Remise en route

Thèmes : listes, tableaux, parcours, tri récursif, codage binaire d'un entier positif

1 Trois plus grandes valeurs

On souhaite déterminer les trois plus grandes valeurs présentes dans un tableau. On testera les fonctions de cet exercice à l'aide d'un tableau `numpy` aléatoire qu'on peut obtenir de la façon suivante :

```
import numpy as np
t = np.random.randint(bas, haut, taille)
```

Question 1. On commence par s'attaquer à un problème plus simple. Écrire une fonction `valmax(t)` prenant en entrée un tableau (ou une liste) non vide et renvoyant la plus grande valeur de ce tableau. *On ne fera bien sûr pas usage de la fonction native `max`.*

Question 2. Écrire une fonction `val3max(t)` prenant en entrée un tableau (ou une liste) contenant des entiers positifs et renvoyant un triplet (a, b, c) contenant les 3 valeurs les plus grandes apparaissant dans le tableau avec $a < b < c$. Par exemple sur la liste $[1, 5, 7, 4, 2, 3, 1, 7]$ la réponse sera $(4, 5, 7)$, même si la valeur 7 apparait deux fois. La complexité de votre solution devra être en $O(n)$ où n est la longueur de la liste.

Remarque : Le triplet résultat pourra contenir des valeurs -1 s'il n'y a moins de 3 valeurs distinctes dans la liste. Par exemple sur la liste $[4, 4, 4, 4, 4]$ le résultat sera $(-1, -1, 4)$.

Indications. Se baser sur ce qui a été fait pour `valmax`. Lors du parcours du tableau :

- mémoriser dans 3 variables a, b, c les trois plus grandes valeurs trouvées jusqu'à présent ;
- pour chaque valeur rencontrée lors du parcours, tester s'il convient de modifier a, b, et/ou c ;
- a, b, c pourront être initialisées avec la valeur -1.

2 Tri par partition-fusion

On souhaite trier une liste selon le principe *diviser pour régner* en utilisant l'algorithme du *tri par partition-fusion*. Pour trier une liste, par exemple $\ell = [1, 7, 3, 4, 2, 10, 5]$, on procède ici en trois étapes :

a) on partitionne en deux listes $\ell_0 = [1, 3, 2, 5]$ (contenant les cases de ℓ d'indice pair) et $\ell_1 = [7, 4, 10]$ (contenant les cases de ℓ d'indice impair) ;
b) on trie *récursivement* les listes ℓ_0 et ℓ_1, on obtient $u_0 = [1, 2, 3, 5]$ et $u_1 = [4, 7, 10]$;
c) on fusionne les listes u_0 et u_1, obtenant ainsi $u = [1, 2, 3, 4, 5, 7, 10]$ qui correspond bien au tri de ℓ par ordre croissant.

Question 3. Écrire une fonction `partition(l)` prenant en entrée une liste d'entiers et renvoyant le couple (ℓ_0, ℓ_1) défini ci-dessus.

Indications. Débuter initialement avec $\ell_0 = \ell_1 = []$, puis parcourir la liste. Lors du parcours, ajouter les valeurs rencontrées dans ℓ_0 ou dans ℓ_1 selon la parité de l'indice.

Question 4. Écrire une fonction `fusion(u0, u1)` prenant en arguments deux listes u_0 et u_1 *supposées déjà triées par ordre croissant* et renvoyant une liste u triée par ordre croissant contenant les valeurs de u_0 et u_1. On utilisera l'algorithme suivant :

FUSION(u_0, u_1):
$(u, i_0, i_1) \leftarrow ([], 0, 0)$
tant que i_0 et i_1 sont des indices valides **faire**
  si $u_0[i_0] \leq u_1[i_1]$ **alors**
    ajouter $u_0[i_0]$ dans u
    incrémenter i_0
  sinon
    ajouter $u_1[i_1]$ dans u
    incrémenter i_1
renvoyer $u + u_0[i_0 :] + u_1[i_1 :]$ (concaténation)

Question 5.

a) Écrire une fonction de test vérifiant le bon fonctionnement de la fonction fusion à l'aide d'assertions.
b) Donner un argument permettant de justifier que la fonction `fusion` termine.
c) Écrire une fonction `est_triée(l)` renvoyant `True` si et seulement si la liste `l` est triée par ordre croissant.
d) Ajouter une ou des assertions pour tester la validité des entrées dans la fonction `fusion`.

Question 6.

a) Écrire une fonction récursive `tri_fusion(l)` prenant en entrée une liste `l` et renvoyant une nouvelle liste `u` correspondant au tri par ordre croissant de `l`.

   Indications. Utiliser l'algorithme proposé et les fonctions précédentes. Bien identifier les cas de base de la récursivité.

b) Tester votre fonction de tri. On rappelle que la complexité temporelle de cet algorithme est $O(n \log n)$ et qu'il n'existe pas de meilleure complexité pour un tri par comparaisons.

3 Entiers conjugués

Un entier `n` $\in [\![0, 255]\!]$ peut être représenté de façon unique à l'aide d'un tableau de 8 bits (il s'agit donc d'un octet) où chaque bit représente une puissance de 2 :

1	2	4	8	16	32	64	128

Ainsi, l'entier 41 sera représenté sous forme de tableau de bits de la façon suivante :

1	0	0	1	0	1	0	0

car $41 = 32 + 8 + 1$.

Question 7.

a) Écrire une fonction `décodage(t)` prenant en entrée un tableau (numpy) `t` de 8 bits et renvoyant l'entier `n` qu'il représente.

b) Inversement, écrire une fonction `codage(n)` prenant en entrée un entier `n` $\in [\![0, 255]\!]$ et renvoyant le tableau (numpy) de son codage sur un octet.

Indication. La case d'indice `i` sera obtenue en étudiant la parité de `n // (2**i)`.

On appelle *rotation droite* l'opération consistant à transformer un tableau de la forme

a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7

en le tableau

a_7	a_0	a_1	a_2	a_3	a_4	a_5	a_6

Question 8. Écrire une fonction `rotation(t)` réalisant la rotation droite d'un tableau numpy `t` donné : cette fonction modifiera le tableau passé en argument et ne renverra aucun résultat.

Deux tableaux seront dits *conjugués* si on peut obtenir l'un à partir d'un nombre quelconque de rotations droite de l'autre. Deux entiers seront dits conjugués si leurs représentations sous forme d'octet sont deux tableaux conjugués.

Question 9.

a) Écrire et tester une fonction prenant en argument deux entiers et testant s'ils sont conjugués.

b) Écrire une fonction prenant en argument un entier `n` $\in [\![0, 255]\!]$ et affichant tous les entiers `m` $\in [\![0, 255]\!]$ tels que `n` et `m` sont conjugués.