**Exercices 19 à 22**

---

**Exercice 19 (complexité de l'algorithme d'exponentiation rapide).**

(1) Rappeler le principe de l'algorithme d'exponentiation rapide pour calculer la puissance $n$-ième ($n \in \mathbb{N}$) d'une matrice carrée $A$ donnée.
(2) Soit $C(n)$ le nombre de multiplications matricielles nécessaires pour calculer $A^n$ avec cet algorithme. Calculer $C(90)$.
(3) Montrer que $C(n) \leqslant 2 + C\big(\lfloor \frac{n}{2} \rfloor\big)$ pour tout $n \geqslant 1$, où $\lfloor x \rfloor$ désigne la partie entière (inférieure) de $x$.
(4) Montrer par récurrence sur $p \in \mathbb{N}$ que si $n \leqslant 2^p$, alors $C(n) \leqslant 2p$.
(5) En déduire que $C(n) = O(\log n)$.
(6) **(TD*)** Soit $C_1(n)$ le nombre d'occurences du chiffre 1 dans l'écriture de $n$ en base 2, et $C_0(n)$ le nombre d'occurences du chiffre 0. Trouver l'expression exacte de $C(n)$ en fonction de $C_0(n)$ et $C_1(n)$.

---

**Exercice 20 (résidu numérique d'un entier).** On appelle *résidu numérique* (ou *racine numérique*) d'un entier $n \geqslant 1$, noté $R(n)$, le chiffre obtenu en itérant (autant de fois que nécessaire) la transformation qui consiste à changer $n$ (écrit en base 10) en la somme de ses chiffres. Par exemple, le résidu numérique de 87 est 6, car $8 + 7 = 15$ et $1 + 5 = 6$.

(1) **(TP)** Écrire une fonction récursive `residu1(n)`, qui calcule le résidu numérique en utilisant la propriété
$$R(n) = R(p + q),$$
où $p$ et $q$ sont respectivement le quotient et le reste dans la division euclidienne de $n$ par 10 (on ne demande pas de justifier cette propriété). Tester cette fonction sur quelques exemples.
(2) **(TP)** Écrire une fonction python récursive `residu2(n)`, qui calcule le résidu numérique en utilisant la définition de $R$. On pourra utiliser une compréhension de liste de type `[int(x) for x in str(n)]`. Vérifier sur quelques exemples que cette fonction renvoie bien le même résultat que `residu1`.
(3) **(TD*)** Montrer que $n - R(n)$ est un multiple de 9, puis que
$$R(n) = n - 9 \left\lfloor \frac{n-1}{9} \right\rfloor,$$
où $\lfloor x \rfloor$ désigne la partie entière (inférieure) de $x$.

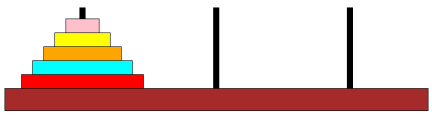
---

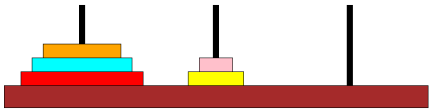
**Exercice 21 (suite récurrente d'ordre 2).** On considère dans cet exercice la suite définie par $u_0 = 0$, $u_1 = 1$ et la relation de récurrence $u_{n+2} = nu_{n+1} - n^2 u_n$ pour tout $n \in \mathbb{N}$.

(1) Écrire une fonction python **non récursive** `suite1(n)` qui renvoie le terme d'indice `n` de la suite $(u_n)$.
(2) Écrire une fonction **récursive** `suite2(n)` qui renvoie la liste des termes d'indices 0 à `n` de la suite $(u_n)$. La fonction `suite2` devra être simplement récursive (autrement dit, chaque appel de type `suite2(n)` produit au plus un appel récursif à la fonction `suite2`).
(3) Écrire une fonction **doublement récursive** `suite3(n)` qui renvoie le terme d'indice `n` de la suite $(u_n)$ (autrement dit, chaque appel de type `suite3(n)` produit, sauf exception, deux appels récursifs à la fonction `suite3`).
(4) Rappeler la complexité (en termes de nombre d'appels récursifs) des fonctions `suite2` et `suite3`.
(5) **(TP)** Implémenter la fonction `suite3` et calculer à l'aide de cette fonction les premières valeurs de la suite $(u_n)$. Jusqu'à quel indice peut-on raisonnablement faire le calcul ?
(6) **(TP)** Estimer, pour les premières valeurs de $n$, le temps de calcul $t_n$ consommé par l'appel `suite3(n)` (voir un exemple de code ci-dessous), puis afficher les valeurs de $t_{n+1}/t_n$ pour $n = 0, 1, 2$, etc. Que remarque-t-on ? Ce comportement était-il prévisible ?

```
from time import perf_counter
t0 = perf_counter()
suite3(20)
dt = perf_counter()-t0 # temps consommé par le calcul suite3(20)
```

**Exercice 22 (les tours de Hanoï, TP).** Le problème des *tours de Hanoï* est le suivant: on dispose d'un plateau avec trois tiges (noires sur la figure ci-dessous) et $N$ disques de différents diamètres (et de différentes couleurs sur la figure). Les disques sont percés en leur centre pour pouvoir s'enfiler sur les tiges. Le but du jeu est, partant de la position initiale (à gauche), de parvenir à la position finale (à droite), en déplaçant les disques **un par un** d'une tige à une autre, avec la contrainte qu'un disque ne peut jamais être posé sur un disque plus petit. Dans la position intermédiaire ci-dessous, on ne peut donc faire que trois mouvements: déplacer le disque supérieur de la tige gauche vers la tige droite, ou déplacer le disque supérieur de la tige centrale vers l'une des deux autres tiges.


position initiale


position intermédiaire


position finale

Pour rendre l'exercice plus ludique, nous utiliserons le module `hanoi_tkinter` que vous pouvez télécharger sur la page Moodle du cours. Copiez le fichier `hanoi_tkinter.py` dans votre répertoire de travail et commencez à tester avec le programme suivant (écrit dans un fichier nommé `hanoi.py` par exemple):

```
from hanoi_tkinter import Hanoi

H = Hanoi(3,1.5) # crée l'objet H avec 3 disques et une temporisation de 1,5 seconde

H.move(0,2) # effectue un mouvement de la tige 0 (gauche) vers la tige 2 (droite)
H.move(0,1) # effectue un mouvement de la tige 0 (gauche) vers la tige 1 (milieu)
H.quit() # pour fermer la fenêtre (après appui sur la touche Entrée)
```

(1) Complétez le programme (5 mouvements à trouver) pour arriver à la position finale.

(2) On souhaite maintenant résoudre le problème des tours de Hanoï pour un nombre de disques $N$ quelconque. Pour cela, on adopte une stratégie de type *diviser pour régner* qui repose sur la remarque suivante: pour déplacer les $n$ disques supérieurs de la tige $A$ vers la tige $B$ (en supposant que les disques éventuellement déjà présents sur la tige $B$ et la tige $C$ sont plus gros que ces $n$ disques), il suffit:
- de déplacer les $n-1$ disques supérieurs de la tige $A$ vers la tige $C$;
- de déplacer le disque supérieur (initialement le $n$-ième donc) de la tige $A$ vers la tige $B$;
- de déplacer les $n-1$ disques supérieurs de la tige $C$ vers la tige $B$.

Écrire un algorithme récursif mettant en œuvre ce principe sous la forme d'une fonction solve(n,A,B,C). Quelle condition d'arrêt de la récursion peut-on utiliser ?

(3) Implémenter cet algorithme en une fonction python `solve(H,n,A,B,C)`, où `H` est l'objet python retourné par `Hanoi()` (utile pour faire les déplacements), `n` le nombre de disques à déplacer (un entier entre 1 et le nombre total de disques $N$), et `A,B,C` trois entiers distincts dans $\{0,1,2\}$ repérant respectivement la tige de départ, la tige d'arrivée et la tige restante comme dans la question précédente. Tester cette fonction `solve()` pour résoudre le problème des tours de Hanoï dans le cas de 5 disques (temporisation conseillée: 0,5 s), puis dans le cas de 10 disques (temporisation conseillée: 10 ms).

(4) **(*)** Modifiez la fonction `solve` pour qu'elle renvoie la liste des mouvements (liste de couples par exemple) au lieu de les effectuer, puis calculer le nombre de mouvements effectués pour $N$=1,2,3, etc. Que conjecturez-vous sur le nombre de mouvements effectués dans le cas de $N$ disques ?

(5) **(**)** En observant le mouvement du plus petit disque, et en remarquant qu'il n'y a qu'un seul mouvement possible lorsque le plus petit disque ne bouge pas, construisez un algorithme **non récursif** pour résoudre le problème, et testez-le.
Indication: l'appel `H.move()` lève une exception en cas de mouvement impossible ou illégal. On pourra utiliser la gestion des exceptions (instructions `try` et `except`) pour déterminer le bon mouvement lorsque le plus petit disque ne bouge pas.