Correction du devoir 1 d'informatique

Exercice 1

1. Notons $\mathcal{P}(k)$ l'assertion : $P_k = a^k$ et $i_k = k$ et montrons que ceci définit un invariant de boucle.

Comme $P_0 = 1$ et $i_0 = 0$, l'assertion $\mathcal{P}(0)$ est vraie.

Supposons que $\mathcal{P}(k)$ est vraie.

On a

$$P_{k+1} = P_k * a = a^k \times a = a^{k+1}$$

Puis

$$i_{k+1} = i_k + 1 = k + 1$$

Donc $\mathcal{P}(k+1)$ est vraie.

$P_k = a^k$ et $i_k = k$ est un invariant de boucle.

2. On peut construire le tableau suivant.

k	0	1	2	3	4	$\cdots$	Dernière boucle
P_k	1	a	a^2	a^3	a^4	$\cdots$	a^n
i_k	0	1	2	3	5	$\cdots$	n

On souhaite que la fonction retourne a^n. On doit donc s'arrêter quand $P_k = a^n$. Compte-tenu de l'invariant de boucle, on a $k = n$. Donc la boucle doit s'arrêter quand $i_n = n$.

Réponse : **while** i<=n-1.

Exercice 2

1. **a.** Proposition de script :

```
def factorielle(n):
    p=1
    i=1
    while i<=n:
        p = p*i
        i = i+1
    return p
```

b. **Terminaison**

La suite $(n - i)$ est une suite **d'entiers positifs** et **strictement décroissante**. Donc au bout d'un certain nombre de tours de boucle, on aura $n - i < 0$. Compte-tenu du test d'arrêt, ceci justifie la terminaison de l'algorithme.

c. Invariant de boucle

Notons p_k et i_k les valeurs prises par p et i après la k-iéme boucle, avec la convention que p_0 et i_0 sont les valeurs prises par p et i avant la première boucle.

On peut construire le tableau suivant.

k	0	1	2	3	4	$\cdots$	Dernière boucle
p_k	1	1	2	6	24	$\cdots$	
i_k	1	2	3	4	5	$\cdots$	$n+1$

Notons $\mathcal{P}(k)$ l'assertion : $\boxed{p_k = k! \text{ et } i_k = k+1}$ et montrons que ceci définit un invariant de boucle.

Comme $p_0 = 1$ et $i_0 = 1$, l'assertion $\mathcal{P}(0)$ est vraie.

Supposons que $\mathcal{P}(k)$ est vraie.

On a

$$p_{k+1} = p_k * i_k = k! \times (k+1) = (k+1)!$$

Puis

$$i_{k+1} = i_k + 1 = k + 2$$

Donc $\mathcal{P}(k+1)$ est vraie.

Correction

Compte-tenu du test d'arrêt, la valeur prise par i_k à la dernière boucle est $n+1$. Donc à la sortie de la dernière boucle, on a $i_k = n+1$, donc $k = n$ (on a donc n tours de boucle). Et à la sortie de la dernière boucle, on a $p_n = n!$. D'où la correction de l'algorithme.

2. Proposition de script :

```
def estdivisible(n):
    if factorielle(n)%(n+1)==0:
        return True
    else:
        return False
```

3. a. En notant s_k la valeur de s à la sortie de la $k-$ième boucle, et en notant factorielle(k) la valeur prise par la fonction factorielle pour l'entier k, on peut construire le tableau suivant.

k	1	2	3	4
factorielle(k)	1	2	6	24
s_k	0+1 = 1	1+2 = 3	3+6 = 9	9+24 = 33

Pour $n = 4$, on a `k in range(1,5)`, donc l'indice de boucle k varie entre 1 et 4, donc `mystere`(4) renvoie la valeur $\boxed{33}$.

b. Quand on appelle `mystere`(n), on effectue n tours de boucle. À chaque tour de boucle, le nombre de multiplications est égale au nombre de multiplications pour effectuer `factorielle`(k).

Or à l'appel de `factorielle`(k), l'algorithe `factorielle` effectue k tours de boucle, et à chaque tour

de boucle 1 multiplication. Donc à l'appel de `factorielle`(k), on a k multiplications.

Il s'ensuit que le nombre de multiplications qu'effectue `mystere`(n) est égal à : $\sum_{k=1}^{n} k = \frac{n\,(n+1)}{2}$.

> Le nombre de multiplications est $\frac{n\,(n+1)}{2}$. La complexité est en $O(n^2)$.

c. Proposition de script :

```
def mystere(n):
    s=0
    p=1
    for k in range(1,n+1):
        p = p*k
        s = s + p
    return s
```

On a en fait inclus le calcul de la factorielle dans l'algorithme.

Au début on a 2 affectations. Ensuite à chaque tour de boucle, on a 2 opérations et 2 affectations. Donc au total, comme on a n tours de boucle, le nombre d'opérations est égal à $2+4\,n$. D'où la complexité linéaire.

Exercice 3

1. On obtient le tableau suivant :

i	0	1	2	3	4	5	6
p_i	1	2	2	2	2^9	2^9	2^{41}
A_i	2	2^2	2^4	2^8	2^{16}	2^{32}	2^{64}
k_i	41	20	10	5	2	1	0

2. Notons $\mathcal{P}(k)$ l'assertion : $p_i\,A_i^{k_i} = 2^n$ et montrons que ceci définit un invariant de boucle.

Comme $p_0 = 1,\ A_0 = 2$ et $k_0 = n$, l'assertion $\mathcal{P}(0)$ est vraie.

Supposons que $\mathcal{P}(k)$ est vraie.

On a deux cas suivant la parité de k_i.

- Si k_i est pair.

  On a alors

  $$p_{i+1} = p_i,\ A_{i+1} = A_i^2 \quad \text{et} \quad k_i = 2\,k_{i+1}$$

  Donc

  $$p_{i+1}\,A_{i+1}^{k_{i+1}} = p_i\,A_i^{2\,k_{i+1}} = p_i\,A_i^{k_i} = 2^n$$

- Si k_i est impair.

  On a alors

  $$p_{i+1} = p_i\,A_i,\ A_{i+1} = A_i^2 \quad \text{et} \quad k_i = 2\,k_{i+1} + 1$$

  Donc

  $$p_{i+1}\,A_{i+1}^{k_{i+1}} = p_i\,A_i\,A_i^{2\,k_{i+1}} = p_i\,A_i^{2\,k_{i+1}+1} p_i\,A_i^{k_i} = 2^n$$

Donc $\mathcal{P}(k+1)$ est vraie.

$$\boxed{p_i\, A_i^{k_i} = 2^n \text{ est un invariant de boucle.}}$$

3. La boucle s'arrête quand $k_i <= 0$. Comme c'est un entier, elle s'arrête quand $k_i = 0$. Donc, d'après l'invariant de boucle, la fonction retourne $p_i = 2^n$.

$$\boxed{\text{Cette fonction retourne } 2^n.}$$

4. Soit $n = a_0 + a_1\,2 + a_2\,2^2 + a_3\,2^3 + \ldots + a_{r-1}\,2^{r-1} + a_r\,2^r$, où $a_i \in \{0,1\}$ et $a_r \neq 0$.
On a $k_0 = n$, puis $k_1 = a_1 + a_2\,2 + a_3\,2^2 + \ldots + a_{r-1}\,2^{r-2} + a_r\,2^{r-1}$, $k_2 = a_2 + a_3\,2 + a_4\,2^2 + \ldots + a_{r-1}\,2^{r-3} + a_r\,2^{r-2}$.
On peut montrer par récurrence que

$$k_i = a_i + a_{i+1}\,2 + a_{i+2}\,2^2 + \ldots + a_{r-1}\,2^{r-i-1} + a_r\,2^{r-i}$$

On a donc $k_r = a_r\,2^{r-r} = a_r = 1$. La boucle s'arrête donc pour $k_{r+1} = 0$. On a donc $r+1$ tours de boucle.
Par exemple $41 = 1 + 0\,2 + 0\,2^2 + 1\,2^3 + 0\,2^4 + 1\,2^5$, et on a 6 tours de boucle.
Comme $a_i \in \{0,1\}$ et $a_r \neq 0$, on a

$$2^r \leqslant n \leqslant 1 + 1\,2 + 1\,2^2 + 1\,2^3 + \ldots + 1\,2^{r-1} + 1\,2^r$$

Soit

$$2^r \leqslant n \leqslant \frac{1-2^{r+1}}{1-2} = 2^{r+1} - 1 < 2^{r+1}$$

On en déduit que

$$r\ \ln 2 \leqslant \ln n \leqslant (r+1)\ \ln 2$$

Puis

$$\frac{\ln n}{\ln 2} - 1 \leqslant r \leqslant \frac{\ln n}{\ln 2}$$

Enfin

$$\frac{\ln n}{\ln 2} \leqslant r+1 \leqslant \frac{\ln n}{\ln 2} + 1$$

Comme on a $r+1$ tours de boucle, on en déduit que

$$\boxed{\text{la complexité de cet algorithme est en } O(\ln n).}$$

Exercice 4

1. Proposition de script :

```
def nombreZeros(t,i):
    if t[i] == 1:
        return 0
    else:
        S = 0
        k = i
        while (k < len(t) and t[k] == 0):
            S = S+1
            k = k+1
    return  S
```

Commentaires. Si on ne met pas la commande `k < len(t)`, si on demande dans l'exemple de l'énoncé `nombreZeros(t1,16)`, on obtient un message d'erreur ' out of range '. En effet l'algorithme devra après le premier tour de boucle tester t1[17] qui n'existe pas.

2. Proposition de script :

```
def nombreZerosMax(t):
    n = len(t)
    Max = 0
    for i in range(n):
    a = nombreZeros(t,i)
        if a > Max:
            Max = a
    return  Max
```

3. Considérons d'abord la fonction `nombreZeros`.
Si t[i]=1 l'algorithme n'effectue qu'un seul test, donc la complexité dans le meilleur des cas est $O(1)$.
Supposons que i=0 et que la liste ne contient que des 0. Notons n la taille de la liste. L'algorithme effectue un test, 2 affectations et puis 6 n affectations, test ou opérations. Donc au total $3 + 6\,n$. Donc la complexité dans le pire des cas est $O(n)$.
Considérons maintenant la fonction `nombreZerosMax`.
Si la liste ne contient que des 1, on a d'abord 2 affectations puis dans chaque boucle le calcul de `nombreZeros(t,i)` et un test. Compte-tenu de la complexité $O(1)$ de la fonction `nombreZeros` dans ce cas, on aura une complexité en $O(n)$.
Si la liste contient que des 0, on a d'abord 2 affectations puis dans chaque boucle le calcul de `nombreZeros(t,i)` et un test et éventuellement une affectation. Comme il n'y a que des 0, à chaque tour de boucle la complexité de la fonction `nombreZeros` sera en $O(i)$. Au total on aura une complexité en $\sum_{i=0}^{n-1} O(i)$, ce qui donne une complexité en $O(n^2)$.

4. Proposition de script :

```
def nombreZerosMaxAmeliore(t,i):
    Max = 0
    i = 0
    while i < len(t):
        a = nombreZeros(t,i):
        if a > Max:
            Max = a
        if a == 0:
            i = i+1
        else:
            i = i+a
    return Max
```

Commentaires. Avec cet algorithme on ne parcourt la liste qu'une seule fois. En effet :

- Quand `a=0` ce qui correspond à `t[i]==1`, on passe de `i` à `i+1`.
- Quand pour un indice i, on passe de `t[i-1]==1` à `t[i]==0`, alors `a` est égal au nombre de zéros consécutifs à partir de cet indice i et on passe alors de `i` à `i+a`, ce qui revient à passer " au dessus " des 0, et à reprendre la boucle au premier 1 qui suit les 0 successifs.

Cet algorithme sera donc linéaire.
On peut améliorer un peu l'algorithme en remplaçant `i=i+a` par `i=i+a+1`, ce qui évite de considérer le premier 1 après les 0 successifs.