Échauffement

1 Algorithmique sur une structure séquentielle.

Exercice 1. ♥ *Recherche linéaire*

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

Exercice 2. *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 3. *Aplatissement*

Implémenter une fonction `aplatissement` qui prend en argument une liste de listes d'entiers, $L = [L_0, L_2, \ldots, L_n]$ et renvoie la concaténation de l'ensemble des listes dans l'ordre, $L_0 \cdot L_2 \cdot \ldots L_n$. Donner la complexité temporelle de votre algorithme en fonction de la taille de l'entrée.

Exercice 4. ★ *Aplatissement de dictionnaires*

Implémenter une fonction `aplatissement_dict` avec les spécifications suivantes :

Algorithme `aplatissement_dict`

ENTRÉES: $L = [D_0, D_2, \ldots, D_n]$ une liste de dictionnaires dont les clés sont des chaînes de caractère et les valeurs des listes d'entiers

SORTIES: D un dictionnaire tel que :

— $\texttt{clé}(D) = \bigcup_{0 \leq i \leq n} \texttt{clé}(D_i)$

— $\forall c \in \texttt{clé}(D), D[c] = \bullet_{0 \leq i \leq n} D_i[c]$

En sortie on attend donc 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.
e.g. si l'entrée est :

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

la sortie sera :

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

Exercice 5. ♥ *Recherche Dichotomique*

Implémenter une fonction `recherche_dichotomique` renvoyant l'indice de la dernière occurrence d'un mot dans une liste de mots triés par ordre alphabétique, en temps logarithmique.
Prouver la correction de votre programme et justifiez sa complexité.

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 et renvoi 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**.
Prouver la correction de votre programme et justifiez sa complexité.
Proposer un jeu de tests pour vérifier la correction de votre programme.

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 dernier 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 et renvoi 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**.
Prouver la correction de votre programme et justifiez sa complexité.
Proposer un jeu de tests pour vérifier la correction de votre programme.

Exercice 8. ♥ *Ordonnancement*

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 élément 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*

Implémenter une fonction `ordonnancement_optimal` qui prend en paramètre une liste d'entiers L et réparti chaque élémentde 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.

2 Tris

Exercice 10. ♥ *Tri par comptage*

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

Exercice 11. ♥ *Tri partition-fusion*

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

Exercice 12. ♥ *Tri rapide*

Implémenter la fonction `tri_rapide` qui prend en entrée un liste d'entier et renvoie une liste triée selon l'algorithme du tri rapide, en utilisant un pivot aléatoire.
Pour la génération aléatoire on utilisera la fonction `randint` *du module* `random`.
`randint(start, stop)` *génère un entier entre* `start` *et* `stop` *inclus.*

3 Récursivité

Les algorithmes de cette partie devront être implémentés autant que possible par une méthode récursive

Exercice 13. *Fonction d'Ackermann*

Définir une fonction `ackermann` qui implémente la fonction d'Ackermann définie par :

$$A(m,n) = \begin{cases} n+1 & \text{si } m = 0 \\ A(m-1,1) & \text{si } m > 0 \text{ et } n = 0 \\ A(m-1, A(m,n-1)) & \text{si } m > 0 \text{ et } n > 0. \end{cases}$$

Exercice 14. ♥ *Exponentiation rapide*

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

Exercice 15. ♥ *Conversion binaire*

Implémenter une fonction `binaire` qui prend en entrée un entier relatif et le converti en binaire (complément à deux) sous la forme d'une chaîne de caractères représentant l'écriture binaire du nombre.

Exercice 16. ♥ *Fibonacci*

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

Exercice 17. ♥ *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 18. ★ *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, en dessous, au plus léger, au dessus. La 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'instructions 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 $n \rightarrow m$ où n est la zone de départ du déplacement et m la zone d'arrivée.
e.g. :

$$A \rightarrow B$$
$$A \rightarrow C$$
$$B \rightarrow C$$

4 Traitement d'images

Pour essayer vos fonctions sur de réelles images vous pouvez utiliser la librairie `pillow` *de python.*

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

# Ouvrir un image
img = Image.open("chemin/vers/image.jpg")
# Convertir l'image en Noir et Blanc (matrice de booléens)
img.convert("1")
# Convertir l'image en niveau de gris (matrice d'entiers entre 0 et 255)
img.convert("L")

# Obtenir une matrice à partir de l'image
m = np.array(img)

# Obtenir une image à partir de la matrice
img2 = Image.fromarray(m)

# Afficher l'image
img2.show()
```

Exercice 19. *Identification des composantes connexes*

On considère des images en noir et blanc représentés par une matrice de booléens. Chaque groupement de pixels noirs forme une composante connexe que l'on cherche à identifier.
Implémenter une fonction `etiquettage_composante` qui prend une image en noir et blanc et renvoi une copie où chaque composante connexe est étiquetée. L'étiquetage consiste en l'attribution d'un numéro de 1 à n, le nombre de composantes, à chaque composante. Dans la matrice finale, la valeur de chaque pixel d'une composante dans l'image sera celle de l'étiquette.

Exercice 20. ♥ *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.
Implémenter une fonction `filtrage` qui prend une image et un filtre et renvoie une copie de l'image sur laquelle a été appliquer le filtre.

Exercice 21. ★ *Flou Gaussien*

On considère des images en niveau de gris représentés par une matrice de flottants entre 0 et 1.
Appliquer un flou gaussien sur une image en niveau de gris revient à appliquer un filtre passe-bas en transformant 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, $\mathcal{H} = (h_{i,j})_{0 \leq i,j \leq 2r}$, de taille $2r+1$, où r est un entier positif correspondant au rayon du noyau. Le coefficient (i, j) du noyau gaussien est calculé à l'aide de la fonction gaussienne suivante :

$$G(i, j) = \frac{1}{2\pi\sigma^2} e^{\frac{i^2+j^2}{2\sigma^2}}$$

où σ est un réel correspondant à l'écart-type de la fonction gaussienne.
—
Implémenter une fonction `noyau_gaussien` qui prend en paramètre un rayon et un écart-type et renvoie le noyau gaussien 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 côté $2r + 1$ centré sur le pixel, pondérée par les valeurs du noyau gaussien. Soit $\mathcal{A} \in M_{n,m}([0, 1])$ une image en niveau de gris. Le résultat du flou gaussien, $B = (b_{i,j})_{0 \leq i \leq n, 0 \leq j \leq m}$ est définie par :

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

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

5 Algorithmique des Graphes

Exercice 22. ♥ *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 23. ♥ *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 24. *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 25. ★ *3-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.