CPE Lyon - 3IRC, 4ETI Année 2022/2023

Structures de données et algorithmes avancés

TP 3 - Récursivité / Diviser pour régner

Exercice 1. Une première fonction récursive

Ecrivez une fonction **récursive** qui affiche les nombres de 1 à n dans l'ordre croissant, et une autre qui les affiche dans l'ordre décroissant.

Exercice 2. Suite de Syracuse / Conjecture de Collatz

La suite de Syracuse d'une entier $N > 0$ est définie par récurrence de la manière suivante : $u_0 = N$ et

$$u_{n+1} = \begin{cases} u_n/2 & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{si } u_n \text{ est impair} \end{cases} \tag{1}$$

Par exemple :

- la suite de Syracuse de $N = 13$ est : 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1...
- la suite de Syracuse de $N = 28$ est : 28, 14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1...

On constate que quand on atteint le nombre 1, la suite boucle sur le cycle 4, 2, 1, 4, 2, 1.... La ***conjecture de Syracuse*** ou ***conjecture de Collatz*** affirme que quel que soit le nombre N de départ, on finit *toujours* par retomber sur ce cycle. Cette conjecture a été vérifiée pour tous les nombres $N < 2,95 \times 10^{20}$, mais n'a jamais pu être **prouvée** mathématiquement. Cette conjecture mobilisa tant les mathématiciens durant les années 1960, en pleine guerre froide, qu'une plaisanterie courut selon laquelle ce problème faisait partie d'un complot soviétique visant à ralentir la recherche américaine.

1. Ecrivez une fonction récursive `syracuse1` qui calcule et **renvoie sous forme d'une liste** la suite de Syracuse d'un entier donné en paramètre (s'arrêter dès qu'on tombe sur 1).
2. Modifiez la fonction précédente pour qu'elle **renvoie une liste de trois éléments** :
   - la liste des nombres de la suite ;
   - le *temps de vol*
   - l'*altitude maximale*, i.e. la valeur maximale de la suite.
3. Ecrivez à présent une fonction récursive `syracuse2` qui calcule et **renvoie** uniquement le temps de vol de la suite d'un entier donné en paramètre, **sans calculer explicitement la suite** ?
4. En utilisant le module Python `time` et la fonction `syracuse2`, chronométrez le temps mis pour calculer la **liste** des temps de vol de tous les entiers inférieurs à 10^6.

   Dans l'exemple donné ci-dessus, on constate que la suite $13, 40, 20...$, obtenue pour $N = 13$ est de nouveau calculée pour $N = 28$, mais aussi pour $N = 14, 7, 22, 34, 17, 52, 26...$; et il en est donc de même pour les temps de vols. Il serait beaucoup plus judicieux de garder en mémoire les temps de vols déjà calculés, et de les réutiliser directement ! Ainsi, au lieu de recalculer intégralement le temps de vol associé à un entier, il suffirait de vérifier s'il n'a pas déjà été calculé et stocké ! Par exemple, si on sait que le temps de vol de 20 est égal à 8, le temps de vol de 40 est simplement égal à $8 + 1 = 9$. On appelle cette technique **mémoïsation**.
5. Mettez en œuvre cette technique en utilisant une **liste** Python. Quel est désormais le temps de calcul ?
6. Même question avec un **dictionnaire** Python. Quelle solution est la plus efficace et pourquoi ?

Exercice 3. Suite de Fibonacci

La suite de Fibonacci est une suite omniprésente en mathématiques et en informatique, et se rencontre également dans la nature : $F_0 = 1, F_1 = 1$ et

$$\forall n \geq 2, F_n = F_{n-1} + F_{n-2}$$

1. Ecrivez une fonction récursive `fibonacci1` **basée uniquement sur cette définition**.
2. A l'aide du module `time`, observez l'évolution du temps de calcul pour $F_5, F_{10}, F_{20}, F_{30}$ et F_{40}. Que pouvez-vous conjecturer quant à la complexité de cet algorithme ? Pourquoi cet algorithme précédent est-il inefficace ?
3. Comme pour la suite de Syracuse, il serait plus astucieux de conserver dans un tableau les résultats des calculs déjà effectués (***mémoïsation***). Ecrivez une fonction récursive `fibonacci2` mettant en œuvre ce procédé ; comparez les temps de calcul avec `fibonacci1`. Quelle est la complexité de cet algorithme ?
4. Observez les premiers termes de la suite de Fibonacci :

$$\begin{aligned} F_2 &= 1 + 1 \\ F_3 &= 2 + 1 \\ F_4 &= 3 + 2 \\ F_5 &= 5 + 3 \\ F_6 &= 8 + 5 \end{aligned}$$

   Si on note $F_k = a + b$, quelle sera la valeur de F_{k+1} ? Déduisez-en une fonction récursive `fibonacci3` faisant apparaître une **récursion terminale**. Comparez les temps de calcul avec `fibonacci2`. Quelle est la complexité de cet algorithme ?

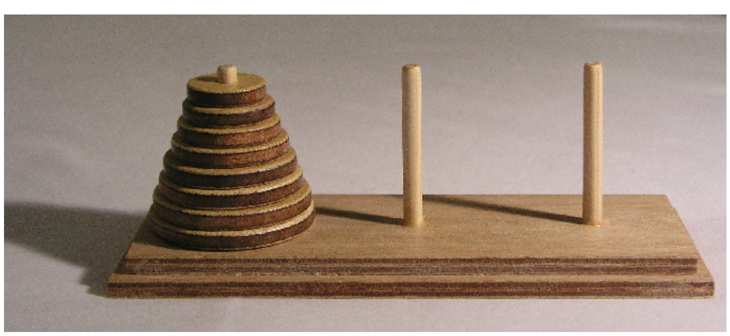
Exercice 4. Diviser pour régner

Ecrivez une fonction **`maxliste`** qui reçoit une liste en paramètre, et qui renvoie le plus grand élément de cette liste **en utilisant la technique *Diviser pour régner***.

Exercice 5. Tours de Hanoï

Les Tours de Hanoï sont un jeu de réflexion consistant à déplacer des disques de diamètres différents d'une tour de « départ » à une tour d'« arrivée » en passant par une tour « intermédiaire », tout en respectant les règles suivantes :

- on ne peut déplacer plus d'un disque à la fois ;
- on ne peut placer un disque que sur un autre disque plus grand que lui ou sur un emplacement vide.



Ecrivez une fonction récursive **`hanoi`** permettant de résoudre le problème des Tours de Hanoï à n disques. Elle affichera la solution sous la forme :

```
Déplacer un disque du pilier 1 vers le pilier 3
Déplacer un disque du pilier 1 vers le pilier 2
Déplacer un disque du pilier 3 vers le pilier 2
Déplacer un disque du pilier 1 vers le pilier 3
Déplacer un disque du pilier 2 vers le pilier 1
Déplacer un disque du pilier 2 vers le pilier 3
Déplacer un disque du pilier 1 vers le pilier 3
```

Exercice 6. Recherche dichotomique

1. La **recherche séquentielle** (ou **recherche linéaire**) consiste à parcourir les éléments d'un tableau successivement jusqu'à trouver l'élément recherché.
   (a) Quelle est la complexité de cet algorithme dans le meilleur cas ?
   (b) Quelle est la complexité de cet algorithme dans le pire des cas ?
   (c) Quelle est la complexité de cet algorithme en moyenne ?

2. La **recherche dichotomique** utilise la propriété qu'un tableau T de longueur n est déjà trié pour accélérer la recherche d'un nombre k :
   - si $T[n/2] = k$, la recherche est terminée ;
   - si $T[n/2] > k$, on recommence le processus sur la moitié gauche du tableau ;
   - si $T[n/2] < k$, on recommence le processus sur la moitié droite du tableau.

   Ecrivez une fonction `dicho` qui reçoit en paramètres une liste **d'entiers triée par ordre croissant** ainsi qu'un entier, et qui renvoie
   - l'indice de l'élément dans la liste, s'il est présent,
   - -1 sinon.

   💡 vous pouvez utiliser la méthode `sort` de Python pour trier votre liste. Nous verrons dans le Cours 4 comment fonctionne cette méthode.

3. Appliquez la méthode par itération pour calculer la complexité de votre fonction **`dicho`**, puis validez votre résultat à l'aide du Master Theorem.

Exercice 7. Master Theorem

1. Appliquez le Master Theorem aux récurrences suivantes :
   (a) $T(n) = 8T(\frac{n}{2}) + 1000n^2$
   (b) $T(n) = 2T(\frac{n}{2}) + 10n$
   (c) $T(n) = 2T(\frac{n}{2}) + n^2$

2. Expliquez pourquoi on ne peut pas appliquer le Master Theorem aux récurrences suivantes :
   (a) $T(n) = 2^n T(\frac{n}{2}) + n$
   (b) $T(n) = \frac{1}{2}T(\frac{n}{2}) + n$
   (c) $T(n) = 64T(\frac{n}{8}) - n^2 \log n$
   (d) $T(n) = T(\frac{n}{2}) + n(2 - \cos n)$
   (e) $\star$ $T(n) = 2T(\frac{n}{2}) + \frac{n}{\log n}$

3. En effectuant le changement de variable $m = \lg n$, résolvez la récurrence $T(n) = 2T(\sqrt{n}) + \lg n$ (**rappel** : lg désigne le log en base 2).