P7 - Utilisation avancée des boucles

Dans ce chapitre, on approfondit l'étude des boucles `for` et `while` et on découdre la notion de complexité d'un algorithme.

1 Parcours d'un tableau par valeurs

Pour l'instant, lorsqu'on voulait parcourir un tableau (c'est à traiter successivement toute ou partie de ses valeur), on utilisait une boucle for. Par exemple :

```python
def somme(t):
    res = 0
    for i in range(len(t)):
        res += t[i]
    return res
```

On dit ici qu'on parcourt le tableau par indice. En effet, la variable `i` étant modifiée à chaque tour de la boucle représente les différents indices du tableau (de `0` à `len(t)-1`). Il existe en python une autre méthode pour effectuer ce parcours :

```python
def somme(t):
    res = 0
    for x in t:
        res += x
    return res
```

On parle ici de parcours par valeur. Cette fois, la variable `x` ne représente plus des indices mais directement les différentes valeurs du tableau (de `t[0]` à `t[len(t)-1]`).

2 Boucles imbriquées

On parle de boucles imbriquées lorsque dans une boucle, dite principale, on effectue une autre boucle, dite secondaire. Voyons quelques exemples de boucles imbriquées :

```python
n=10
for i in range(n):
    for j in range(n):
        #boucles simplement imbriquees
        print(i,j)

n=10
for i in range(n):
    for j in range(i):
        #boucles imbriquees avec j<i
        print(i,j)

```

```python
n=10
for i in range(n):
    for j in range(i, n):
        #boucles imbriquees avec j>=i
        print(i,j)

n=10
for i in range(n):
    for j in range(n):
        for k in range(n):
            #trois boucles imbriquees simplement
            print(i, j, k)
```

Recopier et expliquer les comportement des ces différentes boucles. Noter le nombre d'affichage de chaque programme.

3 Complexité

La complexité d'un programme (ou d'un algorithme) est une mesure du nombre d'opérations effectuée par celui-ci pour un entrée de taille donnée. Considérons les fonctions suivantes :

```python
def f1(n):
    a = 0
    for i in range(n):
        a = a+1
    return a


def f2(n):
    a = 0
    for i in range(n):
        for j in range(n):
            a = a+1
    return a

```

```python
def f3(n):
    a = 0
    for i in range(n):
        for j in range(n):
            for k in range(n):
                a = a+1
    return a

def f4(n):
    a = 0
    while n > 0:
        n = n//2
        a = a+1
    return a
```

Chacune de ces fonctions renvoient exactement le nombre de tours effectuées par les différentes boucles les composant, résumons quelques résultats dans un tableau :

`n`	`f1(n)`	`f2(n)`	`f3(n)`	`f4(n)`
10	10	100	1000	4
100	100	10 000	1 000 000	7
1 000	1 000	1 000 000	1 000 000 000	10
n	n	n^2	n^3	$\log_2(n)^*$

* : $\log_2(n)$ est l'opération "inverse" de 2^n, c'est à dire par exemple que $\log_2(4096) = 12$ puisque $2^{12} = 4096$.

La dernière ligne du tableau désigne le nombre d'opérations effectués **en fonction de n**. Voyons quelques points de vocabulaire :

— Un algorithme dont le nombre de d'opérations est environ proportionnel (pour plus de précision sur cette notion, chercher *notations de Landau*) à n est dit **linéaire**, c'est le cas généralement des programmes ne faisant intervenir que des boucles simples.

— Un algorithme dont le nombre de d'opérations est environ proportionnel à n^2 est dit **quadratique**, c'est le cas généralement des programmes faisant intervenir des doubles boucles imbriquées.

— Un algorithme dont le nombre de d'opérations est environ proportionnel à n^3 est dit **cubique**, c'est le cas généralement des programmes faisant intervenir des triples boucles imbriquées.

— Un algorithme dont le nombre de d'opérations est environ proportionnel à $\log_2(n)$ est dit **logarithmique**.

Comme on peut le voir dans le tableau, pour effectuer la même tâche, on préférera un algorithme logarithmique à un algorithme linéaire, à un quadratique, à un cubique, etc.