01 - Piles, files et récursivité : Exercices en plus

Exercice 1 : On rappelle que si a et b sont deux entiers naturels, non tous les deux nuls, tels que a ≥ b, $\text{PGCD}(a,b) = \text{PGCD}(a-b,b)$

1. (Sur une feuille pour comprendre le procédé) En utilisant plusieurs fois de suite cette propriété, calculer $\text{PGCD}(36,24)$, puis $\text{PGCD}(16,9)$. Vous n'avez pas le droit de faire des divisions ou des multiplications. Vous réfléchirez à la condition qui vous a permis, à la fin de votre processus, de conclure.
2. Ecrire une fonction non récursive d'arguments a et b qui calcule le PGCD de a et b.
3. Ecrire une fonction récursive d'arguments a et b qui calcule le PGCD de a et b .

Exercice 2 : a) Présentation du problème

Etant donnée une chaine de caractères, contenant des parenthèses ouvrantes et fermantes, on veut savoir si l'expression est bien parenthésée. Voici une définition théorique d'une expression bien parenthésée :

Une expression bien parenthésée est soit sans parenthèses, soit la concaténation de deux expressions bien parenthésées, soit une expression bien parenthésée mise entre parenthèses.

Exemples :

Bien parenthésée (e)	Mal parenthésée (e
mon expression	mon expression)
mon expression (bon exemple)	)mauvais exemple
$(2+3)*(4-(5-x))$	$(2+3)^*(4-(5-\text{x})$
je n'ai plus d'exemples(...)	(je sors))(

Remarquons que pour qu'une expression soit bien parenthésée, il faut que le nombre de parenthèses ouvrantes soit égal au nombre de parenthèses fermantes, mais ce n'est pas suffisant.

Programmation en Python

Ecrire en Python une fonction **`parenthesage`** ayant pour argument une expression, de type chaine de caractères, qui retourne un booléen nous indiquant si l'expression est bien parenthésée ou non.

On utilisera une pile et les fonctions primitives associées. Voici l'idée du fonctionnement de cette pile : on crée une pile vide en variable locale, on parcourt l'expression et à chaque fois que l'on rencontre une parenthèse ouvrante, on la stocke dans la pile. Si on rencontre une parenthèse fermante, on dépile une parenthèse ouvrante (et s'il n'y a rien à dépiler ? ?).

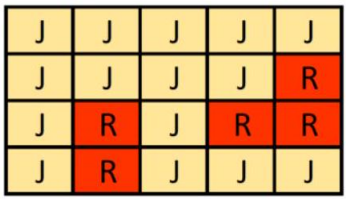
Je vous laisse le soin de réfléchir à ce qu'il faut obtenir une fois que notre expression est entièrement parcourue.

Exercice 3 (*Traitement d'image : recherche des composantes connexes*) : Cet exercice est une version simplifiée d'une vraie étude commandée par la société LIMAGRAIN à l'école d'ingénieurs clermontoise ISIMA.

Le problème est le suivant : dans une photographie d'un tas de grain de maïs, on cherche à cerner les groupes de grains défectueux, qui sont rouges au lieu d'être jaunes.

On suppose que la photographie est modélisée par une liste de liste de caractères qui peuvent être ' R ' ou ' J '.

Par exemple la liste M=[['J','J','J','J','J'],['J','J','J','J','R'],['J','R','J','R','R'],['J','R','J','J','J']] correspond à l'image :



Sur cette image, on souhaite détecter les positions des 2 groupes de grains rouges. (on parle de 2 composantes connexes)

La fonction devra donc, à partir de la liste M ci-dessus, renvoyer la liste [[[2,1],[3,1]], [[1,4],[2,3],[2,4]]] (l'ordre des éléments dans la liste, et également dans les sous-listes, n'a pas d'importance)

Voici le principe pour construire votre fonction :

- On crée une liste `visites` de même taille que M , dont tous les éléments sont initialisés à `False`. Au fur et à mesure de l'examen de la couleur de chaque grain, on passera l'élément de visites correspondant à `True`
- On balaye les éléments de la liste M
  - Si cette position n'a pas été visitée, on indique qu'elle vient de l'être, et si cette position est un grain rouge :
    - On crée une pile vide (elle va nous servir à stocker les positions des grains de la composante connexe à déterminer, pour étudier à nouveau leurs voisins) et on y met la position de ce grain rouge.
    - On construit une liste correspondant à une composante connexe, initialisée avec la position de ce grain rouge.
    - Tant que la pile n'est pas vide :
      - on dépile la dernière position
      - on récupère la liste des positions des voisins (on suppose que l'on a une fonction `voisins` qui fait cela)
      - pour chaque voisin non encore visité, on passe leur statut à `True`, et s'ils sont rouges on les ajoute à la composante connexe et on les met dans la pile.
    - Notre composante connexe est complète, on l'ajoute à la liste finale que l'on renverra
- On renvoie la liste des composantes connexes.