UNIVERSITÉ BORDEAUX 1 Sciences Technologies DISVE Licence	ANNÉE : 2011/2012 SEMESTRE 2 **PARCOURS :** Licence LIMI201 & LIMI211 **UE J1MI2013 :** Algorithmes et Programmes **Épreuve :** Devoir surveillé **Date :** Vendredi 13 avril 2012 **Heure :** 11 heures **Durée :** 1 heure 30 Documents : non autorisés

SUJET + CORRIGE

Avertissement

- La plupart des questions sont indépendantes.
- Les fonctions doivent être écrites en `Python` en respectant les indentations.
- L'espace laissé pour les réponses est suffisant (sauf si vous utilisez ces feuilles comme brouillon, ce qui est fortement déconseillé).
- La note maximale est de 42/40 → 20/20.

Question	Points	Score
Suites et tableaux	12	
Tableaux et prédicats	10	
Fonction mystère	9	
Complexité	11	
Total:	42	

Exercice 1: Suites et tableaux (12 points)

On considère la suite définie par :

$$\begin{cases} u_0 = 2 \\ u_n = 3 * u_{n-1} - 1 \end{cases}$$

(a) (2 points) Écrire une fonction itérative qui prend comme paramètre un entier naturel `n` et calcule le terme u_n de la suite en utilisant une boucle `while`.

Solution:

```
def u-while(n) :
    u = 2
    i = 0
    while i < n :
        u = 3*u -1
        i += 1
    return u
```

(b) (2 points) Écrire une fonction itérative qui prend comme paramètre un entier naturel `n` et calcule le terme u_n de la suite en utilisant une boucle `for`.

Solution:

```
def u-for(n) :
    u = 2
    for i in range(1,n+1) :
        u = 3*u -1
    return u
```

(c) (2 points) Écrire une fonction récursive qui prend comme paramètre un entier naturel `n` et calcule le terme u_n de la suite.

Solution:

```
def u-rec(n) :
    if n == 0 :
        return 2
    else :
        return 3* u-rec(n-1) -1
```

(d) (3 points) Écrire une fonction qui renvoie un tableau contenant les n premiers termes de la suite (cette fonction ne doit utiliser aucune des fonctions précédentes, et peut utiliser les fonctions de la bibliothèque `bibTableau.py`).

Solution:

```
def u-tableau(n) :
    t = creerTableau(n)
    if n > 0 :
        t[0] = 2
        for i in range(1,n) :
            t[i] = 3*t[i-1] -1
    return t
```

(e) (3 points) Écrire une fonction qui, étant donné un entier `m`, calcule l'indice du premier terme de la suite supérieur ou égal à `m` (exemple : si m = 30, la fonction retournera 3 car tous les termes d'indice inférieur à 3 sont plus petits que 30).

Solution:

```
def u-sup(m) :
    u = 2
    i = 0
    while u < m :
        u = 3*u -1
        i += 1
    return i
```

Exercice 2: Tableaux et prédicats (10 points)

Trois exemples de tableaux pour illustrer les définitions :

```
>>> a = [1,4,3]
>>> b = [1,0,0]
>>> c = [1,0,2]
```

Soit `t` un tableau d'entiers de taille n.

(a) (3½ points) Écrire une fonction `sansDoublons(t)` qui retourne `True` si le tableau d'entiers `t` est sans doublons (c'est à dire sans apparition multiple d'un élément), `False` sinon.
Exemples :

```
>>> sansDoublons(a) : True
>>> sansDoublons(b) : False
>>> sansDoublons(c) : True
```

Solution:

```
def sansDoublons(t) :
    for i in range(len(t)-1) :
        for j in range(i+1,len(t)) :
            if (t[i]==t[j]):
                return False
    return True
```

(b) (1½ points) Donner et justifier la complexité de cette fonction.

Solution: Deux boucles imbriquées avec un test de sortie dans la boucle interne :
– Pire des cas (le test est toujours faux) : $\sum_{i=0}^{i=n-2}((n-1)-(i+1)+1) = \sum_{i=1}^{i=n-1} i = \frac{n(n-1)}{2}$. Soit $\mathcal{O}(n^2)$, où n est la longueur du tableau.
– Meilleur des cas (le premier test est vrai) : Soit $\Omega(1)$.

(c) (3 points) Écrire une fonction `interne(t)` qui retourne `True` si $\forall i \in [0, n[, t[i] \in [0, n[$ et retourne `False` sinon.
Exemples :

```
>>> interne(a) : False
>>> interne(b) : True
>>> interne(c) : True
```

Solution:

```
def interne(t) :
    for i in range(len(t)) :
        if (t[i]<0) | (t[i]>=len(t)) :
            return False
    return True
```

(d) (2 points) `t` est une permutation si $\forall i \in [0, n[, t[i] \in [0, n[$ et si `t` ne contient pas de doublons.
Exemples :

```
>>> permutation(a) : False
>>> permutation(b) : False
>>> permutation(c) : True
```

Ecrire une fonction `permutation(t)` qui retourne `True` si `t` est une permutation, `False` sinon.

Solution:

```
def permutation(t) :
    return interne(t) & sansDoublons(t)
```

Exercice 3: Fonction mystère (9 points)

Soit la fonction `mystere(t, k)` où `t` est un tableau d'entiers non vide et `k` vérifiant $0 \leq$ `k` $<$ `len(t)`.

```
def mystere(t, k) :
    if k == len(t) - 1 :
        return True
    if t[k] > t[k+1] :
        return False
    return mystere(t, k+1)
```

(a) Soit `t` = [6, 9, 4, 8, 12]

i. (1 point) Que retourne `mystere(t, 2)` (Donner la liste des appels récursifs) ?

> **Solution:** `mystere(t,2), mystere(t,3), mystere(t,4)` → `True`

ii. (1 point) Que retourne `mystere(t, 0)` (Donner la liste des appels récursifs) ?

> **Solution:** `mystere(t,0), mystere(t,1)` → `False`

(b) (2 points) Que fait la fonction `mystere` dans le cas général ?

> **Solution:** La fonction `mystere(t, k)` retourne `True` si la suite d'entiers contenue dans le tableau `t` à partir de l'indice `k` est croissante et retourne `False` sinon.

(c) (1 point) Quel est le nombre maximum d'appels récursifs (en fonction de `n` et `k`) de la fonction `mystere(t,k)` si le tableau `t` est de longueur `n` ?

> **Solution:** Le pire des cas correspond à l'appel `mystere(t,k)` pour un tableau croissant à partir de l'indice `k`. Il y a dans ce cas `n-k` appels récursifs en comptant l'appel initial.

(d) (1 point) En utilisant la fonction `mystere`, écrire une fonction `estCroissant(t)` qui retourne `True` si la suite d'entiers contenue dans le tableau `t` est croissante et retourne `False` sinon.

> **Solution:**

```
def estCroissant(t) :
    return mystere(t,0)
```

(e) (3 points) En vous inspirant de la fonction `mystere`, écrire une fonction récursive `estDans(t, x, k)` qui retourne `True` si `x` apparait dans le tableau `t` à partir de l'indice `k`, `False` sinon.

> **Solution:**

```
def estDans(t, x, k) :
    if k > len(t) - 1 :
        return False
    if t[k] == x :
        return True
    return estDans(t, x, k+1)
```

Exercice 4: Complexité (11 points)

(a) Soit la fonction `syr(n)` où `n` est un entier :

```
def syr(n) :
    while n > 1 :
        n = n // 2
    return n
```

i. (1 point) Donner les suites des valeurs successives prises par la variable `n` lors des deux appels `syr(32)` et `syr(20)`.

Solution:
- `syr(32)` : $32, 16, 8, 4, 2, 1$ retourne 1.
- `syr(20)` : $20, 10, 5, 2, 1$ retourne 1.

ii. (2 points) Quelles sont les complexités (meilleur et pire des cas) de la fonction `syr` ?

Solution:
- Meilleur des cas : $\Omega(log_2(n))$.
- Pire des cas : $\mathcal{O}(log_2(n))$.
- Soit : $\Theta(log_2(n))$.

(b) Soit la fonction `syrac(n)` où `n` est un entier :

```
def syrac(n) :
    while n > 1 :
        if n%2 == 0 :
            n = n // 2
        else :
            n = 1
    return n
```

i. (1 point) Donner les suites des valeurs successives prises par la variable `n` lors des deux appels `syrac(32)` et `syrac(20)`.

Solution:
- `syrac(32)` : $32, 16, 8, 4, 2, 1$ retourne 1.
- `syrac(20)` : $20, 10, 5, 1$ retourne 1.

ii. (2 points) Quelles sont les complexités (meilleur et pire des cas) de la fonction `syrac` ?

Solution:
- Meilleur des cas : un nombre impair, $\Omega(1)$.
- Pire des cas : une puissance de 2, $\mathcal{O}(log_2(n))$.

(c) Soit la fonction `syracuse(n)` où `n` est un entier :

```
def syracuse(n) :
    while n > 1 :
        if n%2 == 0 :
            n = n // 2
        else :
            n = 3*n + 1
    return n
```

i. (1 point) Donner les suites des valeurs successives prises par la variable `n` lors des deux appels `syracuse(32)` et `syracuse(20)`.

> **Solution:**
> – `syracuse(32)` : $32, 16, 8, 4, 2, 1$ retourne 1.
> – `syracuse(20)` : $20, 10, 5, 16, 8, 4, 2, 1$ retourne 1.

Depuis 1937, les mathématiciens et les informaticiens étudient les suites des valeurs successives prises par la variable `n` lors de l'appel à `syracuse(n)`. Ils cherchent à savoir si ces suites sont toutes finies, ou bien s'il en existe des cycliques, ou bien s'il en existe qui contiennent une sous-suite strictement croissante. Ce problème est un problème *ouvert*, ce qui signifie que personne ne connait la réponse.

ii. (2 points) Quelles sont les complexités (meilleur et pire des cas) de la fonction `syracuse` ?

> **Solution:**
> – Meilleur des cas : une puissance de 2, $\mathcal{O}(log_2(n))$.
> – Pire des cas : inconnue.

(d) (2 points) Soit la fonction `syrFor(n)` où `n` est un entier :

```
def syrFor(n) :
    s = 0;
    for i in range(n+1):
        s = s + syr(i)
    return s
```

Quelles sont les complexités (meilleur et pire des cas) de la fonction `syrFor` ?

> **Solution:** Le meilleur des cas est aussi le pire des cas.
> `syrFor` : $\sum_{i=1}^{i=n} log_2(i) = log_2(\prod_{i=1}^{i=n} i) = log_2(n!)$ soit $\Theta(log_2(n!))$.
>
> *La suite n'était pas demandée.*
>
> $$\begin{aligned} \texttt{syrFor} &= \sum_{i=1}^{i=n} log_2(i) \\ &\leq \sum_{i=1}^{i=n} log_2(n) \\ &\leq n\ log_2(n) \end{aligned}$$
>
> Soit $\mathcal{O}(nlog_2(n))$
>
> $$\begin{aligned} \texttt{syrFor} &= \sum_{i=1}^{i=n} log_2(i) \\ &= \frac{1}{2}(\sum_{i=1}^{i=n} log_2(i) + \sum_{i=1}^{i=n} log_2(n+1-i)) \\ &= \frac{1}{2}(\sum_{i=1}^{i=n}(log_2(i) + log_2(n+1-i))) \\ &= \frac{1}{2}(\sum_{i=1}^{i=n} log_2(i(n+1-i))) \\ &\geq \frac{1}{2}(\sum_{i=1}^{i=n} log_2(n)) \\ &\geq \frac{n}{2} log_2(n) \end{aligned}$$
>
> Soit $\Omega(nlog_2(n))$
>
> Soit $\Theta(nlog_2(n))$.