TP n° 04 – Boucles et complexité

I Initialisation des variables

Algorithme 1 : Exemple d'algorithme ne fonctionnant pas.

```
Données : n un entier naturel
Résultat : ?
pour i de 1 à n faire
    si i est pair alors
        carre ← i²
    sinon
        carre ← 0
    fin
    total ← total + carre
fin
```

1. À votre avis, quel est le résultat attendu par le concepteur de l'algorithme 1 ?
2. Pourquoi ne fonctionne-t-il pas ?
3. Proposer une modification permettant à l'algorithme de répondre à l'attente de son concepteur.
4. Ouvrir un fichier de script dans Spyder. L'enregistrer sous un nom adéquat dans le sous-répertoire adéquat de « Dossiers personnels ». À l'aide de votre cours et des TP précédents, essayer d'implémenter l'algorithme correct en langage Python puis lancer le script pour vérifier qu'il fait ce qui est attendu.

II Choix du type de boucle

On rappelle les règles suivantes :
- Si on connaît à l'avance le nombre de répétitions à effectuer ou, plus généralement, si on veut parcourir une valeur itérable[1], on choisit une boucle *inconditionnelle*, c'est-à-dire en langage Python une boucle `for`.
- À l'inverse, si la décision d'arrêter la boucle ne peut s'exprimer que par un test, on choisit une boucle *conditionnelle*, c'est-à-dire en langage Python une boucle `while`.

Quel type de boucle est adapté à la conception d'algorithmes traitant les problèmes suivants :
- Calculer la valeur absolue d'un nombre donné.
- Calculer la norme d'un vecteur donné.
- Déterminer tous les diviseurs d'un nombre entier donné.
- Déterminer le plus petit diviseur (différent de 1) d'un nombre entier donné.
- Déterminer la date du prochain vendredi 13 connaissant la date d'aujourd'hui.

III Factorielle

On définit la factorielle d'un entier n par les relations : $0! = 1 \quad$ et $\quad (n+1)! = (n+1)n!$
On propose le programme de calcul de $n!$ suivant où n est un entier strictement positif :

```
1 p = 1
2 c = 0
3 for c in range(1,n+1):
4   p= c * p
```

Listing 1: Programme de calcul de la fonction factorielle.

Démontrer à l'aide de l'*invariant de boucle* « $p = c!$ » que le programme est bien *correct*.

1. Par exemple, l'ensemble des entiers générés en Python par la fonction `range()`.

IV Terminaison

```
a = 17
b = 7
while a >= b:
  a = a - b
```

Listing 2: Reprise de l'exercice 5 du TP n° 3.

1. Notez à chaque étape la valeur de `a` et de `b`. Quelle est la valeur finale de `a` ? En général, que vaut `a` à la fin du programme ?
2. Prouver que la boucle *termine* en identifiant un *variant de boucle*.

V Complexités

On suppose que n a été affecté (et vaut un entier strictement positif). Combien d'*opérations élémentaires* les programmes suivant vont-ils exécuter ?

```
s = 0
for i in range(n):
  s = s + i**2
```

```
s = 0
for i in range(n):
  for j in range(i,n):
    s = s + j**2
```

```
while n < 10**10:
  n = n * 2
```

VI Comparaison entre deux méthodes d'exponentiation

On propose deux programmes différents pour l'exponentiation d'un réel positif k par un entier strictement positif n (c'est-à-dire le calcul de k^n). On suppose que k est affecté d'une valeur réelle positive.

```
p = 1
c = n
while c > 0:
  p = p * k
  c = c - 1
```

Listing 3: Première méthode d'exponentiation.

```
p = 1
c = n
while c > 0:
  if c%2 == 1:
    p = p * k
  k = k**2
  c = c//2
```

Listing 4: Deuxième méthode d'exponentiation.

1. On prend $n = 13$. Pour chacun des deux programmes 3 et 4, déterminer et noter les valeurs ou expressions successives de p, c et k ainsi que le nombre d'itérations de la boucle Tant que.

2. Compter le nombre d'opérations élémentaires dans chacune des boucles. En déduire lequel des deux programmes réalise une « exponentiation rapide ».
3. On peut montrer que la *complexité temps* du programme d'exponentiation rapide est $O(\log(n))$. Quelle est la *complexité temps* de l'autre programme ?
4. Si vous avez avancé assez rapidement dans le TP et qu'il vous reste du temps, vous pouvez créer grâce à Spyder un script python vous donnant une évaluation du temps d'exécution des deux programmes pour différentes valeurs de n et k grâce au code fourni en Annexe.

VII Recherche par dichotomie

Algorithme 2 : Recherche par dichotomie.

Données : L[1...n] une liste triée de n éléments numérotés de 1 à n inclus et a un élément
Résultat : Un booléen

```
1  g ← 1 ;
2  d ← n ;
3  tant que d − g > 0 faire
4  |  m ← ⌊(d+g)/2⌋ ;                          /* partie entière de (d+g)/2 */
5  |  si L[m] < a alors
6  |  |  g ← m + 1
7  |  sinon
8  |  |  d ← m
9  |  fin
10 fin
11 si L[g] == a alors
12 |  retourner Vrai
13 sinon
14 |  retourner Faux
15 fin
```

On rappelle la définition de la partie entière :
pour tout réel x, c'est l'entier $\lfloor x \rfloor$ tel que $x - 1 < \lfloor x \rfloor \leq x$.
Ainsi, à la ligne 4, on a :

$$\frac{d+g}{2} - 1 < m \leq \frac{d+g}{2}$$

Prouver que l'algorithme *termine*.

VIII Recherche du minimum d'une liste de valeurs numériques

Algorithme 3 : Fonction minimum(L).

Données : Une liste L contenant n éléments numérotés de 1 à n inclus ($L[1...n]$)
Résultat : Le plus petit élément de la liste

```
1 mini ← 0 ;
2 pour i de 1 à n faire
3 |  si L[i] < mini alors
4 |  |  mini ← L[i]
5 |  fin
6 fin
7 retourner mini
```

Algorithme 4 : Fonction minimum(L).

Données : Une liste L contenant n éléments numérotés de 1 à n inclus ($L[1...n]$)
Résultat : Le plus petit élément de la liste

```
1 mini ← L[1] ;
2 pour i de 1 à n faire
3 |  si L[i] < mini alors
4 |  |  mini ← L[i]
5 |  fin
6 fin
7 retourner mini
```

1. Identifier la différence entre les algorithmes 3 et 4 et expliquer pourquoi le premier n'est pas toujours *correct* (dans certains cas, il ne donne pas le minimum de la liste).
2. Comment peut-on améliorer très légèrement l'algorithme qui est toujours correct en évitant une étape de la boucle Pour ?
3. Réécrire l'algorithme en utilisant une boucle Tant que et prouver qu'il *termine* en déterminant son *variant de boucle*. Si vous ne trouvez pas rapidement comment réécrire l'algorithme vous pouvez consulter la réponse en Annexe à l'algorithme 5.
4. Soit la proposition suivante : « mini est le minimum de toutes les valeurs de la liste numérotées de 1 à $i-1$ inclus » ou encore « $\{mini = minimum(L[1...i-1])\}$ ». Vérifier que l'algorithme est *correct* en prouvant que la proposition précédente est bien un *invariant de boucle*.

Annexe

Annexe de l'exercice VI

```
import time
def expo_un(n,k):
    a=time.clock()
    p = 1
    c = n
    while c > 0:
        p = p * k
        c = c - 1
    b=time.clock()-a
    return b
def expo_deux(n,k):
    a=time.clock()
    p = 1
    c = n
    while c > 0:
        if c%2 == 1:
            p = p * k
        k = k**2
        c = c//2
    b=time.clock()-a
    return b
n=6000
k=4
print(expo_un(n,k))
print(expo_deux(n,k))
```

Annexe de l'exercice VIII

Algorithme 5 : Fonction minimum(L).

Données : Une liste L contenant n éléments numérotés de 1 à n ($L[1...n]$)
Résultat : Le plus petit élément de la liste

```
1 mini ← L[1] ;
2 i ← 2 ;
3 tant que i ≤ n faire
4 |   si L[i] < mini alors
5 |   |   mini ← L[i]
6 |   fin
7 |   i ← i + 1
8 fin
9 retourner mini
```

Correction TP n° 04 – Boucles et complexité

I Initialisation des variables

1. Somme des carrés des entiers pairs compris entre 1 et n inclus.
2. Parce que la variable `total` n'a pas été initialisée et que la ligne 7 ne peut donc pas être effectuée.
3. On ajoute une ligne d'initialisation avant la ligne 1 :
   total $\leftarrow 0$.
4. Implémentation de l'algorithme :

```
1 n = int(input(n))
2 total = 0
3 for i in range(1,n+1):
4   if i%2 == 0:
5     carre = i**2
6   else:
7     carre = 0
8   total = total + carre
```

II Choix du type de boucle

— Pas besoin de boucle !
— Boucle `for` : il suffit de parcourir les composantes du vecteur qui sont en nombre bien déterminé.
— Boucle `for` : tous les diviseurs d'un nombre entier n donné sont compris entre 1 et n ; le nombre d'étapes de calcul est donc facile à déterminer.
— Boucle `while` : il n'est pas nécessaire de parcourir la totalité des diviseurs de n, il suffit de déterminer le premier des diviseurs et on peut arrêter la boucle dès qu'il est rencontré, ce qu'on traduit dans la condition de la boucle.
— Boucle `while` : sauf à entreprendre un très lourd travail préparatoire, il n'est pas possible d'anticiper combien d'étape de calcul il faudra mettre en œuvre pour déterminer cette date.

III Factorielle

— L'invariant de boucle est la proposition « p=c ! ».
— À l'initialisation, on a : $p = 1$, $c = 0$ et $1 = 0!$, la proposition est donc vraie.
— À la première itération dans la boucle : $c = 1$, $p = c * p = 1 * 1 = 1$ et $1 = 1!$, la proposition est donc vraie.
— On suppose donc $p_i = c_i!$ vraie. À la (i+1)^e itération : $c_{i+1} = c_i + 1$, $p_{i+1} = c_{i+1} * p_i = (c_i + 1) * c_i! = c_{i+1}!$. La proposition est donc toujours vraie dans la boucle.
— À la terminaison de la boucle, on a $c = n$ et donc finalement $p = n!$ ce qui prouve la correction de l'algorithme.

IV Terminaison

1. $(a = 17, b = 7)$; $(a = 10, b = 7)$; $(a = 3, b = 7)$. La valeur finale de a est 3. Le programme renvoie le reste de la division euclidienne de a par b (`a % b` en python).
2. — Un variant de boucle est $a - b$. Initialement $a = 17$ et $b = 7$ donc $a \geq b$ et la boucle commence.
   — On a comme condition de boucle $a \geq b$ donc on a toujours $a - b \geq 0$ dans la boucle.
   — D'autre part, si on note $(a-b)_+$ la valeur de $(a-b)$ à la fin d'une itération et $(a-b)_0$ sa valeur au début d'une itération, on a $(a-b)_+ = (a-b)_0 - b < (a-b)_0$ car $b > 0$.
   $(a - b)$ est donc une variable strictement décroissante dans la boucle.
   — $(a - b)$ est donc positif et strictement décroissant dans la boucle : la boucle termine.

V Complexités

Premier programme :

- une affectation avant la boucle ;
- n itérations de la boucle puisque i parcourt par incrément de 1 l'intervalle $[\![0, (n-1)]\!]$;
  - trois opérations par itération de la boucle : une affectation, une somme, une mise au carré ;
- on a donc le nombre total d'opérations : $N_{op} = 1 + 3\,n$;
- complexité linéaire $O(n)$.

Deuxième programme :

- une affectation avant la boucle ;
- n itérations de $i = 0$ jusqu'à $i = n-1$ de la première boucle ;
  - $(n-i)$ itérations de la deuxième boucle ;
    - trois opérations par itération de la deuxième boucle : une affectation, une somme, une mise au carré ;
- on a donc le nombre total d'opérations : $N_{op} = 1 + \sum_{i=0}^{n-1} (3\,(n-i)) = 1 + \frac{3}{2}\,n + \frac{3}{2}\,n^2$ car $\sum_{i=0}^{n-1} (3\,(n-i)) = 3\,n \times n - 3\,\frac{(n-1)n}{2}$.
- complexité quadratique $O(n^2)$.

Troisième programme :

- trois opérations par itération de la boucle : un test, une affectation, une multiplication ;
- on cherche le nombre k d'itérations telles que le critère de la boucle `while` soit rempli : à la k^{e} itération le test est $2^{(k-1)}\,n < 10^{10}$. On a donc $k < 1 + \log_2(10^{10}) + \log_2(\frac{1}{n})$; dans le pire des cas, $n = 1$ donc $\log_2(\frac{1}{n}) = 0$ et on a $1 + \lfloor \log_2(10^{10}) \rfloor = 34$ itérations ;
- au total, on a dans le pire des cas 102 opérations.

VI Comparaison entre deux méthodes d'exponentiation

1. Premier programme (à gauche) : 13 itérations de boucle.
   Deuxième programme (à droite) : 4 itérations de boucle.

	p	c	k	p	c	k
Initialisation	1	13	k	1	13	k
Itération 1	k	12	k	k	6	k^2
Itération 2	k^2	11	k	k	3	k^4
Itération 3	k^3	10	k	k^5	1	k^8
Itération 4	k^4	9	k	k^{13}	0	k^{16}
Itération 5	k^5	8	k			
Itération 6	k^6	7	k			
Itération 7	k^7	6	k			
Itération 8	k^8	5	k			
Itération 9	k^9	4	k			
Itération 10	k^{10}	3	k			
Itération 11	k^{11}	2	k			
Itération 12	k^{12}	1	k			
Itération 13	k^{13}	0	k			

2. Premier programme : 2+5x13=67 opérations. Deuxième programme : au maximum 9 opérations par itération et seulement 7 si le critère de l'instruction `if` n'est pas respecté ; d'où 2+9x3+7=36 opérations, puisque l'opération p=p*k n'est faite qu'à la première, la troisième et la quatrième itérations. Le deuxième programme semble donc plus rapide.
3. On trouve pour le premier programme $C_t = 2\,C_e + 5\,n\,C_e$. Complexité temps linéaire $O(n)$.

VII Recherche par dichotomie

Un variant de boucle est $d-g$.
Initialement : $d=n>1$, $g=1$ et donc $d-g>0$. La boucle commence.
On note $(d-g)_+$ la valeur de $(d-g)$ à la fin d'une itération. On a, d'après la définition d'un partie entière, $\frac{d+g}{2}-1<m\leq\frac{d+g}{2}$ et deux cas se présentent :

— Si $L[m]<a$ alors $g=m+1$ et $(d-g)_+=d-(m+1)$ donc $\frac{d-g}{2}-1<(d-g)_+=d-(m+1)<\frac{d-g}{2}$.

— Si $L[m]\geq a$ alors $d=m$ et $(d-g)_+=m-g$ donc $\frac{d-g}{2}-1<(d-g)_+=m-g\leq\frac{d-g}{2}$.

Dans les deux cas : $(d-g)_+<\frac{d-g}{2}$. Étant donné que dans la boucle on a toujours $d-g>0$ et on en conclut que $(d-g)_+<d-g$ et donc que $(d-g)$ est strictement décroissante ce qui prouve la terminaison.

VIII Recherche du minimum d'une liste de valeurs numériques

1. Les initialisations aux lignes 1 des algorithmes sont différentes. Le premier algorithme n'est pas correct car si la liste ne contient que des éléments strictement positif, l'algorithme renvoie comme minimum la valeur 0, ce qui est faux. Le premier algorithme n'est correct que si la liste contient au moins une valeur négative ou nulle.
2. À la ligne 1, on définit le minimum comme étant *a priori* le terme $L[1]$. Il est donc inutile de le tester dans la boucle. On peut donc commencer la boucle par la valeur $i=2$.
3. Donné en Annexe de l'énoncé.
4. À l'initialisation $i=2$, $mini=L[1]$ et $L[1...i-1]=L[1...1]$ ne contient que la valeur $L[1]$ qui en est donc *de facto* le minimum. La proposition est donc vérifiée avant la première itération.

   Au début de la première itération, $i=2$ et $mini=L[1]$. Deux cas se présentent :

   — Si $L[2]<mini=L[1]$ alors $mini=L[2]$, puis on a $i=2+1=3$ et $mini$ est bien le minimum de $L[1...i-1]=L[1...2]$ car $mini=L[2]<L[1]$. La proposition est donc vérifiée en fin d'itération.

   — Si $L[2]\geq mini=L[1]$ alors $mini=L[1]$, puis on a $i=2+1=3$ et $mini$ est bien le minimum de $L[1...i-1]=L[1...2]$ car $mini=L[1]\leq L[2]$. La proposition est donc vérifiée en fin d'itération.

   On suppose donc la proposition vérifiée au début de la i$^{\text{e}}$ itération : $mini=minimum(L[1...i-1])$.
   Deux cas se présentent :

   — Si $L[i]<mini=minimum(L[1...i-1])$ alors $mini=L[i]$ et, à la fin de la i$^{\text{e}}$ itération, $mini$ est bien le minimum de $L[1...(i+1)-1]$ car $mini=L[i]<minimum(L[1...i-1])$. La proposition est donc vérifiée donc au début de la (i+1)$^{\text{e}}$ itération.

   — Si $L[i]\geq mini=minimum(L[1...i-1])$ alors $mini=minimum(L[1...i-1])$ et, à la fin de la i$^{\text{e}}$ itération, $mini$ est bien le minimum de $L[1...(i+1)-1]$ car $L[i]\geq mini=minimum(L[1...i-1])$. La proposition est donc vérifiée au début de la (i+1)$^{\text{e}}$ itération.

   La proposition est donc tout le temps vérifiée dans la boucle.

   À la terminaison de la boucle (qui existe car $n-i$ est un variant de boucle évident), i vaut $n+1$ et on a donc bien $mini=minimum(L[1...n])$, ce qui prouve la correction de l'algorithme.

   Remarque : plus rigoureusement l'invariant de boucle est « $\{mini=minimum(L[1...i-1])\}$ et $\{i-1\leq n\}$ » pour s'assurer qu'on ne dépasse pas la taille de la liste en cherchant à accéder à l'élément $L[n+1]$ qui n'existe pas puisque l'index de la liste va de 1 à n.