Devoir d'informatique n° 3 (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 particulière au respect des alignements et des indentations des séquences d'instructions Python. La calculatrice est interdite.
Vous numéroterez vos copies et ferez apparaître clairement sur la première page le nombre de copies.

Exercice 1

1. Ecrire une fonction **factorielle** prenant comme argument un entier naturel **n** et qui renvoie **n!** (*On n'acceptera pas bien sûr de réponse utilisant la propre fonction factorielle du module math ou numpy de Python.*).

2. Ecrire une fonction **seuil** prenant comme argument un entier **M** et qui renvoie le plus petit entier **n** tel que **n!** $>$ **M**.

3. Ecrire une fonction booléenne nommée **est_divisible** prenant comme argument un entier naturel **n** et qui renvoie **True** si **n!** est divisible par **n+1** et **False** sinon.

4. On considère la fonction suivante nommée **mystere**

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

   (a) Quelle valeur renvoie **mystere(4)** ?

   (b) Expliquer ce que fait cette fonction **mystere** en faisant intervenir un invariant de boucle

   (c) Déterminer le nombre de multiplications qu'effectue **mystere(n)** (*On supposera que l'appel à* ***factorielle(k)*** *nécessite k multiplications*).

   (d) Proposer une amélioration du script de la fonction **mystere** afin d'obtenir une complexité linéaire i.e en $\bigcirc$**(n)**

Exercice 2

La fonction **mystere** suivante prend comme arguments deux tableaux (à une dimension) **t** et **s**.

```
def mystere(t, s):
  lt , ls = len(t) , len(s)
  if lt != ls:
    return(False)
  i = 0
  while i < lt and  t[i] == s[ls - i - 1] :
    i = i + 1
  return (i == lt)
```

1. Que retournent les appels suivants ?

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

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

   (c) **mystere([1,2,5,4], [4,3,2,1])**

   (d) En général, quel est le résultat d'un appel de la fonction **mystere** ? Justifier brievement.

2. Evaluer la complexité en temps (dans le meilleur des cas et dans le pire des cas)de la fonction **mystere**

Exercice 3

1. Soit la fonction **syr** suivante qui prend comme argument un entier naturel **n** :

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

   (a) Donner les suites des valeurs successives prises par la variable **k** lors des deux appels **syr(32)** et **syr(20)**

   (b) Quelles sont les complexités en temps dans les meilleurs et pires des cas de la fonction **syr**

2. Soit la fonction **syrac** suivante qui prend comme argument un entier naturel **n** :

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

   (a) Donner les suites des valeurs successives prises par la variable **k** lors des deux appels **syrac(32)** et **syrac(20)**

   (b) Quelles sont les complexités en temps dans les meilleurs et pires des cas de la fonction **syrac**

3. Soit la fonction **syracuse** suivante qui prend comme argument un entier naturel **n** :

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

   On rappelle que le probleme de savoir si en partant d'un entier naturel quelconque, on arrive nécessairement à 1 est un *probleme ouvert* i.e. on ne connait pas encore de réponse à cette question.

   (a) Donner les suites des valeurs successives prises par la variable **k** lors des deux appels **syracuse(32)** et **syracuse(20)**

   (b) Quelles sont les complexités en temps dans les meilleurs et pires des cas de la fonction **syracuse**

4. Soit la fonction **syrFor** suivante qui prend comme argument un entier naturel **n** :

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

   Quelles sont les complexités en temps dans les meilleurs et pires des cas de la fonction **syrFor** ?

Exercice 4

1. Ecrire une fonction **sommeChiffres** prenant comme argument un entier naturel **n** et qui renvoie la somme des chiffres du nombre **n** (en base 10)
   On veut écrire une fonction qui teste si un nombre est multiple de 9 en utilisant la propriété suivante
   Un entier est multiple de 9 si et seulement si la somme de ses chiffres est multiple de 9.
   Le principe est de répéter le calcul de la somme des chiffres jusqu'à obtenir un nombre d'un seul chiffre.
   Exemple : si on teste avec $n = 9565938$. On calcule la somme de chiffres : on trouve 45. On réitère le processus avec 45 : on trouve 9. On s'arrête et on conclut que 9565938 est bien un multiple de 9.

2. Ecrire une fonction **estMultiplede9** prenant comme argument un entier naturel **n** et qui renvoie **True** si c'est un multiple de 9 et **False** sinon, en utilisant la propriété indiquée. On n'aura pas le droit d'utiliser les opérateurs % et / ou // dans cette question

3. Ecrire une fonction **indiceDernierMultiplede9** prenant comme argument un tableau unidimensionnel **t** d'entiers et qui retourne le plus grand indice de **t** où se trouve un multiple de 9, et retourne **None** si aucun multiple de p n'est présent dans le tableau **t** . On demande un algorithme qui minimise le temps d'exécution dans le cas le plus favorable.

4. Ecrire une fonction **supprimerPremierMultiplede9** prenant comme argument un tableau unidimensionnel **t** et qui renvoie le tableau obtenu à partir de **t** en supprimant le premier élément qui est un multiple de 9.

Exercice 5

On veut représenter un tableau **t** de nombres par un autre tableau **c** appelé **codage de t**. Les suites consécutives de valeurs identiques de **t** sont représentées dans **c** par deux nombres **r,v** où **r** est le nombre de répétitions de la valeur **v** dans une telle suite. Une valeur **v** de **t** qui ne se répète pas est donc représentée par **1, v**. Une valeur **v** qui se répète deux fois consécutivement est représentée par **2, v**, et ainsi de suite.
Par exemple, le codage de **t= [0, 0, 0, 0, 5, -2, -2, -2, 0, 0, 0, 0, 0]** est **c= [4, 0, 1, 5, 3, -2, 5, 0]**.
Un entier à un indice pair dans **c** représente donc un nombre de répétitions consécutives d'une valeur de **t**.

1. Ecrire une fonction **decoder** qui prend comme argument un tableau **c** de longueur paire et qui renvoie le tableau **t** dont **c** est le codage.

2. Quelle est la complexité de votre fonction **decoder** ? Justifier la réponse
   On veut écrire la fonction de codage qui prend en entrée un tableau **t** et renvoie son codage **c**.

   (a) Ecrire une fonction **longueurBloc** prenant comme argument un tableau **t** et un entier **pos** et qui renvoie la longueur du plus long bloc de positions consécutives du tableau **t** commençant à la position **pos**, et composé de cases consécutives qui contiennent toutes la valeur **t[pos]**. Par exemple avec **t = [0, 0, 0, 0, 5, -2, -2, -2, 0, 0, 0, 0, 0]**, les appels **longueurBloc(t, 0)**, **longueurBloc(t, 1)**, **longueurBloc(t, 4)**, **longueurBloc(t, 8)** doivent retourner respectivement 4, 3, 1 et 5.

   (b) En utilisant la fonction **longueurBloc**, écrire une fonction **codage** prenant comme argument un tableau **t** et qui renvoie le codage du tableau **t**.

CORRECTION

Exercice 1

1.
```
def factorielle(n):
  p = 1
  for k in range(1, n+1):
    p = p * k
  return(p)
```

2.
```
def seuil(M):
  p, n = 1, 1
  while p <= M :
    n = n + 1
    p = p * n
  return(n)
```

3.
```
def est_divisible(n):
  p = factorielle(n)
  if p %(n+1) == 0:
    return(True)
  else :
    return(False)
```

4. On considère la fonction suivante
```
def mystere(n):
  s = 0
  for k in range(1, n+1):
    s = s + factorielle(k)
  return(s)
```

(a) **mystere(4) retourne 33**

(b) On reprend les noms des variables locales de la fonction **mystere**. On note $(u_n)_{n\in\mathbb{N}^*}$ la suite définie par : $\forall n \in \mathbb{N}^*, u_n = \sum_{k=1}^{n} k!$. Montrons que l'expression "$s - u_k$" est un invariant de boucle.
A la première itération dans la boucle. s vaut 0 en entrée et on lui ajoute 1! soit s vaut 1 à la fin de la première itéeration ce qui est exactement la valeur de u_1.
Hérédité. A la $k+1$-ième itération, k prend la valeur $k+1$, s prend la valeur $s+(k+1)!$ et $u_{k+1} = u_k + (k+1)!$. Donc $s - u_k$ a la même valeur au début de l'étape $k+1$ qu'à la fin de cette étape.
Ainsi **$s - u_k$ est bien un invariant de boucle**. Or cette expression vaut 0 en entrée : elle vaut donc toujours 0 en sortie, soit lorsque k vaut n.
Ainsi **mystere(n) retourne $u_n = \sum_{k=1}^{n} k!$**

(c) Lors de l'appel **mystere(n)**, on effectue à l'étape k, 1 addition, 1 affectation et k multiplications (en calculant factorielle(k)).
Donc au total nous effectuons n additions et $\frac{n(n+1)}{2}$ **multiplications**

(d) On améliore la complexité en temps, en calculant en parallèle le produit des premiers entiers et la somme des premières factorielles.

```
def mystererapide(n):
  p, s = 1, 1
  for k in range(2, n+1):
    p = p * k
    s = s + p
  return(s)
```

Exercice 2

La fonction **mystere** suivante prend comme arguments deux tableaux (à une dimension) **t** et **s**.

```python
def mystere(t, s):
  lt , ls = len(t) , len(s)
  if lt != ls:
    return(False)
  i = 0
  while i < lt and  t[i] == s[ls - i - 1] :
    i = i + 1
  return (i == lt)
```

1. Que retournent les appels suivants ?

   (a) **mystere([ ], [ ]) retourne True**

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

   (c) **mystere([1,2,5,4], [4,3,2,1])retourne False**

   (d) La fonction prend en paramètres deux tableaux **t** et **s**, et vérifie si la suite des éléments dans **t**, lue du plus petit indice au plus grand indice, est la même que la suite dans des éléments dans **s** lue en sens inverse

2. Dans la meilleur des cas, les tableaux ne sont pas de la même longueur et au bout d'un test, on retourne le résultat : **complexité dans le meilleur des cas : $\bigcirc(1)$**
   Dans le pire des cas, les tableaux sont à l'envers l'un de l'autre ou quasiment (ils différent de leur miroir par le dernier élément à tester). Dans ce cas, on a effectué 2 len(t) tests de len(t) additions : **complexité dans le pire des cas : $\bigcirc(\text{len}(t))$**

Exercice 3

1. Soit la fonction **syr** suivante qui prend comme argument un entier naturel **n** :

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

   (a) Lors de l'appel **syr(32)**, la variable **k** prend successivement les valeurs :
   $32, 16, 8, 4, 2, 1$ **et le retour est** 1
   Lors de l'appel **syr(20)**, la variable **k** prend successivement les valeurs :
   $20, 10, 5, 2, 1$ **et le retour est** 1

   (b) Les complexités dans les pires et les meilleurs des cas pour l'appel de **syr(n)** sont identiques : il s'agit de $\bigcirc(\log_2(n))$

2. Soit la fonction **syrac** suivante qui prend comme argument un entier naturel **n** :

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

(a) Lors de l'appel **syrac(32)**, la variable **k** prend successivement les valeurs :
$32, 16, 8, 4, 2, 1$ **et le retour est** 1
Lors de l'appel **syrac(20)**, la variable **k** prend successivement les valeurs :
$20, 10, 5, 1$ **et le retour est** 1

(b) Dans le meilleur des cas, à savoir lorsque n est impair, la complexité est $\bigcirc(1)$.
Dans le pire des cas, à savoir lorsque n est une puissance de 2, la complexité est $\bigcirc(\log_2(n))$.

3. Soit la fonction **syracuse** suivante qui prend comme argument un entier naturel **n** :

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

On rappelle que le probleme de savoir si en partant d'un entier naturel quelconque, on arrive nécessairement à 1 est un *probleme ouvert* i.e. on ne connait pas encore de réponse à cette question.

(a) Lors de l'appel **syracuse(32)**, la variable **k** prend successivement les valeurs :
$32, 16, 8, 4, 2, 1$ **et le retour est** 1
Lors de l'appel **syracuse(20)**, la variable **k** prend successivement les valeurs :
$20, 10, 5, 16, 8, 4, 2, 1$ **et le retour est** 1

(b) Dans le meilleur des cas, à savoir lorsque n est une puissance de 2, la complexité est $\bigcirc(\log_2(n))$
Dans le pire des cas, la complexité est inconnue, au moins tant que le problème de savoir si la suite finit par arriver à 1 si on part d'un entier n quelconque reste ouvert.

4. Soit la fonction **syrFor** suivante qui prend comme argument un entier naturel **n** :

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

Dans les pires et les meilleurs des cas, l'appel à **syrFor(n)** nécessite $\sum_{k=1}^{n} \log_2(k)$ additions, divisions et tests, soit une complexité de $\bigcirc(n\log_2(n))$

Exercice 4

1.
```
def sommeChiffres(n):
  k, s = n, 0
  while k > 0:
    s = s + k % 10
    k = k // 10
  return(s)
```

2.
```
def estMultiplede9(n):
  k = n
  while k > 9:
    k = sommeChiffres9(k)
  if k ==9 or k == 0 :
    return(True)
  else : return(False)
```

3.
```
def indiceDernierMultiplede9(t):
  for k in range(len(t)-1, -1, -1) :
    if estMultiplede9(t[k]):
      return(k)
  return(None)
```

```
def supprimerPremierMultiplede9(t):
  tab, c = [] , 0
  for k in range(len(t)) :
    if not estMultiplede9(t[k]) or c > 0 :
      tab.append(t[k])
    if estMultiplede9(t[k]) :
      c = 1
  return(tab)
```

Exercice 5

On veut représenter un tableau **t** de nombres par un autre tableau **c** appelé **codage de t**. Les suites consécutives de valeurs identiques de **t** sont représentées dans **c** par deux nombres **r,v** où **r** est le nombre de répétitions de la valeur **v** dans une telle suite. Une valeur **v** de **t** qui ne se répète pas est donc représentée par **1, v**. Une valeur **v** qui se répète deux fois consécutivement est représentée par **2, v**, et ainsi de suite.
Par exemple, le codage de **t= [0, 0, 0, 0, 5, -2, -2, -2, 0, 0, 0, 0, 0]** est **c= [4, 0, 1, 5, 3, -2, 5, 0]**. Un entier à un indice pair dans **c** représente donc un nombre de répétitionsconsécutives d'une valeur de **t**.

1.
```
def decoder(c):
  tab = []
  for k in range(len(c)//2) :
    n , val = c[2*k] , c[2*k+1]
    for k in range(n) :
      tab.append(val)
  return(tab)
```

2. La complexitéde la fonction **decoder** est la somme des valeurs aux positions paires de **c**.

(a)
```
def longueurBloc(t,pos):
  c, k, n = 0, pos, len(t)
  while k < n and t[k] == t[
      pos]:
    c, k = c + 1, k + 1
  return(c)
```

(b)
```
def codage(t):
  tab, pos, n = [], 0, len(t)
  while pos < n :
    p = longueurBloc(t, pos)
    tab.append(p)
    tab.append(t[pos])
    pos = pos + p
  return(tab)
```