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 IDLE. 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 IDLE 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
```