Mathématiques supérieures le 13 Mai 2024

Correction du Devoir Surveillé n°6

Ce sujet est composé de 5 exercices. On prendra soin de bien justifier ses calculs.
Veillez à bien respecter l'indentation des programmes Python que vous écrirez sur votre copie.

Exercice 1.

1. Écrire une fonction **itérative** `serieIt(n)` qui renvoie la somme $\sum_{k=1}^{n} \frac{1}{k^2}$.
2. Écrire une fonction **récursive** `serieRec(n)` qui renvoie la somme $\sum_{k=1}^{n} \frac{1}{k^2}$.

Correction.

1. Version itérative :

```
def somme_inv_carres(n):
    S=0
    for i in range(1,n+1):
        S+=1/i**2
    return S
```

2. Version récursive :

```
def somme_inv_carres_rec(n):
    if n==1:
        return 1
    else:
        return somme_inv_carres_rec(n-1)+1/n**2
```

Exercice 2.

Montrer que les algorithmes suivants se terminent et déterminer leurs complexités temporelles en terme d'opérations arithmétiques *(le détail des calculs du coût doit figurer sur la copie)*.

Pour chacun des algorithmes, on suppose que l'argument `n` est un entier positif.

1.
```python
#Algorithme 1
def f1(n):
    t = 6
    for i in range(n):
        for j in range(20):
            t=3*t-1
    return t
```

2.
```python
#Algorithme 2
def f2(n):
    z = 1
    while z<=n:
        z=2*z
    return z
```

3.
```python
#Algorithme 3
def f3(n):
    if n==0:
        return 1
    else:
        return f3(n-1)+2
```

Correction.

1. On a affaire à deux boucles `for` imbriquées dont le nombre d'itérations est déterminé dès le début de l'algorithme donc il se termine.

   Le coût $c_1(n)$ de `f1(n)` est donné par :

   $$c_1(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{19} 2 = \sum_{i=0}^{n-1} 40 = 40n$$

   car :

   — dans la boucle `for` b_1 dépendante de `j`, pour chaque `j`, il y a 2 opération ;

   — puis on somme pour `j` allant de 0 à $20-1=19$. pour trouver son coût $c_{b_1} = \sum_{j=0}^{19} 2 = 40$.

   — dans la boucle `for` b_2 dépendante de `i`, pour chaque `i`, il y a c_{b_1} opérations ;

   — puis on somme pour `i` allant de 0 à $n-1$.

   Par suite, la complexité est $C_1(n) = O(n)$ (complexité linéaire).

2. On remarque que $v = n - z$ est un variant de boucle donc l'algorithme se termine. En effet, la variable z est initialisée à 1 et double à chaque itération donc z est à valeurs entières ≥ 1 et comme n est entier, la quantité $v = n - z$ est un entier.
   - Avant la boucle, si $n < 1$ alors la boucle n'est pas initialisée donc l'algorithme se termine et si $n \geq 1 = z$, la boucle est initialisée et on a $v = n - z = n - 1$ est un entier positif.
   - Au début de chaque itération, la condition $z \leq n$ étant vraie, on a $v = n - z$ positif. Ainsi, au début de chaque itération, v est un entier positif.
   - La quantité valant $n - z$ au début d'une itération, à la fin de cette même itération, on a $v = n - 2z < n - z$ car $z \geq 1$ donc v décroît strictement à chaque itération.

   Ainsi, v est bien un variant de boucle.

   Déterminons la complexité de cet algorithme. On considère le nombre d'itérations N de la boucle `while`. Le coût $c_2(n)$ de `f2(n)` est alors donné par :

$$c_2(n) = \sum_{k=1}^{N} 1 = N.$$

   car on a une opération à chaque tour de boucle donnée par le * du `2*z`

   Maintenant, encadrons le nombre N de d'itérations en fonction de `n`.

   On considère la suite $(z_k)_{k\in\mathbb{N}}$ telle que z_0 désigne la valeur initiale de la variable `z` et pour chaque $k \geq 1$, z_k désigne la valeur de la variable `z` à la fin de la k-ième itération.
   Ainsi, cette suite vérifie :

$$\begin{cases} z_0 = 1 \\ z_{k+1} = 2z_k \quad \forall k \in \mathbb{N}^* \end{cases}$$

   Il s'agit d'une suite géométrique et sa forme explicite est donnée par $z_k = 2^k$ pour tout $k \in \mathbb{N}$.

   Comme N est le numéro de la dernière itération et donc du dernier tour où la condition "`z<=n`" est vérifiée, on a :

$$\begin{cases} 2^N = & z_N \leq n \\ 2^{N+1} = & z_{N+1} > n \end{cases}$$

   d'où

$$\log_2(n) - 1 < N \leq \log_2(n).$$

   Il en résulte que :

$$\log_2(n) - 1 < c_2(n) = N \leq \log_2(n);$$

   et ainsi, la complexité est $C_2(n) = \log_2(n)$ (complexité logarithmique) car on a $c_2(n) \underset{n\to+\infty}{\sim} \log_2(n)$.

3. On considère la quantité $a = n$ où n est la valeur de l'argument de `f3` à chaque appel récursif. Lors de l'appel initial `f3(n)`, a vaut n qui est un entier positif par hypothèse.
   - Si $n = 0$, la fonction renvoie 1 donc l'algorithme se termine.
   - Si $n > 0$, alors comme n est un entier, $n \geq 1$. Ainsi, lors de l'appel récursif `f3(n-1)`, on a $a = n - 1 \geq 0$ et $a = n - 1 < n$ qui est la valeur précédente de a. Ainsi a est une quantité entière positive qui décroit strictement à chaque appel récursif.

Par suite, a est un variant récursif et donc l'algorithme se termine.
Déterminons sa complexité. On note $c(n)$ le coût de `f3(n)`. Alors on a :

$$\begin{cases} c(0) = 0 \\ c(n) = c(n-1) + 2 \quad \text{si } n \geq 1 \end{cases}$$

Ainsi, $c(n) = 2n$. Il en résulte que la complexité de `f3` est $C(n) = O(n)$ (linéaire).

Exercice 3.

On considère l'algorithme suivant qui est censé calculer, pour un argument n entier naturel, le terme u_n de la suite $(u_k)_{k\in\mathbb{N}}$ telle que, pour tout $k \in \mathbb{N}$, $u_k = 2^k - 1$.

```
1 def suite(n):
2     u=0
3     for i in range(1,n+1): #i va de 1 à n
4         u=2*u+1
5     return u
```

1. Montrer que la propriété $P(i)$ ="à la fin de l'itération i, la variable `u` vaut $2^i - 1$" est un invariant de boucle pour cet algorithme.
2. En déduire que l'algorithme est correct.

Correction.

1. Montrons que P est un invariant de boucle :
   - **Initialisation :** Montrons $P(1)$. La variable `u` est initialisée à 0, donc au début de l'itération 1, `u` vaut 0. Ensuite, on calcule la valeur `2*u+1`$= 2 \times 0 + 1 = 1$ que l'on affecte à `u`. Donc à la fin de l'itération 1, `u` vaut $1 = 2^1 - 1$. Ainsi, $P(1)$ est vérifié.
   - **Transmission :** Soit $i \geq 1$. On suppose que $P(i)$ est vérifiée. Alors au début de l'itération $i+1$, la variable `u` garde la valeur qu'elle avait à la fin de la précédente, à savoir $2^i - 1$ (d'après $P(i)$).
     On calcule ensuite la valeur `2*u+1`$= 2 \times (2^i - 1) + 1 = 2^{i+1} - 1$ que l'on affecte à `u`. Donc à la fin de l'itération $i+1$, `u` vaut $2^{i+1} - 1$. Ainsi, $P(i+1)$ est vérifié.

   Par suite, P est un invariant de boucle.
2. Comme P est un invariant de boucle, alors, pour un argument entier $n \geq 1$, $P(n)$ est vérifiée i.e. à la fin de l'itération n, la variable `u` vaut $2^n - 1$. Or comme il s'agit de la dernière itération de la boucle `for`, c'est cette valeur qui est renvoyée par la fonction. Ainsi, `suite(n)`$= 2^n - 1 = u_n$.
   On remarque également que si `n`$= 0$, `suite(n)`$= 0 = u_0$.
   Donc l'algorithme est correct.

Exercice 4.

Pour chacune des propositions suivantes, écrire une programme simple d'argument `n` qui possède une complexité en :

1. $O(n)$,
2. $O(n^2)$,
3. $O(n^3)$,
4. $O(2^n)$,
5. $O(\log(n))$.

Exercice 5.

On considère les algorithmes suivants :

— Se terminent-t-ils ?

— Si oui, déterminer leurs coûts puis leurs complexités.

```python
#Algo1
def f1(n):
    s = 0
    while s < n:
        s = s + 2
```

```python
#Algo2
def f2(n):
    r = 1
    while r <= 2*n:
        r = 3 * r
```

```python
#Algo3
def f3(n):
    a=5
    while a >= 0:
        a = a**2+4*a+4
```

```python
#Algo4
def f4(n):
    a = 1
    while a < n:
        a = 3*a-1
```

```python
#Algo5
def f5(n):
    u = 1
    s = 0
    while u <= n:
        for i in range(u):
            s=s+5
        u = u+2
```

Correction.

1.