Exercices sur la récursivité

NSI Terminale

On pourra traiter, dans l'ordre :

- des exercices précédés d'un ◊
- d'autres exercices, par exemple *Permutations de caractères* ou *Conversion des nombres romains*

Correction possible pour les exercices

Généralités sur les algorithmes récursifs

◊ Que calculent les fonctions `even` et `odd` ?

Voici la déclaration en Python de deux fonctions :

```
def even(n):
    if n == 0:
       return True
    else:
       return not odd(n)

def odd(n):
    if n == 0:
       return False
    else:
       return not even(n)
```

Que pensez-vous des expressions calculées par chacune des deux fonctions dans les deux cas de base et récursif ?

◊ Encore la fonction `even`

Voici la déclaration en Python de la fonction `even` :

```
def even(n):
    if n == 0:
       return True
```

```
    else:
        return even(n - 2)
```

Que pensez-vous de cette fonction ?

Algorithmes récursifs sur les nombres

◊ Somme de deux entiers

Proposez un algorithme récursif de calcul de la somme de deux entiers naturels a et b en supposant que les seules opérations de base dont vous disposez sont

- l'ajout de 1 à un entier a : *a+1*
- le retrait de 1 à un entier a : *a-1*
- et les comparaisons à 0 d'un entier a : *a=0*, *a>0* et *a<0*.

Puis programmez cet algorithme en Python pour en faire une fonction `add` à deux paramètres.

◊ Produit de deux entiers

Proposez un algorithme récursif de calcul du produit de deux entiers naturels a et b en supposant que les seules opérations de base dont vous disposez sont

- la somme de deux entiers a et b : *a+b*
- le retrait de 1 à un entier a : *a-1*
- et la comparaison à 0 d'un entier a : *a=0*.

Puis programmez cet algorithme en Python pour en faire une fonction `mult` à deux paramètres.

Puissance entière d'un nombre réel

Proposez un algorithme récursif de calcul de la puissance n-ième (n entier > 0) d'un nombre réel a en supposant que les seules opérations de base dont vous disposez sont

- le produit de deux réels a et b : $a \times b$
- le retrait de 1 à un entier a : *a-1*
- et la comparaison à 0 d'un entier a : *a=0*.

Puis programmez cet algorithme en Python pour en faire une fonction `power` à deux paramètres.

Algorithme d'Euclide

L'algorithme d'Euclide permet de calculer le pgcd de deux nombres entiers, c'est-à-dire le plus grand entier positif divisant ces deux nombres, par des divisions successives.

Voici le déroulement de cet algorithme pour le calcul du pgcd de *a=119* et *b=544*

a		b		q		r
119	=	544	x	0	+	119
544	=	119	x	4	+	68
119	=	68	x	1	+	51
68	=	51	x	1	+	**17**
51	=	**17**	x	3	+	0

Le pgcd de 119 et 544 est le dernier reste non nul, c'est-à-dire 17.

Le pgcd n'est pas défini lorsque les deux nombres sont nuls.

Exprimez de manière récursive cet algorithme. Vous pourrez supposer que les deux entiers a et b sont positifs ou nuls, et que l'un au moins de ces deux entiers n'est pas nul.

Codez cet algorithme en Python en utilisant l'opérateur modulo.

Algorithme récursif géométrique

◊ Coloriage

Supposons données les coordonnées (entières) *(x ; y)* d'un pixel situé à l'intérieur d'une région du plan délimitée par une courbe fermée. Les points sur la courbe délimitant la région sont de couleur noire, et ceux à l'intérieur sont blancs. On souhaite donner la couleur rouge à tous les points à l'intérieur de la région.

Proposez un algorithme récursif pour effectuer ce coloriage. (Vous pourrez utiliser deux fonctions `get_color(x, y)` qui renvoie la couleur du pixel de coordonnées *(x ; y)* et `set_color(x, y, color)` qui fixe la couleur du pixel de coordonnées *(x ; y)*.

Pour simplifier on peut supposer ici que les couleurs sont définies par des valeurs globales `BLACK`, `WHITE` et `RED`.

Algorithmes récursifs sur les chaînes de caractères

◊ Palindrome

Un *palindrome* est un mot dont les lettres lues de gauche à droite sont les mêmes que celles lues de droite à gauche. Les mots `radar`, `elle`, `été`, `ici` sont des palindromes.

Réalisez un prédicat qui teste si un mot est un palindrome.

Permutations des caractères

Dans cet exercice, on appelle *permutation* d'une chaîne de caractères s toute chaîne de même longueur que s contenant les mêmes caractères que s. Par exemple, la chaîne `'eadbc'` est une permutation de la chaîne `'abcde'`.

Réalisez une fonction récursive qui construit la liste de toutes les permutations possibles d'une chaîne s.

NB il sera probablement nécessaire de définir des fonctions auxiliaires, on essaiera de les coder récursivement aussi.

Conversion des nombres romains

On suppose défini le dictionnaire :

```
VALEUR_ROMAIN = { 'M' : 1000, 'D' : 500, 'C' : 100, 'L' : 50, 'X' : 10, 'V' : 5, 'I' : 1}
```

Réalisez une fonction récursive `romain_to_arabe` qui prend en paramètre une chaîne de caractères représentant un « nombre romain » et dont le résultat est l'entier correspondant.

```
>>> romain_to_arabe('X')
10
>>> romain_to_arabe('XCI')
91
>>> romain_to_arabe('MMXIX')
2019
```

NB Il est nécessaire de prendre en compte le cas où la valeur correspondante au second caractère est supérieure à celle du premier.

Algorithmes sur les listes récursives

On considère ici les listes définies récursivement comme présenté lors de la séance de cours.

On ne s'autorise donc que l'existence de la liste vide (`[]`) et des opérations permettant

- de construire un couple *(x, R)* : `[x]+R`, où `R` est une liste ;
- d'accéder à la tête d'une liste `l` non vide : `l[0]` ;
- d'accéder au reste d'une liste `l` non vide : `l[1:]`.

Somme des éléments d'une liste

Donnez une version récursive du calcul de la somme des éléments d'une liste de nombres.

Dernier élément

Réalisez la fonction `last` qui renvoie le dernier élément d'une liste non vide et déclenche une exception si la liste est vide.

Concaténer deux listes

Concaténer deux listes `l1` et `l2`, c'est construire une liste contenant les éléments de `l1` suivis de ceux de `l2`.

Réalisez la fonction `concat` qui renvoie la concaténation des deux listes passées en paramètre.

Appliquer une fonction à tous les éléments d'une liste

Appliquer une fonction f à tous les éléments d'une liste l, c'est construire une liste contenant les images *f(x)* de tous les éléments x de l.

Par exemple, appliquer la fonction `carre` à la liste *[1,2,3]* donne la liste *[1,4,9]*.

Réalisez une fonction nommée `map` qui applique une fonction aux éléments d'une liste.

Mélange

Réalisez récursivement une fonction nommée `shuffle` paramétrée par deux listes `l1` et `l2` qui renvoie une liste dont les éléments sont ceux de ces deux listes dans un ordre alterné, tant que c'est possible.

Si l'une des listes est plus courte que l'autre, on termine avec les éléments non utilisés de la plus longue liste.

```
>>> shuffle([1,5,3,9,7],[8,2,6])
[1,8,5,2,3,6,9,7]
```

Autres exercices

NB Dans les exercices suivants, vous pouvez à nouveau utiliser les listes natives de Python.

Les tours de Hanoï

Il s'agit d'un exemple très classique d'algorithme récursif.

Voici ce qu'en dit Wikipedia

> Les tours de Hanoï sont un jeu de réflexion imaginé par le mathématicien français Édouard Lucas, et consistant à déplacer des disques de diamètres différents d'une tour de « départ » à une tour d'« arrivée » en passant par une tour « intermédiaire », et ceci en un minimum de coups, tout en respectant les règles suivantes :
>
> - on ne peut déplacer plus d'un disque à la fois ;
> - on ne peut placer un disque que sur un autre disque plus grand que lui ou sur un emplacement vide.

Trouver (puis programmer) un algorithme pour résoudre ce problème n disques. On pourra se contenter d'afficher sur la sortie standard les déplacements réalisés au cours de la résolution (par exemple *"dique de taille 3 déplacer de la tour 1 à la tour 2"*).

Tri par insertion

Réalisez une version récursive du *tri par insertion* vu en première ### Permutations

Cet exercice est similaire à celui du calcul des permutations sur les chaînes de caractères.
On cherche ici à produire la liste de toutes les listes obtenues par permutation des éléments d'une liste donnée.

On peut alors reproduire l'exemple donné sur Wikipedia pour produire toutes les permutations de la liste `["Belle Marquise","vos beaux yeux","me font mourir","d'amour"]` dont le texte est tiré du Bourgeois gentilhomme (Acte II Scène IV) de Molière.

Organisation des rencontres d'un championnat

Dans le cadre d'un championnat sportif (ou autre) on dispose de la liste de tous les joueurs (ou équipes) concernés. On souhaite organiser la liste de toutes les rencontres possibles entre ces joueurs. Chaque joueur devant rencontrer tous les autres une et une seule fois.

On considère que chaque joueur est identifié par un nombre (qui peut par exemple correspondre à une clef dans une table qui permet d'accéder aux informations détaillées sur le joueur). Une liste de joueurs est donc en fait la liste des nombres associés à ces joueurs.

Une rencontre est représentée par un couple (tuple) dont les deux composantes sont les numéros des deux joueurs impliqués.

Donnez et codez un algorithme récursif qui produit, à partir d'une liste de joueurs, la liste de toutes les parties possibles entre ces joueurs.

```
>>> rencontres([1,2,3,4])
[ (1,2) , (1,3) , (1,4), (2,3), (2,4), (3,4) ]
```

Mettre l'*ours* en *cage*

On dispose d'une variable `DICO` qui est une liste de mots de quatre lettres. (cf dico.py).

On dispose de deux mots *m1* et *m2* de `DICO` et on cherche à construire une liste de mots telle que :

- la liste commence par *m1* et termine par *m2*
- deux mots consécutifs de la liste ne diffèrent que d'une lettre, sans tenir compte de l'ordre des lettres dans chacun des mots (on dira qu'ils sont *voisins*).

On veut donc écrire une fonction `solve`, paramétrée par deux chaînes de caractères, dont le résultat est cette liste de mots quand elle existe et `None` dans le cas contraire.

```
>>> solve('ours', 'cage')
['ours', 'duos', 'ducs', 'dues', 'dure', 'bure', 'brie', 'baie', 'aime', 'came', 'cage']
>>> solve('ours', 'orme')
None
```

On ne cherche pas nécessairement à produire le chemin le plus court.

NB Il faut, à partir d'un mot, calculer ses mots voisins, puis les essayer un à un jusqu'à en trouver un, s'il existe, qui permette d'atteindre le mot cible. On construit ainsi progressivement un chemin du mot initial jusqu'au mot final. Lors de la construction du chemin on peut à un moment aboutir à une impasse, c'est-à-dire atteindre un mot dont tous les voisins sont déjà dans le chemin. Dans ce cas il faut "revenir en arrière" pour essayer des voisins qui ont pu être laissés de côté.
Lors de la recherche, il faut bien sûr éviter de tourner en rond en revenant sur un mot déjà utilisé dans le chemin partiellement construit.

Il est donc certainement pertinent de disposer de l'information

- sur les mots déjà utilisés dans le chemin en construction ;
- sur les mots qu'il reste à examiner, ceux sont des voisins du dernier mot du chemin en construction.

Remarque Ce problème entre dans la classe plus large des problèmes qui consistent à chercher un chemin dans un graphe. Ici les nœuds du graphe sont les mots et il existe une arête entre deux mots s'ils sont voisins. Ce sujet sera abordé en algorithmique (dans le chapitre sur les graphes).

Flocons de Von Koch

Utilisez le module `turtle` pour tracer des flocons de Von Koch.