
...         print(str(N+1)+s1+format(mdT,'.3e')+s2+format(dT/(N+1),'.3e'))
...
>>> x=7**11
>>> y=3**12
>>> temps('x+y')
20834 boucle(s) effectuee(s).
Meilleur temps (en seconde) : 5.300e-06
Temps moyen (en seconde) : 8.141e-06
>>> temps('x*y')
35461 boucle(s) effectuee(s).
Meilleur temps (en seconde) : 5.200e-06
Temps moyen (en seconde) : 9.325e-06
>>> temps('x**y')
1 boucle(s) effectuee(s).
Meilleur temps (en seconde) : 2.779e+00
Temps moyen (en seconde) : 2.779e+00
```

Même s'il est patent sur l'exemple précédent que les opérations ont des coûts très variés en temps, pour simplifier les choses, nous considèrerons que toutes les opérations arithmétiques ou d'affection sont des opérations élémentaires, ainsi que l'instruction `if` ou les fonctions mathématiques standards. En revanche ***les boucles `for` et `while` n'en sont pas et il convient de bien tenir compte du nombre de pas de boucle effectués pour calculer la complexité en temps***.

b) Complexité dans le meilleur... ou dans le pire des cas ?

```
alea=[r.randrange(1,101) for k in range(30)]
def cherche(l,u):
    n=len(l)
    i=0
    while (i<n)and(l[i]<u):
        i+=1
    return i
print(cherche(alea,90))
```

L'algorithme mettra évidemment moins de temps si l'élément recherché se trouve en première position dans la liste que s'il se trouve en dernière position.
Pour cette raison, on est amené à distinguer deux types de complexité en temps :

- la complexité dans le meilleur des cas ;
- la complexité dans le pire des cas.

Ex. 7.6 Calculer pour une liste `l` à n éléments, la complexité de la fonction `cherche` dans le meilleur et dans le pire des cas.

Cor. 7.6

c) Ordre de grandeur de la complexité

Les remarques précédentes suffisent à se convaincre que la mesure de la complexité n'est pas une science exacte...
Pour cette raison, on ne cherche pas à évaluer de façon précise la complexité en temps d'un algorithme. On en donne un ordre de grandeur en fonction de la taille n des paramètres d'entrée en utilisant la notion de $\mathcal{O}$.

Définition 7.4 (Complexité)

On dira qu'un algorithme est de complexité $\mathcal{O}(f(n))$ s'il existe deux constantes constantes k_1, k_2 réelles telles que ***sa complexité*** $c(n)$ ***dans le pire des cas*** vérifie

$$k_1 f(n) < c(n) < k_2 f(n)$$

où n est la taille des paramètres d'entrée de l'algorithme.

Voici les complexités fréquemment rencontrées :

	Nom courant	Temps pour $n = 10^6$	Remarques
$\mathcal{O}(1)$	temps constant	1ns	Le temps d'exécution ne dépend pas des données traiter. C'est le cas des opérations élémentaires.
$\mathcal{O}(log\ n)$	logarithmique	10ns	En pratique cela correspond à une exécution quasi-instantanée.
$\mathcal{O}(n)$	linéaire	1ms	Le temps d'exécution d'un tel algorithme ne devient supérieur à une minute que pour des données de taille comparable à celles des mémoires vives disponibles actuellement. Le problème de la gestion de mémoire se posera donc avant celui de l'efficacité en temps.
$\mathcal{O}(n^2)$	quadratique	1/4h	Cette complexité reste acceptable pour des données de taille moyenne ($n < 10^6$), mais pas au-delà.
$\mathcal{O}(n^k)$	polynômiale	30 ans si $k = 3$	On peut voir des complexités en $O(n^3)$ voire $O(n^4)$.
$\mathcal{O}(2^n)$	exponentielle	plus de $10^{300\ 000}$ milliards d'années	Un algorithme d'une telle complexité est inutile sauf pour de très petites données ($n < 50$).

Ex. 7.7 Quelle est la complexité de la fonction `cherche` ?

Cor. 7.7

III.3. Exercices

a) Suite de Fibonacci

- Calculer la complexité des deux algorithmes suivant permettant d'obtenir le n-ième terme de la suite de Fibonacci :

```
>>> def f(n):
...     if n<2:
...         return n
...     else:
...         return f(n-1)+f(n-2)
...
>>> def f2(n):
...     a=0
...     b=1
...     i=1
```

```
...     while i<n:
...         a,b=a+b,a
...         i+=1
...     return a
...
```

- On peut améliorer l'algorithme `f` en utilisant une mémoire des valeurs déjà calculées :

```
>>> memoire=[0,1]
>>> def f(n):
...     global memoire
...     if n<len(memoire):
...         return memoire[n]
...     else:
...         memoire.append(f(n-1)+f(n-2))
...         return memoire[-1]
...
```

Donner la complexité dans le pire des cas du nouvel algorithme.

```
>>> temps('f(900)')
652 boucle(s) effectuee(s).
Meilleur temps (en seconde) : 6.100e-06
Temps moyen (en seconde) : 7.442e-06
>>> temps('f2(900)')
6227 boucle(s) effectuee(s).
Meilleur temps (en seconde) : 7.610e-05
Temps moyen (en seconde) : 7.813e-05
```

b) Calcul du PGCD

Voici deux algorithmes permettant d'obtenir le PGCD de deux entiers :

```
>>> def PGCD1(a,b):
...     diviseurs=[k for k in range(1,min(a,b)+1) if a%k==0 and b%k==0]
...     return diviseurs[-1]
...
```

```
>>> def PGCD2(a,b):
...     while b!=0:
...         a,b=b,a%b
...     return a
...
```

1) Calculer la complexité du premier algorithme.
2) Soient u et v deux entiers strictement positifs tels que $u > v$.
   Montrer que le reste r de la division euclidienne de u par v vérifie : $r \leqslant \min(u-v; v-1)$.
3) Soit $m \in \mathbb{R}$ et $f : x \in [0; m] \mapsto \min(m-x; x-1)$. Montrer que f passe par un maximum que l'on précisera.
4) En déduire que la complexité (dans le pire des cas) du second algorithme est en $\mathcal{O}(\ln(\min(a,b)))$.
5) Pour chacun des deux algorithmes :

- choisir aléatoirement 1000 couples $(a;b)$ d'entiers compris entre 10^4 et 10^6 ;
- calculer le PGCD des couples $(a;b)$ ainsi choisis et les temps de calcul réels ;
- tracer les points d'abscisse $\min(a;b)$ et d'ordonnée le temps de calcul précédemment trouvé.

Vérifier que la représentation obtenue est conforme aux complexités calculées.

III.4. Recherche par dichotomie

Ex. 7.8 Importer le module `random` en lui donnant le nom (ou alias) `r`. Créer une liste nommée `alea` contenant 30 entiers tirés aléatoirement à l'aide de la fonction `randrange` du module. Afficher la liste `alea`.

Cor. 7.8

Ex. 7.9 Utiliser la méthode `sort` de cette liste pour la trier. Afficher la liste ainsi triée.

Cor. 7.9

Il est possible de profiter du fait que la liste est triée pour améliorer la complexité de notre algorithme de recherche. Pour cela on utilise la méthode de ***dichotomie*** :

Méthode : Dichotomie

La recherche par ***dichotomie dans une liste triée*** consiste à diviser la liste en deux sous-listes $L = L_1 + L_2$ de même longueur : la liste étant triée, tous les éléments de L_1 sont inférieurs à tous les éléments de L_2.
Si l'élément recherché est supérieur à un élément se trouvant en milieu de liste, on le cherchera à nouveau dans L_2. Sinon, on le cherchera dans L_1.
En utilisant la possibilité offerte par Python d'écrire des fonctions récursives, la programmation d'une telle fonction devient simple et très efficace en terme de complexité.

Ex. 7.10 Écrire une fonction `chercheBis(L,u)` renvoyant l'indice du premier élément de la liste `L` supérieur à `u` si cet élément existe, et renvoyant `len(L)` si aucun élément de la liste n'est supérieur à `u`.

Cor. 7.10

Ex. 7.11 Quelle est la complexité de la fonction `chercheBis(L,u)` ?

Cor. 7.11

Ex. 7.12 Soit f une fonction continue strictement monotone sur un intervalle $I = [a;b]$ et telle que $f(a)f(b) < 0$. Utiliser la méthode de dichotomie pour écrire une fonction `dicho(f,a,b)` permettant d'obtenir une valeur approchée à 10^{-5} près du point $c \in I$ tel que $f(c) = 0$.
En déduire une valeur approchée de l'unique solution de l'équation $e^x + x = 2$.

Cor. 7.12

Ex. 7.13 Modifier votre fonction `dicho(f,a,b,eps=1e-5)` pour qu'elle prenne un paramètre nommé optionnel `eps` donnant la précision souhaitée pour l'approximation obtenue.
En déduire une valeur approchée à 10^{-15} près de l'unique solution de l'équation $e^x + x = 2$.