Interrogation 3 (5 novembre)

durée : 1 heure

Pour chacun des algorithmes, on justifiera avec soin :
- que l'algorithme termine (si une boucle « tant que » est utilisée).
- que l'algorithme renvoie bien le résultat demandé.

Une réponse non justifiée sera notée sur les trois quarts des points.

On peut utiliser les opérateurs `//` et `%` sur les entiers. On évitera d'utiliser des fonctionnalités trop avancées de python, surtout celles non vues en TD : on rédigera en pseudo-code avec des techniques élémentaires. En cas de doute, poser simplement la question.

Dans cette interrogation, « complexité » signifie « complexité en temps », on ne demandera pas de complexité en espace. Les complexités sont entendues dans le pire des cas.

Exercice 1. *(Limites, équivalents)*

1. *Déterminer la limite de* $(3\sqrt[n]{2} - 2\sqrt[n]{3})^n$.
2. *Déterminer un équivalent de* $\left(1 + \frac{\ln n}{n}\right)^n$.
3. *Montrer que* $\sum_{k=0}^{n} k! \sim n!$.

Exercice 2.

1. *Montrer qu'une suite équivalente à* 2^n *est croissante à partir d'un certain rang.*
2. *Une suite équivalente à* n^2 *est-elle croissante à partir d'un certain rang ?*

Exercice 3. *La suite dite de « Tribonacci » est la suite* (u_n) *définie par* $u_0 = 0$, $u_1 = 0$, $u_2 = 1$ *et pour tout* $n \geq 0$, $u_{n+3} = u_{n+2} + u_{n+1} + u_n$. *Dans cet exercice, on demande plusieurs implémentations d'une fonction recevant en entrée un entier naturel* n *et retournant la valeur de* u_n.

1. *Écrire une implémentation récursive simple. (Sans boucle.)*
2. *Écrire une implémentation itérative, sans récursivité, de complexité linéaire en* n.
3. *Bonus : écrire une implémentation récursive, sans boucle, de complexité linéaire en* n. *L'idée est la même qu'à la question précédente, on pourra utiliser une deuxième fonction intermédiaire pour la récursivité à l'intérieur de la fonction principale.*

À chaque fois, on prouvera la terminaison, la correction et dans les deux derniers cas, la complexité. Plus d'informations sur cette suite sur `https://oeis.org/A000073` *ou Wikipédia.*

Correction

Correction de l'exercice 1

1. On a $\sqrt[n]{2} = e^{\frac{\ln 2}{n}} = 1 + \frac{\ln 2}{n} + o\left(\frac{1}{n}\right)$. De même, $\sqrt[n]{3} = 1 + \frac{\ln 3}{n} + o\left(\frac{1}{n}\right)$. Donc :
$$3\sqrt[n]{2} - 2\sqrt[n]{3} = (3-2) + \frac{3\ln 2 - 2\ln 3}{n} + o\left(\frac{1}{n}\right) = 1 + \frac{\ln(8/9)}{n} + o\left(\frac{1}{n}\right)$$
On calcule la puissance n-ème avec l'exponentielle et le log de la même manière :
$$(3\sqrt[n]{2} - 2\sqrt[n]{3})^n = \exp\left(n \ln\left(1 + \frac{\ln(8/9)}{n} + o\left(\frac{1}{n}\right)\right)\right) = \exp(\ln(8/9) + o(1)) \to 8/9$$
2. On a $u_n = exp\left(n\ln\left(1 + \frac{\ln n}{n}\right)\right)$. Comme $\frac{\ln n}{n} \to 0$, on a $\ln\left(1 + \frac{\ln n}{n}\right) = \frac{\ln n}{n} + O\left(\frac{\ln^2 n}{n^2}\right)$, donc $n\ln\left(1 + \frac{\ln n}{n}\right) = \ln n + O\left(\frac{\ln^2 n}{n}\right)$. Finalement, $u_n = exp\left(\ln n + O\left(\frac{\ln^2 n}{n}\right)\right) \sim n$.
Remarque : on n'était pas loin d'avoir de façon précise le terme d'après, mais ce n'était pas demandé : $u_n = n - \frac{\ln^2(n)}{2} + o(\cdots)$. Pour vous entraîner, vous pouvez calculer le troisième terme, qui devrait être $\frac{\ln^3(n)(3\ln n + 8)}{24n}$.
3. Posons $u_n = \sum_{k=0}^n k!$. En majorant brutalement les $n-1$ premiers termes, on obtient :
$$1 \leq \frac{u_n}{n!} = \sum_{k=0}^n \frac{k!}{n!} = \frac{0!}{n!} + \frac{1!}{n!} + \cdots + \frac{1}{n(n-1)} + \frac{1}{n} + 1 \leq (n-1) \times \frac{1}{n(n-1)} + \frac{1}{n} + 1 \to 1$$

Correction de l'exercice 2

1. Comme 2^n n'est jamais nul, l'hypothèse équivaut à $u_n/2^n \to 1$. Voici deux rédactions.
Rédaction 1. La suite u_n est strictement positive partir d'un certain rang, et à partir de ce rang-là, on peut écrire $\frac{u_{n+1}}{u_n} = \frac{u_{n+1}}{2^{n+1}}\frac{2^{n+1}}{u_n} \to 2$. Ceci implique la croissance.
Rédaction 2. On revient à la définition de convergence. Soit $N \in \mathbb{N}$ tel que pour tout $n \geq N$, $\frac{u_n}{2^n} \in [2/3, 4/3]$. Supposons par l'absurde qu'il existe $n \geq N$ tel que $u_{n+1} < u_n$. Alors, on a $\frac{u_{n+1}}{2^n} < \frac{u_n}{2^n} \leq \frac{4}{3}$ et donc $\frac{u_{n+1}}{2^{n+1}} = \frac{1}{2}\frac{u_{n+1}}{2^n} < \frac{4}{6} = \frac{2}{3}$. C'est absurde car on doit avoir $\frac{u_{n+1}}{2^{n+1}} \in [2/3, 4/3]$.
2. Non. Considérons la suite (u_n) dont les premières valeurs sont : $0, -1, 4, 4-1, 16, 16-1, 36, 36-1, 64, 64-1$ etc.
Elle est définie par :
$$u_n = \begin{cases} n^2 & \text{si } n \text{ est pair,} \\ (n-1)^2 - 1 & \text{sinon} \end{cases}$$
Pour $n > 0$, notons $v_n := u_n/n^2$. Alors $v_n = 1$ si n est pair, et si n est impair, alors $v_n = \frac{(n-1)^2-1}{n^2} = 1 - \frac{1}{n}$. Ceci montre que v_n tend vers 1 et donc que $u_n \sim n^2$.

Correction de l'exercice 3

Remarque : en conditions de TP, les fonctions pourraient être testée et chronométrées avec :

```python
def test_tribo():
  assert tribo(3) == 1
  assert tribo(4) == 2
  assert tribo(5) == 4
  assert tribo(9) == 44
```

```python
  print("Test : tout semble ok")

import time
def perf_tribo(n):
  start = time.perf_counter()
  t = tribo(n)
  end = time.perf_counter()
  delta = end - start
  return delta
```

1. Implémentation récursive naïve :

```python
def tribo(n):
  if n <= 1:
    return 0
  if n == 2:
    return 1
  return tribo(n-1)+tribo(n-2)+tribo(n-3)
```

   Remarque : avec cette implémentation, tribo(30) met quelques seconde à s'exécuter sur ma machine (récente, 2020) et tribo(33) met presque seize secondes.
   Questions à creuser pour s'entraîner aux TP : estimer la complexité de cette implémentation. Mesurer le temps d'execution de tribo(15), tribo(20), tribo(25), essayer d'estimer le temps d'exécution de tribo(n) avec n entre 30 et 35, si si le temps estimé semble raisonnable, vérifier. Essayer de déterminer expérimentalement la complexité, en supposant que c'est une suite géométrique en n.

2. Implémentation itérative qui s'exécute en $O(n)$:
   Premier essai : on calcule la liste de toutes les valeurs jusqu'à n, que l'on stocke dans une liste et on retourne la dernière valeur de cette liste :

```python
def tribo(n):
  t =  [0,0,1]
  if n <= 2:
    return t[n]

  for i in range(n-2):
    t.append(t[i]+t[i+1]+t[i+2]) # plus clair (?) : t[-3]+t[-2]+t[-1]
  return t[-1] # ou t.pop(), ou t[len(t)-1]
```

   On peut comparer la vitesse d'exécution avec la version précédente : le calcul de tribo(1000), et même de tribo(10000), est quasiment instantané. Celui de tribo(100000) prend trois secondes. Enfin, le calcul de tribo(1000000) a semblé prendre bien plus de 30 secondes et a été abandonné. À noter que le stockage de toutes les valeurs de la suite est évidemment susceptible de ralentir l'exécution, ce qui motive une modification.

   Amélioration : on ne garde en mémoire que les trois dernières valeurs à chaque fois. Ceci utilise beaucoup moins de mémoire. (Complexité spatiale en $O(1)$ au lieu de $O(n)$.)

```python
def tribo(n):
  t =  [0,0,1]
  if n <= 2:
    return t[n]

  for i in range(n-2):
    somme  = t[0]+t[1]+t[2]
    t[0] = t[1]
    t[1] = t[2]
    t[2] = somme
  return t[2]
```

   Ou, plus court en utilisant les possibilités d'affectations multiples de Python :

```python
def tribo(n):
  t =  [0,0,1]
  if n <= 2:
    return t[n]

  for i in range(n-2):
    t[0], t[1], t[2] = t[1], t[2], t[0]+t[1]+t[2]

  return t[2]
```

Remarque : toujours sur la même machine, le calcul de tribo(1000000) s'est exécuté en moins d'une minute. Le processus a utilisé 20 Go de mémoire, suite à quoi sortir de Python a pris quasiment une minute...

3. Implémentation récursive de complexité linéaire : on va utiliser une fonction récursive auxiliaire qui prend en entrée un tableau contenant les trois dernières valeurs calculées.

```python
def tribo_aux(t,n):
  if n == 2:
    return t[2]
  t[0], t[1], t[2] = t[1], t[2], t[0]+t[1]+t[2]
  return tribo_aux(t,n-1)

def tribo(n):
  if n <= 1:
    return 0
  return tribo_aux([0,0,1],n)

```

Noter que par défaut, Python limite de toute façon le nombre d'appels récursifs.