Remise en jambe

I Algorithmique sur une structure linéaire.

Exercice 1. *Recherche linéaire - dernière position*

Implémenter une fonction `recherche_lineaire_fin` renvoyant l'indice de la dernière occurrence d'un mot dans une liste de mots.

Exercice 2. *Recherche linéaire - toutes les positions*

Implémenter une fonction `recherche_lineaire_tout` renvoyant la liste des indices de chaque occurrences d'un mot dans une liste de mots.

Exercice 3. *Calcul de la somme maximale*

Implémenter deux fonctions `somme_max` et `somme_max_lineaire` renvoyant la plus grande somme de deux éléments parmi une liste d'entiers. La première implémentera un algorithme naïf en $O(n^2)$. La seconde proposera une solution en $O(n)$.

Exercice 4. *Aplatissement*

Implémenter une fonction `aplatissement` qui prend en argument une liste de liste d'entiers et renvoie la concaténation de l'ensemble des listes dans l'ordre. Donner la complexité temporelle de votre algorithme en fonction de la taille de l'entrée.

Exemple : Si l'entrée est `[[1,2],[8,3,2],[1],[],[2,3]]` la sortie sera : `[1,2,8,3,2,1,2,3]`

Exercice 5. *Aplatissement de dictionnaires*

Implémenter une fonction `aplatissement_dict` avec les spécifications suivantes :

- **Entrée** : une liste de dictionnaires dont les clés sont des chaînes de caractère et les valeurs sont des listes d'entiers.
- **Sortie** : un dictionnaire dont l'ensemble de clés est exactement l'ensemble des clés de tous les dictionnaires de la liste d'entrée, et la valeur associée à une clé est la concaténation des valeurs associées à cette clé dans chaque dictionnaire où la clé est définie.

Exemple : Si l'entrée est

```
L = [{'a'  : [1,2],
      'ba' : [4] },
     {'a'  : [ 6,9,0,1 ],
      'c'  : [ 3 ] },
     {'ba' : [ 4,6 ] } ]
```

la sortie sera :

```
{ 'a' : [ 1,2,6,9,0,1 ],
  'ba' : [ 4,4,6 ],
  'c' : [ 3 ] }
```

Exercice 6. *Montagne*

Une montagne est une liste non vide d'entiers qui croit strictement puis, éventuellement, décroît strictement. Le pic est l'entier le plus grand d'une montagne. Implémenter une fonction `recherche_pic` qui prend en entrée une montagne est renvoie un couple (i, p) avec i l'indice du pic de la montagne et p sa valeur. **L'algorithme utilisé devra avoir une complexité logarithmique**.

Proposer un jeu de tests pour vérifier la correction de votre programme.

Montrer la terminaison de l'algorithme.

Exercice 7. *Montagne avec plateau*

Une montagne avec plateau est une liste non vide d'entiers qui croit strictement, stagne éventuellement, puis, éventuellement, décroît strictement. Le plateau débute à l'indice i de la liste tel que, de l'indice 0 à i la liste soit strictement croissante et décroissante de i au dernière indice de la liste. Le plateau se termine à l'indice j de la liste tel que, de l'indice 0 à j la liste soit croissante et strictement décroissante de j au dernier indice de la liste. Le pic d'une montagne se situe au milieu du plateau. Implémenter une fonction `recherche_pic_plateau` qui prend en entrée une montagne avec plateau est renvoie un couple (i, p) avec i l'indice du pic de la montagne et p sa valeur. **L'algorithme utilisé devra avoir une complexité logarithmique**.

Proposer un jeu de tests pour vérifier la correction de votre programme.

Exercice 8. *Ordonnancement Glouton*

Implémenter une fonction `ordonnancement` qui prend en paramètre une liste d'entiers L et réparti en temps linéaire, à l'aide d'une heuristique gloutonne, chaque éléments de L dans deux listes L_1 et L_2 en essayant de minimiser la différence entre la somme des éléments de L_1 et L_2. La fonction renverra dans un couple les deux listes.

Trouver un exemple sur lequel votre algorithme ne renvoie pas la répartition optimale.

Exercice 9. *Ordonnancement Optimal*

Implémenter une fonction `ordonnancement_optimal` qui prend en paramètre une liste d'entiers L et réparti chaque éléments de L dans deux listes L_1 et L_2 en minimisant la différence entre la somme des éléments de L_1 et L_2. La fonction renverra dans un couple les deux listes.

Prouver la correction et la terminaison de votre fonction.

II Tris

Exercice 10. *Tri par comptage*

Implémenter une fonction `tri_lineaire` qui prend en entrée une liste d'entiers et une borne supérieur de l'ensemble des valeurs de la liste, et renvoie une copie de la liste triée en temps linéaire.

Exercice 11. *Tri par insertion*

Implémenter la fonction `tri_insertion` qui prend en entrée un liste d'entier et ordonne ses éléments par effet de bord selon l'algorithme du tri par insertion.

Exercice 12. *Tri fusion*

Implémenter la fonction `tri_fusion` qui prend en entrée un liste d'entier et renvoie une liste triée selon l'algorithme du tri fusion.

III Récursivité

Les algorithmes de cette partie devront être implémenté par une méthode récursive

Exercice 13. *Exponentiation rapide*

Implémenter une fonction `puissance` qui implémente l'algorithme d'exponentiation rapide.

Exercice 14. *Conversion binaire*

Implémenter une fonction `binaire` qui prend en entrée un entier relatif, `n`, et un nombre de bit, `k`, et converti `n` en binaire en complément à deux sur `k` bits. Le résultat sera sous la forme d'une chaîne de caractères représentant l'écriture binaire du nombre.

Exercice 15. *Fibonacci*

Implémenter une fonction `fibonnaci` qui calcul le $n^{\text{ème}}$ terme de la suite de Fibonacci **en temps linéaire**.

Exercice 16. *Calcul approché de $\sqrt{2}$*

Implémenter deux fonctions `u` et `v`, mutuellement récursives, qui calculent les $n^{\text{ème}}$ termes des suites récurrentes croisées suivante :

$$\begin{cases} u_0 = 1 & v_0 = 2 \\ u_{n+1} = \frac{2\,u_n\,v_n}{u_n+v_n} & v_{n+1} = \frac{u_n+v_n}{2} \end{cases}$$

Exercice 17. *Tours de Hanoï*

Un entrepot se fait livrer des cartons chaque jour dans la zone de dépôt A. Les cartons sont empilés du plus lourd au plus léger. Chaque pile doit être déplacée par un automate dans la zone de traitement C. L'automate dispose d'une zone de transit B pour effectuer le déplacement. L'automate ne peut prendre que le carton sur le sommet de la pile et ne doit jamais empiler un carton sur un carton plus léger que lui.

Implémenter une procédure qui affiche la séquence d'instruction à envoyer à l'automate pour réaliser le déplacement de la pile en fonction du nombre de cartons livrés. L'automate comprend des instructions de la forme $n \to m$ où n est la zone de départ du déplacement et m la zone d'arrivée.

e.g. :

$$\begin{array}{c} \text{A} \to \text{B} \\ \text{A} \to \text{C} \\ \text{B} \to \text{C} \end{array}$$

IV Traitement d'images

On donne ici les instructions de manipulation d'images. On supposera que la chaîne de caractère donnant le chemin de l'image est accessible via la variable `chemin`.

```python
from PIL import Image
import numpy as np

# Ouverture d'une image via son chemin.
# L'instruction renvoie un objet Image que l'on affecte à la variable img
img = Image.open(chemin)

# Conversions de l'image
# On peut convertir l'objet Image, pour avoir une image en niveau de gris ou
# en noir et blanc. On obtient un nouvel objet Image.

# Conversion en noir et blanc, "1" pour "un seul bit" d'information.
# La matrice contient des entiers des booléens vrai ou faux. Vrai pour blanc.
img_nb = img.convert("1")
# Conversion en niveau de gris, "L" pour "Luminance".
# La matrice contient des entiers entre 0 (noir) et 255 (blanc)
img_ng = img.convert("L")
# Conversion en couleur, "RGB" pour "Red Green Blue".
# La matrice contient des triplets d'entiers entre 0 (éteint) et
# 255 (intensité maximale).
img_rgb = img.convert("RGB")

# Récupération d'une matrice depuis l'image.
# Ici c'est la matrice de img, mais on pourrait mettre img_bn, img_ng ...
matrice = np.asarray(img).tolist()

# Création d'un objet Image depuis une matrice.

# Noir et blanc. La matrice contient des booléens.
n_img_nb = Image.fromarray(np.asarray(img_nb, dtype=np.bool))
# Niveau de gris. La matrice contient des entiers entre 0 et 255.
n_img_ng = Image.fromarray(np.asarray(img_ng, dtype=np.uint8))
# Couleur. La matrice contient des triplets d'entiers entre 0 et 255.
n_img_rgb = Image.fromarray(np.asarray(img_rgb, dtype=np.uint8))

# Visualisation de l'image
n_img_ng.show() # Visualisation externe
n_img_ng.save(autre_chemin) # Sauvegarde dans un fichier
```

Exercice 18. *Identification des composantes connexes*

On considère des images en noir et blanc représentés par une matrice de booléens. Chaque groupement de pixels noirs forment une composante connexe que l'on cherche à identifier. Par groupement on veut dire, un ensemble de pixel noir maximal tel que chaque pixel est adjacent (éventuellement en diagonal) à un autre pixel de l'ensemble.

Implémenter une fonction `etiquettage_composante` qui prend une image noir et blanc et renvoie une copie où chaque composante connexe est étiquetée, *i.e.* un numéro de 1 à n (le nombre de composantes) est attribué à chaque composante et la valeur de chaque pixel de cette composante dans l'image est celle de l'étiquette.

Exercice 19. *Filtrage*

On considère des images RGB représentés par une matrice de triplés. Un filtre est une matrice 3×3. Pour appliquer un filtre sur une image il faut l'appliquer à chaque pixel. C'est à dire multiplier la matrice avec le triplé du pixel, vu comme un vecteur de trois coordonnées.

Implémenter une fonction `filtrage` qui prend une image et un filtre et renvoie une copie de l'image sur laquelle a été appliqué le filtre.

Exercice 20. *Flou Gaussien*

On considère des images en niveau de gris représentés par une matrice d'entiers entre 0 et 255.

Appliquer un flou gaussien sur une image en niveau de gris revient à appliquer un filtre passe-bas. On transforme chaque pixel en le résultat d'un produit de convolution entre le voisinage du pixel et un noyau gaussien.

Le noyau gaussien est une matrice, $\mathrm{H} = (h_{i,j})_{0 \leq i,j \leq 2r}$, de taille $2r + 1$, où r est un entier positif correspondant au rayon du noyau. Le coefficient $(i,j) va être définie à partir de la fonction gaussienne.

En 2 dimension la fonction gaussienne est définie par :

$$\mathrm{G}(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

où σ est un réel correspondant à l'écart-type de la fonction gaussienne. Cette fonction est centrée en 0 et est définie sur un espace continu. Pour l'adapter dans le cadre discret, on commence par calculer des coefficients $g_{i,j} = e^{-\frac{(i-r)^2+(j-r)^2}{2\sigma^2}}$ puis de définir $h_{i,j} = \frac{g_{i,j}}{\sum_{0 \leq i,j \leq 2r} g_{i,j}}$.

Implémenter une fonction `noyau_gaussien` qui prend en paramètre un rayon et un écart-type et renvoie le noyau gaussien discret associé.

Le flou gaussien est obtenu en calculant pour chaque pixel (i, j) la moyenne du niveau de gris des voisins du pixel dans un carré de coté $2r + 1$ centré sur le pixel, pondérée par les valeurs du noyau gaussien. Soit $\mathrm{A} \in \mathrm{M}_{n,m}(\mathbb{N})$ une image en niveau de gris. Le résultat du flou gaussien, $\mathrm{B} = (b_{i,j})_{0 \leq i \leq n, 0 \leq j \leq m}$ est définie par :

$$b_{i,j} = \lfloor \sum_{k=i-r}^{i+r} \sum_{l=j-r}^{j+r} m a_{i+k,j+l} * h_{k,l} \rfloor$$

où on décide par convention que $a_{i,j} = 0$ pour tout i, j tel que i ou j n'est pas dans $[\![0, n]\!]$.

Implémenter une fonction `flou_gaussien` qui prend en paramètre un rayon, un écart-type et une image en niveau de gris et qui renvoie une copie de l'image sur laquelle a été appliqué un flou gaussien suivant le rayon et l'écart-type donné en paramètre.

V Algorithmique des Graphes

Exercice 21. *Connexité*

Implémenter une fonction `connexite` qui prend en argument un graphe représenté par sa liste d'adjacence et qui vérifie si le graphe est connexe.

Exercice 22. *Détection de cycles*

Implémenter une fonction `cycles` qui prend en argument un graphe représenté par sa matrice d'adjacence et qui vérifie si le graphe possède un cycle.

Exercice 23. *2-coloration*

On dit qu'un graphe est 2-coloriable si on peut assigner à chaque nœud du graphe une couleur parmi deux, rouge et noir par exemple, de telle manière que deux voisins n'ont jamais la même couleur.

Implémenter une fonction `est_2_coloriable` qui vérifie si un graphe est 2-coloriable en temps linéaire.

Exercice 24. *2-coloration*

On dit qu'un graphe est 3-coloriable si on peut assigner à chaque nœud du graphe une couleur parmi trois, rouge, noir et vert par exemple, de telle manière que deux voisins n'ont jamais la même couleur.

Implémenter une fonction `est_3_coloriable` qui vérifie si un graphe est 3-coloriable. Il n'est pas possible de faire moins qu'une complexité temporelle exponentielle.