TP

listes Python (tableaux)

11 juin 2019

Faites tous les exercices dans un même fichier `TP-liste.py`. Ecrire un code propre et clair (comme une correction pour les élèves), donner des noms de variable parlants. Testez toutes vos fonctions au fur et à mesure pour les vérifier (avec des valeurs refléchies !). Laissez les tests en dur dans le programme pour que je puisse les voir, mais mettez les en commentaire (avec #) quand vous passez à l'exercice suivant.

Exercice 1 : Moyenne

Écrivez une fonction `moyenne` qui prend en argument une liste et qui renvoie la moyenne des éléments contenus dans la liste.

Exercice 2 : Recherche

Pour chaque item suivant, écrivez une fonction qui prend en argument une liste ℓ et une valeur x et renvoie :

1. `True` si ℓ contient x, et `False` sinon
2. le premier indice d'occurrence de x dans ℓ, ou `False` si un tel indice n'existe pas.
3. le dernier indice d'occurrence de x dans ℓ, ou `False` si un tel indice n'existe pas.
4. le nombre d'occurrences de x dans ℓ.

On nommera les fonctions `contient`, `premierIndice`, `dernierIndice` et `nbOccurrences`.

Exercice 3 : Maximum

Écrivez une fonction `maximum` qui prend comme argument une liste d'entiers et renvoie l'élément maximal de la liste.

Exercice 4 : TestTri

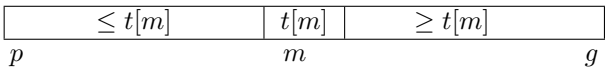
Écrivez une fonction `testTri` qui prend comme argument une liste d'entiers et teste si cette liste est triée en ordre croissant.

Exercice 5 : Dichotomie

Lorsque la liste t dans laquelle on recherche une valeur v est triée, il est plus efficace de procéder en utilisant un algorithme appelé recherche par dichotomie plutôt que la recherche linéaire utilisée à l'exercice 5.

L'idée générale de la recherche par dichotomie est de comparer la valeur v recherchée à un élément $t[m]$ qui se situe au milieu de la liste. Ainsi si v est supérieur à $t[m]$, on sait qu'il est inutile de chercher v au début de la liste, et on peut se concentrer sur les indices supérieurs à m pour continuer notre recherche. De même si v est inférieur à $t[m]$, on sait qu'il est inutile de chercher v à la fin de la liste, et on peut se concentrer sur les indices inférieurs à m pour continuer notre recherche. Ainsi chaque étape permet de diviser par deux la taille de la sous-liste dans laquelle on cherche, ce qui est bien plus efficace que de tester tous les éléments un par un.

Plus précisément, la recherche par dichotomie se fait en manipulant trois indices $p \leq m \leq g$ (pour « petit », « moyen » et « grand ») : p et g délimitent la partie de la liste dans laquelle on est en train de chercher, et m désigne l'indice de l'élément du milieu auquel comparer la valeur v recherchée.



Initialement, p vaut 0, et g vaut la taille de la liste moins un. À chaque étape du calcul, m vaut $(p+g)/2$. Si $t[m]$ vaut v (la valeur recherchée), alors l'algorithme termine. Dans le cas contraire, si $t[m] < v$, alors p prend la valeur $m+1$, sinon g prend la valeur $m-1$ (car alors on sait que $t[m] > v$). Si $p > g$, alors l'algorithme termine.

Écrivez une fonction `rechercheDichotomie` qui prend en paramètre une valeur et une liste supposée triée (vous n'avez pas à le vérifier), et qui utilise une recherche par dichotomie pour trouver un indice d'occurrence de la valeur dans la liste (et renvoie `False` si un tel indice n'existe pas).

Exercice 6 : Création de liste

- Écrivez une fonction `liste` qui demande à l'utilisateur un entier n, puis n autres entiers (l'utilisateur les saisis au clavier), et qui renvoie une liste contenant ces n entiers.
- Écrivez une fonction `inverse` qui prend en argument une liste ℓ et renvoie une nouvelle liste dont les éléments sont les éléments ℓ mais dans l'ordre inverse de celui dans lequel ils sont dans ℓ. (Ne pas utiliser la méthode `reverse` de Python : le but est de faire comme si vous êtes le programmeur qui écrit cette méthode.)
- En utilisant les fonctions précédentes, écrivez un programme qui demande à l'utilisateur un entier n, puis n autres entiers, puis qui affiche ces n entiers dans l'ordre inverse de celui dans lequel ils ont été entrés. Remarquez qu'une seule ligne de code suffit. Mettez ce programme en commentaire avant de passer à l'exercice suivant.

Exercice 7 : Fusion de tableaux triés

Écrire une fonction `fusion` qui prend en argument deux tableaux `t1` et `t2` que l'on supposera triés dans l'ordre croissant, et qui renvoie un nouveau tableau `t` qui est trié dans l'ordre croissant et qui contient tous les éléments de `t1` et de `t2`. Votre fonction doit utiliser une méthode efficace.

Exercice 8 : Tri par sélection

Le tri par sélection est un algorithme permettant de trier un tableau en place, c'est-à-dire sans se servir d'un tableau additionnel. Le principe est le suivant : on commence par chercher l'élément minimal du tableau, et on le met dans la première case du tableau (en l'échangeant avec la valeur qui occupait cette case). Puis on cherche l'élément minimal parmi ceux qui restent, et on le met dans la deuxième case du tableau, etc.

On remarque que lors tri par sélection, à chaque itération i une portion du tableau t a déjà été triée : les éléments d'indice compris entre 0 et $i-1$. A l'étape suivante on va donc chercher le minimum des éléments restants à trier (à partir de l'indice i), et on échange ce minimum avec $t[i]$. La portion du tableau d'indice compris entre 0 et i sera alors triée. On incrémente i jusqu'à atteindre la fin du tableau.

Coder une fonction `triSelection` qui prend en argument un tableau et qui le trie en utilisant l'algorithme de tri par sélection.

Exercice 9 : Tri par insertion

Comme le tri par sélection, le tri par insertion est un algorithme qui permet de trier un tableau en place. Cependant le tri par insertion est en moyenne plus efficace que le tri par sélection.

Le tri par insertion d'un tableau `t` de taille n manipule un indice i, qui délimite dans le tableau deux parties : de 0 à $i-1$, partie déjà triée ; de i à $n-1$, partie restant à trier.

Initialement, i vaut 1. À chaque étape, on compare l'élément `t[`i`]` à chacun, successivement, des éléments `t[`j`]` situés à sa gauche, en commençant par le plus proche. Lorsqu'on trouve le premier (donc le plus grand) indice j tel que `t[`j`]` est inférieur à `t[`i`]` (ou lorsqu'un arrive à $j = -1$), on insère `t[`i`]` « après » `t[`j`]` en décalant les éléments `t[`$j+1$`]` ... `t[`$i-1$`]`. On incrémente ensuite i de 1 à chaque étape, jusqu'à atteindre la fin du tableau.

Exemple : tri par insertion du tableau [5, 3, 1, 2, 4] :

5	**3**	1	2	4	Etape 1 : on insère 3 dans [5].
3	5	**1**	2	4	Etape 2 : on insère 1 dans [3,5].
1	3	5	**2**	4	Etape 3 : on insère 2 dans [1,3,5].
1	2	3	5	**4**	Etape 4 : on insère 4 dans [1,2,3,5].
1	2	3	4	5	Fin : le tableau est trié.

Coder une fonction `triInsertion` qui prend en argument un tableau et qui le trie en utilisant l'algorithme de tri par insertion. Vérifier que votre fonction est correcte grâce à plusieurs tests pertinents, en particulier dans le cas où la même valeur apparaît plusieurs fois dans le tableau donné en entrée.

Listes de listes (= tableaux à deux dimensions = matrices)

Exercice 10 : Incrémente

Écrivez une fonction `incremente` qui prend en argument un tableau bidimensionnel (c'est-à-dire une liste de listes) contenant des entiers, et qui ajoute 1 à chaque entier du tableau.

Exercice 11 : Recherche2

Écrivez une fonction `recherche2` qui prend en argument un tableau bidimensionnel (c'est-à-dire une liste de listes) et une valeur et teste si cette valeur apparaît dans une des cases du tableau.

Exercice 12 : Carré magique

Un carré magique est une matrice carrée telle que la somme de chaque rangée, de chaque colonne et de chaque diagonale a la même valeur. Un carré magique de n lignes est dit normal s'il contient chaque entier compris entre 1 et n^2 exactement une fois. Par exemple, le tableau suivant est un carré magique normal :

6	7	2
1	5	9
8	3	4

Dans la suite on écrit des fonctions pour tester si un tableau bidimensionnel est un carré magique normal.

1. Écrivez une fonction `carre` qui teste si un tableau bidimensionnel d'entiers donné en paramètre est une matrice carrée (c'est-à-dire un tableau dont toutes les lignes et les colonnes ont même longueur).
2. Écrivez deux fonctions `sommeLigne` et `sommeColonne` qui prennent en argument un entier i et une matrice carrée et qui renvoient : pour `sommeLigne`, la somme des éléments de la ie ligne du tableau, et pour `sommeColonne`, la somme des éléments de la ie colonne du tableau.
3. Écrivez deux fonctions `sommeDiagonaleMajeure` et `sommeDiagonaleMineure` qui prennent en argument une matrice carrée et renvoient respectivement la somme de la diagonale majeure, et celle de la diagonale mineure du tableau passé en paramètre.
4. Écrivez une fonction `carreMagique` qui prend en argument un tableau bidimensionnel, et teste s'il s'agit d'un carré magique (pas forcément normal).
5. Pour savoir si un carré magique est normal, il faut compter le nombre de fois qu'apparaissent ses éléments. Écrivez une fonction `histogramme` qui prend en argument un tableau bidimensionnel t et qui renvoie l'histogramme du tableau t, c'est-à-dire la liste h de taille n^2 (avec n le nombre de lignes de t) tel que pour tout $i < n^2$, $h[i]$ contient le nombre d'occurrences de la valeur $i+1$ dans le tableau t (on peut aussi choisir de construire h de taille n^2+1 afin que pour $1 \leq i \leq n^2$, $h[i]$ contient le nombre d'occurrences de la valeur i dans le tableau t). Par exemple pour un carré magique 3×3, son histogramme aura 9 cases, pour compter le nombre d'apparition des éléments compris entre 1 et 9.
6. Écrivez une fonction `carreMagiqueNormal` qui prend en argument un tableau bidimensionnel, et teste s'il s'agit d'un carré magique normal.

Exercice 13 : Sous-tableau

Écrivez une fonction `sousTableau` qui prend prend en argument deux tableaux bidimensionnels et qui teste si le premier est contenu dans le deuxième (c'est à dire si on peut découper un rectangle dans le second qui soit égal au premier).