TP n° 3 : Récursivité

MP Lycée Camille Jullian

19-26 novembre 2021

Ce TP a bien sûr pour but de vous entraîner à écrire des programmes récursifs, mais aussi par la même occasion de continuer à réviser la programmation par boucles plus classique. La plupart du temps, les exercices peuvent être traités des deux façons, et il est conseillé d'essayer d'écrire les deux versions du programme !

Exercice 1 : un gentil classique pour débuter

On souhaite écrire un programme qui affiche à l'écran une suite de lignes contenant un nombre croissant d'étoiles jusqu'à un certain entier n fixé comme paramètre (ici pour $n = 5$) :

```
*
**
***
****
*****
```

Écrire un programme Python effectuant cette tâche (en deux versions, l'une récursive et l'autre non, si possible). En écrire ensuite un autre qui fasse une suite de lignes contenant un nombre décroissant d'étoiles de n à 1, puis un dernier avec un nombre d'étoiles décroissant de n à 1 puis croissant de 1 à n.

Exercice 2 : sous-listes d'une liste donnée

Écrire un programme Python affichant la liste de toutes les sous-listes d'une liste l donnée en argument (un programme récursif est plus facile à mettre en oeuvre ici).

Exercice 3 : recherche dichotomique dans une liste triée.

Tout est dans le titre, on veut programmer de façon récursive l'algorithme consistant à chercher la présence d'un élément dans une liste triée par ordre croissant en procédant par dichotomie.

Exercice 4 : calcul de coefficients binomiaux.

Pour calculer le coefficient binômial $\binom{n}{k}$, il existe (au moins) trois méthodes classiques possibles :

- utiliser la définition comme quotient de factorielles $\binom{n}{k} = \dfrac{n!}{k!(n-k)!}$
- utiliser la formule $k\binom{n}{k} = n\binom{n-1}{k-1}$, et appliquer récursivement cette formule jusqu'à obtenir un coefficient trivial à calculer du type $\binom{p}{0}$ (celle-ci est naturellement adaptée à une écriture sous forme de programme récursif).

- utiliser le triangle de Pascal basé sur la formule $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$. Naturellement, si on utilise cette formule, il faudra éviter le piège vu en cours avec la suite de Fibonacci de l'algorithme exponentiel. Le plus simple est de créer à chaque étape la liste de tous les coefficients binômiaux, et donc d'écrire un programme qui renvoie simplement la ligne numéro n du triangle de Pascal (programme récursif ou non, les deux sont programmables ici).

L'exercice consiste évidemment à programmer les trois méthodes.

Exercice 5 : des compléments sur l'arithmétique.

On a déjà donné en cours l'exemple de l'algorithme d'Euclide du calcul de pgcd de deux entiers naturels. Dans cet exercice, on va creuser un peu plus dans cette direction.

1. Le théorème de Bézout assure, pour tout couple (a, b) d'entiers naturels non nuls, l'existence et l'unicité d'un couple d'entiers relatifs (u, v) tel que $au+bv = pgcd(a, b)$. Écrire un programme Python donnant le résultat de l'algorithme d'Euclide étendu, c'est-à-dire à la fois le pgcd des deux entiers a et b donnés en argument, et les deux entiers u et v correspondants. Un tout petit peu de maths seront nécessaires pour trouver comment calculer les valeurs de u et v, au pire vous trouverez facilement des versions de cet algorithme sur le web, le but n'est pas de juger vos capacités en arithmétique ici...
2. Écrire une fonction Python déterminant si un entier naturel est premier ou non (pas de récursivité ici).
3. Écrire une fonction Python renvoyant le plus petit facteur premier d'un entier naturel donné (ici aussi, on pourra sa passer de récursivité).
4. Écrire enfin un programme Python renvoyant la décomposition en facteurs premiers d'un entier donné. On renverra par exemple la liste des facteurs premiers (répétés s'ils apparaissent à une puissance supérieure à 1 dans la décomposition). Ici, on fera (au moins) une version récursive du programme.

Exercice 6 : le problème classique du rendu de monnaie.

Le problème est le suivant : dans une monnaie donnée on dispose de pièces ou billets de certaines valeurs, comment rendre la monnaie lors d'un achat de façon optimale (en utilisant le moins de pièces ou billets possible) ? Les valeurs disponibles seront représentées en Python par une liste présentée par ordre décroissant (ainsi, avec les euros actuels, la liste serait $[200, 100, 50, 20, 10, 5, 2, 1]$).

1. Le premier algorithme proposé pour obtenir une somme n avec les éléments d'une liste L est le suivant : on sélectionne dans la liste L la valeur la plus grande qui soit inférieure ou égale à n, on la stocke et on soustrait cette valeur de n. Tant qu'on a pas atteint 0 (ce qui sera évidemment le cas quand la somme des valeurs sélectionnéess sera égale à n), on recommence. Écrire un programme Python mettant en oeuvre cet algorithme (ici, les deux versions, récursives ou impérative, sont faciles à programmer).
2. Trouver un exemple concret où l'algorithme précédent n'est pas optimal.
3. Proposer un algorithme récursif permettant d'assurer l'optimalité du résultat et le pogrammer. Tester sur des valeurs pas trop grandes de n. Votre programme conviendra-t-il pour de grandes valeurs de n ?

Exercice 7 : perles de Dijkstra.

On dispose de perles de trois couleurs (qu'on symbolisera par 0, 1 et 2 car c'est ce qu'on fera en pratique en Python), et l'on souhaite créer un collier de perles de longueur n ne contenant jamais deux séquences de perles identiques à la suite. Par exemple, pour $n = 4$, on ne peut pas faire 0211 (on aurait deux fois la séquence '1' à la suite), mais pas non plus 0202 (deux fois la séquence '02'

à la suite). Une solution possible est 0201 (il en existe d'autres). Le but de l'exercice est d'écrire un programme affichant une suite de longueur n vérifiant ces conditions. Pour cela, on impose le principe suivant : on va écrire une fonction Python récursive **perles(s,n)** prenant comme arguments une chaine de caractère s et un entier n correspondant au nombre de perles à ajouter à la chaine s pour atteindre la longueur souhaitée. L'algorithme va ensuite fonctionner comme ceci :

- on note x_1 et x_2 les deux valeurs différentes de la valeur finale de la chaîne s.
- si $s+[x_1]$ vérifie la condition de Dijkstra **et** que perles$(s+[x_1], n-1)$ renvoie un succès, alors on renvoie un succès, et la chaîne de caractères obtenue.
- si $s+[x_2]$ vérifie la condition de Dijkstra **et** que perles$(s+[x_2], n-1)$ renvoie un succès, alors on renvoie un succès, et la chaîne de caractères obtenue.
- sinon, on renvoie un échec.

Bien sûr, il faudra écrire un programme annexe testant la condition de Dijkstra sur une chaîne quelconque. Si le programme est correctement écrit, perles([0],n-1) renverra la liste souhaitée.