Devoir d'informatique n° 2 (2 heures)

Ce devoir est constitué de plusieurs petits exercices. L'ordre des exercices ne correspond à aucun critère de difficulté ou de longueur : vous pouvez les traiter dans l'ordre que vous voulez. Veillez à soigner la copie tant pour l'écriture, la propreté que pour la rédaction, la rigueur et l'argumentation. De plus, on prêtera une attention particuliére au respect des alignements et des indentations des séquence d'instructions Python. La calculatrice est interdite.
Vous numéroterez vos copies et ferez apparaître clairement sur la première page le nombre de copies.
Consignes particulières : Lorsque l'on demande d'écrire une fonction classique d'opérations sur les listes ou tout autre objet, et qu'une telle fonction ou méthode existe, vous n'êtes évidemment pas autorisé à l'utiliser

Exercice 1

On se propose d'écrire une fonction **renverse(t)** pour renverser le contenu d'un tableau.
Par exemple, si on passe en paramètre à la fonction **renverse** le tableau **t = [1, 2, 3, 4]**, après l'appel **t** contiendra **[4, 3, 2, 1]**.
La fonction suivante n'atteint pas l'objectif fixé :

```
>>> def renverse(t):
>>>         taille = len(t)         #len est la seule fonction sur les listes acceptée
>>>         for i in range(taille):
>>>                 aux = t[i]
>>>                 t[i] = t[taille-i-1]
>>>                 t[taille-i-1] = aux
```

1. Vérifier que la fonction ne répond pas à l'objectif fixé, en observant le résultat de l'appel sur le tableau **t = [1, 2, 3, 4]**. On donnera le contenu de **t** après chaque tour de boucle.
2. Proposer une modification (légère) de la fonction **renverse** pour renverser effectivement le contenu du tableau.
3. Proposer une modification plus profonde de la fonction **renverse** utilisant la technique de slicing.
4. Ecrire une fonction **egal(t1, t2)** prenant en paramètre les listes de nombres **t1** et **t2** et qui retourne le booléen **True** si les deux tableaux sont identiques (constitués des mêmes éléments et dans le même ordre), et qui retourne le booléen **False** sinon.
   Attention : dans le script de la fonction, on n'aura pas le droit d'utiliser des comparaisons sur d'autres objets que des nombres (entiers ou flottants).
5. Déterminer le nombre d'opérations élémentaires (affectations, comparaisons de nombres, additions de nombres etc.) en fonction de n longueur maximale de **t1** et **t2** de l'appel **egal(t1, t2)** dans le pire des cas. Même question mais pour le meilleur des cas.
6. Un tableau est dit "palindrome" si on lit la même suite de nombres en la parcourant de gauche à droite ou de droite à gauche. Par exemple, [6, 2, 7, 4, 7, 2, 6] est un tableau "palindrome", [2, 0, 0, 2] également.
   Écrire une fonction **estPalindrome(t)** qui retourne **True** ou **False** suivant que **t** est un tableau "palindrome" ou pas.

Exercice 2

Lors d'une course de vélo composée de n étapes, où la ligne de départ d'une étape est la ligne d'arrivée de l'étape précédente, on s'intéresse aux altitudes des différentes lignes d'arrivées et de la ligne du départ de la course. Ces altitudes sont mémorisées dans un tableau **A** : **A[0]** représente l'altitude du point de départ et **A[1]** l'altitude du point d'arrivée de la première étape, **A[2]** représente l'altitude du point d'arrivée de la seconde étape et ainsi de suite jusqu'à **A[n]**, altitude du point d'arrivée de la dernière étape (s'il y a n étapes au total...).
Chaque étape est soit "montante" soit "descendante" et on appelle dénivelé l'écart d'altitude entre deux étapes (un dénivelé est toujours positif que l'étape soit montante ou descendante).

1. Ecrire une fonction **nombreEtapesMontantes** qui prend en paramètre un tableau d'altitudes et calcule le nombre d'étapes " montantes"
2. Ecrire une fonction **sommeDeniveles** qui calcule la somme de tous les dénivelés.

Exemple : pour le tableau A = [500, 1100, 1200, 800, 200, 300] le résultat de **nombreEtapesMontantes** est **3** (il y a trois étapes montantes : les deux premières et la dernière) et la fonction **sommeDeniveles** retourne **1800** (600 + 100 + 400 + 600 + 100)

Exercice 3

La fonction **fibonnacci** ci-dessous calcule f_n, $n-$ième terme de la suite de Fibonacci :

```
>>> def fibonacci(n):
>>>         fib2 = 1
>>>         fib1 = 1
>>>         fib = 1
>>>         k = 1
>>>         while k < n :
>>>                 fib = fib2 + fib1
>>>                 fib2 = fib1
>>>                 fib1 = fib
>>>                 k += 1
>>>         return(fib)
```

En utilisant **fibonacci** on définit une nouvelle fonction **sommeFibo(n)** qui retourne la somme des $n+1$ premiers nombres de la suite de Fibonacci :

```
>>> def sommeFibo(n):
>>>         somme = 0
>>>         for i in range(n+1):
>>>                 somme += fibonacci(i)
>>>         return(somme)
```

1. Quels sont les résultats des appels **sommeFibo(0)**, **sommeFibo(1)** , **sommeFibo(4)** ?
2. Compter le nombre d'additions effectués lors de l'appel de **fibonacci(n)**. Utiliser ce résultat pour déterminer le nombre d'additions effectuées lors de l'exécution de **sommeFibo(n)**
3. Proposer une fonction **sommeFiboBis(n)** qui calcule le même résultat que la fonction **sommeFibo(n)** mais diminue le nombre d'opérations effectuées en évitant l'appel à **fibonacci** pour chaque tour de boucle.

Exercice 4

Pour cet exercice, on suppose que l'on dispose d'une fonction **creeTableau(p, k)**où p est un entier naturel et k est un nombre entier ou un nombre flottant, qui crée une liste de longueur p contenant p fois la valeur k. Par exemple l'appel **creeTableau(5, 2)** donne la liste [2, 2, 2, 2, 2] alors que l'appel à **creeTableau(0, 2)** donne []

1. Ecrire une telle fonction **creeTableau(p, k)**
2. Écrire une fonction **compterChangements(t)** qui prend en paramètre un tableau **t** et qui retourne le nombre d'apparitions d'une valeur différente à la précédente (en comptant la première valeur). Par exemple, les appels à **compterChangements(t)** pour les tableaux **t** = [], **t** = [2] et **t** = [2, 1, 1, 1, 2, 3, 2] donnent les retours : 0, 1 et 5.
3. Déterminer la complexité de cette fonction ?
   Pour représenter une suite de k valeurs v identiques de manière compressée, on utilise une paire

(k, v). En Python, on représentera une paire (k, v) par un tableau à deux éléments [k, v]. Par exemple, la suite 2, 2, 2, 2 est représentée par la paire (4, 2) et en Python par le tableau [4, 2].

4. Ecrire une fonction Python **creerPaire(k, v)** qui retourne le tableau [k, v]
5. Soit la fonction **mystere(t)** prenant comme paramètre un tableau non vide **t** de nombres et dont le script est le suivant :

```
>>> def mystere(t):
>>>         s = creerTableau(compterChangements(t), 0)
>>>         v = t[0]
>>>         k = 1
>>>         j = 0
>>>         for i in range(1, len(t)) :
>>>                 if t[i] != v :
>>>                         s[j] = creerPaire(k, v)
>>>                         k = 1
>>>                         j = j + 1
>>>                         v = t[i]
>>>                 else :
>>>                         k = k + 1
>>>         s[j] = creerPaire(k, v)
>>>         return(s)
```

Que retournent les appels suivants ? Dans la réponse, on attend également l'évolution des différentes variables locales utilisées par la fonction (et on pourra noter ces évolutions dans un tableau)

(a) **mystere([1, 1, 1])**

(b) **mystere([1, 1, 1, 2, 2, 4, 1])**

6. Quelle est la complexité de la fonction **mystere** ?
7. Expliquer en une ou deux phrases ce que retourne la fonction **mystere** en fonction de **t**
8. Ecrire une fonction **reveleMystere** qui réalise l'opération inverse de celle de **mystere** à savoir que si on lui donne comme paramètre, le résultat **res** de l'appel **mystere(t)**, la réponse de **reveleMystere(res)** doit être le tableau **t** initial.

CORRECTION

Exercice 1

On se propose d'écrire une fonction **renverse(t)** pour renverser le contenu d'un tableau. La fonction suivante n'atteint pas l'objectif fixé :

```
>>> def renverse(t):
>>>         taille = len(t)          #len est la seule fonction sur les listes acceptée
>>>         for i in range(taille):
>>>                 aux = t[i]
>>>                 t[i] = t[taille-i-1]
>>>                 t[taille-i-1] = aux
```

1. Résultat de l'appel sur le tableau **t = [1, 2, 3, 4]**

i	instructions	t	aux	t[i]	t[taille - 1 - i]
	initialisation	[1, 2, 3, 4]			
0	aux = t[i]	[1, 2, 3, 4]	1	1	4
0	t[i] = t[taille - i - 1]	[4, 2, 3, 4]	1	4	4
0	t[taille - i - 1] = aux	[4, 2, 3, 1]	1	4	1
1	aux = t[i]	[4, 2, 3, 1]	2	2	3
1	t[i] = t[taille - i - 1]	[4, 3, 3, 1]	2	3	3
1	t[taille - i - 1] = aux	[4, 3, 2, 1]	2	3	2
2	aux = t[i]	[4, 3, 2, 1]	2	2	3
2	t[i] = t[taille - i - 1]	[4, 3, 3, 1]	2	3	3
2	t[taille - i - 1] = aux	[4, 2, 3, 1]	2	3	2
3	aux = t[i]	[4, 2, 3, 1]	1	1	4
3	t[i] = t[taille - i - 1]	[4, 2, 3, 4]	1	4	4
3	t[taille - i - 1] = aux	[1, 2, 3, 4]	1	4	1

L'appel de la fonction **renverse** au tableau **t = [1, 2, 3, 4]** redonne ce même tableau **t = [1, 2, 3, 4]**.

2. La fonction **renverse** inverse deux fois de suite le tableau initial. Il suffit d'arréter la boucle à (1/2)len(t) pour obtenir l'objectif fixé.

```
>>> def renversebis(t):
>>>     taille = len(t)
>>>     for i in range(taille//2):  # penser à la division entière
>>>         aux = t[i]
>>>         t[i] = t[taille-i-1]
>>>         t[taille-i-1] = aux
```

3. Technique de slicing

```
>>> def renverseslice(t):
>>>     aux = t[len(t)-1::-1]
>>>     for k in range(len(t)):
>>>         t[k] = aux[k]
```

4. Fonction **egal(t1, t2)**

```
>>> def egal(t1, t2):
>>>     if len(t1) != len(t2):
>>>         return(False)
>>>     for k in range(len(t1)):
>>>         if t1[k] != t2[k]:
>>>             return(False)
>>>     return(True)
```

5. Si n est la longueur maximal de $t1$ et $t2$, la complexité dans le pire des cas est $\bigcirc(n)$: $n+1$ comparaisons (longueur et termes), 2 calculs de longueur, 1 retour. e pire des cas est lorsque les deux tableaux sont égaux ou lorsqu'ils ont la même taille mais qu'ils ne diffèrent que par le dernier terme.
   Dans le meilleur des cas, la complexité est en $\bigcirc(1)$: 2 calculs de longueur, 1 test et 1 retour.
6. On peut soit utiliser la fonction **egal** soit réécrire une fonction propre à la détermination des tableaux palindromes

```
>>> def estPalindrome(t):
>>>     taille = len(t)
>>>     for i in range(taille//2):
>>>         if t[i] != t[taille - 1 - i]:
>>>             return(False)
>>>     return(True)
```

Exercice 2

1. Fonction **nombreEtapesMontantes** calcule le nombre d'étapes " montantes"

```
>>> def nombreEtapesMontantes(A):
>>>     res = 0
>>>     for i in range(1,len(A)):
>>>         if A[i] >= A[i-1]:
>>>             res += 1
>>>     return(res)
```

2. Fonction **sommeDeniveles** qui calcule la somme de tous les dénivelés.

```
>>> def sommeDeniveles(A):
>>>     res = 0
>>>     for i in range(1,len(A)):
>>>         res += abs(A[i] - A[i-1])
>>>     return(res)
```

Exercice 3

La fonction **fibonnacci** ci-dessous calcule f_n, $n-$ième terme de la suite de Fibonacci :

```
>>> def fibonacci(n):
>>>     fib2 = 1
```

```
>>>     fib1 = 1
>>>     fib = 1
>>>     k = 1
>>>     while k < n :
>>>         fib = fib2 + fib1
>>>         fib2 = fib1
>>>         fib1 = fib
>>>         k += 1
>>>     return(fib)
```

En utilisant **fibonacci** on définit une nouvelle fonction **sommeFibo(n)** qui retourne la somme des $n+1$ premiers nombres de la suite de Fibonacci :

```
>>> def sommeFibo(n):
>>>     somme = 0
>>>     for i in range(n+1) :
>>>         somme += fibonacci(i)
>>>     return(somme)
```

1. **sommeFibo(0)** retourne 1, **sommeFibo(1)** retourne 2 ($= 1 + 1$), **sommeFibo(4)** retourne 12 $= (1 + 1 + 2 + 3 + 5)$
2. Pour l'appel de **fibonacci(n)**, on utilise $2n$ additions, $4n + 4$ affectations, n tests et 1 retour. Lors de l'exécution de **sommeFibo(n)**, à l'étape i on effectue $1 + 2i$ additions ($2i$ dans fibonacci(i) et 1 dans sommeFibo). Donc pour sommeFibo(n), on a : $1 + 3 + 5 + \cdots + (2n + 1)$ additions soit n^2 additions
3. Fonction **sommeFiboBis(n)**

```
>>> def sommeFibobis(n):
>>>     somme = 0
>>>     fib2 = 1
>>>     fib1 = 1
>>>     fib = 1
>>>     k = 0
>>>     while k<n :
>>>         somme += fib2
>>>         fib = fib2 + fib1
>>>         fib2 = fib1
>>>         fib1 = fib
>>>         k += 1
>>>     somme += fib2
>>>     return(somme)
```

Le nombre d'opérations est : $3n+1$ additions, $n+1$ tests, $5n+6$ affectations : Complexité en $\bigcirc(n)$
e

Exercice 4

1. Fonction **creeTableau(p, k)**

```
>>> def creerTableau(p,k):
>>>     return([k]*p)
```

2. Fonction **compterChangements(t)**

```
>>> def compterChangements(t):
>>>     taille = len(t)
>>>     if taille == 0:
>>>         return(0)
>>>     res = 1
>>>     for i in range(1, taille):
>>>         if t[i] != t[i-1]:
>>>             res += 1
>>>     return(res)
```

3. Complexité de cette fonction : $\bigcirc(n)$ où $n = \text{len(t)}$

4. Ecrire une fonction Python **creerPaire(k, v)** qui retourne le tableau [k, v]

```
>>> def creerPaire(k,v):
>>>     return([k,v])
```

5. Soit la fonction **mystere(t)** prenant comme paramètre un tableau non vide **t** de nombres et dont le script est le suivant :

```
>>> def mystere(t):
>>>         s = creerTableau(compterChangements(t), 0)
>>>         v = t[0]
>>>         k = 1
>>>         j = 0
>>>         for i in range(1, len(t)) :
>>>                 if t[i] != v :
>>>                         s[j] = creerPaire(k, v)
>>>                         k = 1
>>>                         j = j + 1
>>>                         v = t[i]
>>>                 else :
>>>                         k = k + 1
>>>         s[j] = creerPaire(k, v)
>>>         return(s)
```

   (a) **mystere([1, 1, 1])** retourne [[3, 1]]

   (b) **mystere([1, 1, 1, 2, 2, 4, 1])** retourne [[3, 1], [2, 2], [1, 4], [1, 1]]

6. La fonction retourne le tableau s des paires qui représente une forme compressée du tableau t.

7. Fonction **reveleMystere**

```
>>> def reveleMystere(t):
>>>     res = []
>>>     for x in t:
>>>         res += [x[1]] *x[0]
>>>     return(res)
```