Structures de données simples en Python

mai 2019

1 Quelques problèmes de parcours de listes

Exercice 1

Écrire les fonctions :

1. `indiceMin(L)` *qui renvoie l'indice de l'élément de valeur minimale dans la liste* `L` *(supposée non vide). En cas d'égalité on renverra l'indice de la première occurrence.*
2. `liste_IndicesMin(L)` *qui renvoie la liste de tous les indices auxquels apparaît la valeur minimale de la liste.*

   Saurez-vous le faire en un seul parcours de `L` *?*
3. `croissante(L)` *qui renvoie* `True` *ou* `False` *selon que la liste* `L` *est triée en ordre croissant ou non.*
4. `palindrome(L)` *renvoie* `True` *si la liste est un palindrome : la séquence des éléments parcourue de gauche à droite est identique à la séquence des éléments parcourue dans l'ordre inverse.*

Exercice 2 (Crible d'Eratosthène)

Pour déterminer l'ensemble des nombres premiers inférieurs à n, on peut créer une liste de longueur $n+1$ (pour ne pas être embêté avec les décalages d'indices) contenant la valeur `True` *à tous les indices (tant qu'on n'a pas prouvé que n est composé, il est supposé premier), puis en « criblant » :*

- *tous les multiples (stricts) de 2 sont composés : on les passe à* `False` *dans la liste ;*
- *tous les multiples (stricts) de 3 sont composés : on les passe à* `False` *dans la liste ;*
- *pour 4, qui est composé (puisque* `L[4] == False`*), on laisse tomber (ses multiples sont déjà rayés) ;*
- *tous les multiples (stricts) de 5 sont composés : ...*
- *pour 6, qui est composé...*
- *etc.*

On continue tant que le nombre par lequel on crible n'a pas dépassé $\sqrt{n}$.

1. *Écrire une fonction prenant en entrée n et renvoyant la liste des entiers premiers inférieurs à n.*
2. *Quelle est la complexité de cette fonction ?*
3. *Combien d'entiers majorés par 10^7 sont premiers ?*

Exercice 3 (Permutations dans des listes)

On peut voir une permutation de $[\![0, n-1]\!]$ comme une liste de n entiers compris entre 0 et $n-1$ dans laquelle chacun des entiers apparaît exactement une fois. Écrire alors :

- `verifier_permutation` *qui vérifie si une liste correspond bien à une permutation.*
- `inverse` *qui calcule la réciproque d'une permutation.*
- `compose` *qui calcule la composée de deux permutations.*

2 Listes par compréhension

Exercice 4

Définir les listes suivantes ***par compréhension*** *:*

1. *les carrés de tous les entiers compris entre 1 et 100 (inclus) ;*
2. *les couples (x, y) d'entiers positifs tels que $x + y = 100$;*
3. *les couples d'entiers de la forme (x, y) tels que $x \leq y$ et les deux entiers x et y sont tous deux inférieurs à 10 (indication : il est plus facile de les ranger par y croissants) ;*
4. *les mêmes qu'à la question précédente, mais rangés par x croissants ;*
5. *les entiers compris entre 1 et 1000 qui ne sont multiples ni de 3 ni de 5.*

Exercice 5

Objectif pour tout cet exercice : la réponse à chaque question tient en 1 ligne. On fera bon usage de la fonction `sum`.

1. *Calculer* $\sum_{1 \leq i \leq j \leq 100} i^j$.
2. *On représente une matrice comme la liste de ses lignes.*
   Écrire une fonction `sommes_lignes(M)` *qui renvoie* $[S_1, \ldots, S_n]$ *où* S_i *est la somme de la* i*-ème ligne de* M.
3. *Écrire une fonction* `Id(n)` *qui renvoie la matrice identité* $I_n = \begin{pmatrix} 1 & 0 & 0 & \ldots & 0 \\ 0 & 1 & 0 & \ldots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \ldots & 1 \end{pmatrix}$.

3 Mémoïzation à l'aide de dictionnaires en Python

3.1 Échauffement

Exercice 6 (Échauffement sur les dictionnaires)

Programmer les fonctions suivantes (qui existent en fait nativement en Python) :

1. `get(dico, key, default)`
   renvoie la valeur associée à `key` *si celle-ci est présente dans* `dico`*, sinon* `default`.
2. `items(dico)`
   renvoie la liste des couples `(cle, valeur)` *du dictionnaire.*
3. `setdefault(dico, key, default)`
   Si `key` *est présente dans* `dico`*, on renvoie la valeur associée ; sinon, on ajoute cette clé avec pour valeur* `default`.

3.2 Mémoïzation simple

La *mémoïzation* consiste à stocker en mémoire une valeur pour ne pas avoir à la recalculer si elle s'avérait utile à nouveau par la suite. Nous allons en voir quelques applications.

Exercice 7 (Fibonacci)

On rappelle la définition de la suite de Fibonacci :

$$\begin{cases} F_0 &= 1 \\ F_1 &= 1 \\ F_{n+2} &= F_{n+1} + F_n \text{ pour tout } n \geq 0 \end{cases}$$

1. *Écrivez une fonction récursive calculant le* n^e *terme de la suite de Fibonacci, en traduisant directement cette définition.*
   Testez-la pour quelques valeurs de n *; à partir de quelle valeur êtes-vous en mesure de prendre une pause bien méritée ?*
2. *Dupliquez cette fonction et modifiez-la pour que :*
   — *à chaque fois qu'un terme est calculé, sa valeur soit sauvegardée dans un dictionnaire ;*
   — *à chaque fois qu'on doit calculer un terme, on vérifie tout d'abord s'il n'est pas déjà présent dans le dictionnaire.*
   (Pensez à passer le dictionnaire en argument à votre fonction.)
   Quelles sont les limites de cette seconde version ?
3. *Il est possible de programmer une version récursive efficace, basée sur*

$$\begin{cases} F_{2n} &= F_{n-1}^2 + F_n^2 \\ F_{2n+1} &= (2F_{n-1} + F_n)F_n \end{cases}$$

   Programmez-en, une fois encore, une version avec et sans dictionnaire.

4. *Mesurez expérimentalement et comparez les temps d'exécution de ces différentes versions.*
   Indication : *la fonction* `clock` *de la librairie* `time` *affiche la durée d'activité du processeur de votre machine depuis le démarrage de Python. Elle devrait vous permettre de mesurer précisément les coûts de vos fonctions.*

Exercice 8 (Syracuse)
Pour tout entier $p \in \mathbb{N}^*$, on définit la suite de Syracuse associée par

$$u_0 = p \text{ et } \forall n \in \mathbb{N},\ u_{n+1} = \begin{cases} u_n / 2 & \text{si } u_n \text{ pair} \\ 3u_n + 1 & \text{sinon} \end{cases}$$

On admet que pour toute valeur raisonnable de p [1]*, la suite finit par boucler sur $1, 4, 2$. On note $vol(p)$ le plus petit n tel que $u_n = 1$, lorsque $u_0 = p$.*

1. *Quelle est la plus grande valeur de $vol(p)$ pour $1 \leq p \leq 1000$? Et jusqu'à $p = 10^7$?*
   Indice dont vous vous doutiez : *une attaque par force brute de ce problème est vouée à l'échec...*
2. *Combien d'entiers $p \leq 10^6$ vérifient $vol(p) \leq 30$?*
3. Optionnel : *combien d'entiers $p \in \mathbb{N}^*$ vérifient $vol(p) \leq 50$?*

3.3 Plus longue sous suite commune

Le problème On définit une sous-suite de A comme étant un mot de la forme $a_{i_1} a_{i_2} \dots a_{i_k}$ avec $i_1 < i_2 < \dots < i_k$, autrement dit une suite de lettres apparaissant dans le même ordre dans A mais non nécessairement consécutives.

On considère deux mots (sur un alphabet quelconque)

$$A = a_1 \dots a_n \quad \text{et} \quad B = b_1 \dots b_m$$

On recherche alors les sous-suites communes à A et B, et plus particulièrement la (ou une des) plus longue. Exemple : $A = aabaababaa$, $B = ababaaabb$ alors leur plus longue sous-suite commune (PLSSC) est $ababaaa$.

Résolution On généralise le problème à tous les préfixes de A et de B, et on note $p(i, j)$ la longueur de la plus longue sous-suite commune à $a_1 \dots a_i$ et $b_1 \dots b_j$.

On démontre assez facilement que

$$p(i,j) = \begin{cases} 1 + p(i-1, j-1) & \text{si } a_i = b_j \\ \max\left(p(i, j-1),\ p(i-1, j)\right) & \text{sinon} \end{cases}$$

Il est donc possible de résoudre le problème par une approche récursive, mais avec une écriture naïve de l'algorithme, de nombreuses valeurs $p(i, j)$ risquent d'être calculées plusieurs fois.

Exercice 9 (Plus longue sous-suite commune) *Adapter la technique de mémoïzation pour calculer $p(i, j)$* ***efficacement*** *tout en conservant une écriture récursive.*

Exercice 10 (Cerise sur le gâteau) *Comment adapter l'algorithme précédent pour calculer non seulement la longueur de la plus longue sous-suite commune mais aussi en exhiber une ?*

1. Aux dernières nouvelles, cela a été vérifié jusqu'à $p = 5 \times 2^{60} \simeq 5.8 \times 10^{18}$.