Récursivité

Les exercices marqués dune étoile (*) sont plus difficile et sont à faire dans un deuxième temps. Pour l'écriture de la documentation des fonctions, on pourra supposer que les listes sont toutes constituées d'entiers.

1 Opérations arithmétiques

1.1 Plus grand diviseur commun

Écrire une fonction récursive qui retourne le PGCD de deux entiers passés en argument. On se servira de l'algorithme d'Euclide :

pgcd$(n, m) =$

- m si $n = 0$
- pgcd(m, n) si $m < n$
- pgcd$(n, m \bmod n)$ sinon

1.2 Multiplication

Écrire une fonction récursive qui fait le produit d'un entier a (positif) par un entier b (positif) en utilisant uniquement l'opération somme.

1.3 Puissance $n$ième d'un nombre

Écrire une fonction récursive qui élève un nombre à la puissance $n$ième (positive) et qui n'utilise que l'opération multiplication.

1.4 Division euclidienne

Écrire deux fonctions récursives `reste` et `quotient` qui renvoient le reste et le quotient de la division euclidienne d'un entier positif a par un entier positif b et qui n'utilisent que la soustraction et l'addition.

1.5 Exponentiation rapide *

La méthode de calcul de a^n vue précédemment nécessite n multiplications :

$$a^n = a \times a \times \ldots \times a$$

Mais on peut faire mieux en remarquant que dans le cas où n est une puissance de 2, on peut écrire :

$$a^{2^p} = (((a)^2)^{2\cdots})^2$$

Dans ce cas on fait p élévations au carré, soit $\lg(n)$ multiplications. Écrire une fonction récursive `exp(a, n)` qui retourne a^n dans la cas où `n` est une puissance de 2, puis modifier cette fonction pour qu'elle traite le cas général où `n` est quelconque.

2 Fonctions récursives sur les listes

2.1 Fonctions `len` et `replicate`

Écrire une fonction récursive qui retourne le nombre d'éléments d'une liste, puis une fonction qui renvoie une liste composée de `n` fois le même élément `el`.

2.2 Fonctions `sum` et `product`

Écrire une fonction récursive qui retourne la somme des éléments d'une liste, puis une autre fonction qui retourne leur produit.

2.3 Fonction `reverse`

Écrire une fonction récursive qui retourne une liste dont les éléments sont placés dans l'ordre inverse de la liste passée en argument.

2.4 Fonction `zip`

Écrire une fonction récursive `zip` qui prend en argument deux séquences et qui retourne la liste des tuples de leurs éléments pris deux à deux, même si les deux séquences ne sont pas de même taille (par exemple `zip([1, 2, 3], [4, 5, 6, 7])` doit retourner `[(1, 4), (2, 5), (3, 6)]`).

2.5 Fonction `map`

Écrire une fonction récursive `map(f, li)` qui prend en argument une fonction `f` et une liste `li` et retourne la liste des images par `f` des élements de `li`.

2.6 Fonction `filter`

Écrire une fonction récursive `filter(cond, li)` qui retourne la liste des éléments de `li` qui vérifient la condition `cond`.

Remarque. `cond` est une fonction booléenne (qui renvoie un booléen), par exemple :

```
def estPositif(x):
    return x >= 0
```

2.7 Fonctions `max` et `isIn`

Écrire une fonction récursive qui retourne la valeur maximum d'une liste, puis une fonction qui renvoie `True` si un élément `el` est présent dans une liste et `False` sinon.

2.8 Fusion de deux listes triées

Écrire une fonction `merge(li1, li2)` qui prend en argument deux listes triées `li1` et `li2` et qui retourne la liste triée constituée de tous les éléments de `li1` et `li2`.

2.9 Méthode de Horner

L'évaluation d'un polynôme $P(x) = a_0 + a_1\,x + \ldots + a_{n-1}\,x^{n-1}$ en un point x par un calcul direct :

```
def P(a:list, x:float)->float:
    S = 0
    for i in range(len(a)):
        S += a[i] * (x**i)
    return S
```

est une solution coûteuse puisqu'elle consiste à calculer à chaque fois la puissance de x. La méthode de Horner consiste à factoriser ce calcul :

$$P(x) = a_0 + x(a_1 + x(+\ldots + x(a_{n-2} + x\,(a_{n-1})\ldots))$$

De telle sorte qu'à chaque étape du calcul on effectue seulement une addition et une multiplication.

Écrire une fonction récursive `P(a, x)` d'évaluation d'un polynôme `a` en un point `x` par cette méthode.

2.10 Fonctions `all` et `any`

Écrire une fonction récursive `all(cond, li)` qui retourne `True` si tous les éléments de `li` vérifient la condition `cond` et `False` sinon. Écrire de même une fonction `any` qui retourne `True` si au moins un élément de `li` vérifie `cond` et `False` sinon.

2.11 Recherche séquentielle

Écrire une fonction récursive `seq_search(el, li)` de recherche séquentielle de l'élément `el` dans la liste `li`. Si `el` n'appartient pas à la liste, la fonction devra retourner `None`.

2.12 Recherche dichotomique *

Écrire une fonction récursive `dich_search(li, el)` qui retourne l'indice de l'élément `el` dans la liste triée `li` si `el` est présent dans `li` et `None` sinon (on suppose que la liste `li` ne contient pas de doublons).

2.13 Fonction `imin` *

Écrire une fonction récursive qui retourne l'indice du minimum d'une lsite de nombres `li`.

3 Suite de Fibonacci

3.1 Algorithme récursif

Écrire une fonction récursive `F(n)` qui calcule le $n$ième terme de la suite de Fibonacci :

$$\begin{aligned} u_0 &= 0 \\ u_1 &= 1 \\ u_n &= u_{n-1} + u_{n-2} \end{aligned}$$

3.2 Algorithme itératif

Écrire une fonction **itérative** `F2(n)` qui calcule le $n$ième terme de la suite de Fibonacci.

3.3 Complexité de l'algorithme récursif

Donner la complexité de l'algorithme itératif. Pour avoir une idée de la complexité de l'algorithme récursif, dessiner quelques niveaux de l'arbre des appels récursifs. A votre avis, cet algorithme a-t-il une complexité polynomiale ?

4 Numération binaire

4.1 Décomposition binaire

Écrire une fonction récursive `bin(n)` qui retourne la liste des bits de la décomposition binaire de l'entier `n`.

4.2 Affichage des nombres sous formes de chaînes *

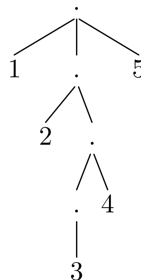
Écrire une fonction récursive `ch_bin(n)` qui **affiche** tous les nombres binaires (positifs) codés sur n bits dans l'ordre. Par exemple pour $n=3$ on doit obtenir :

```
000
001
010
011
100
101
110
111
```

5 Parcours d'arbres *

5.1 Mise à plat d'un arbre quelconque

On peut modéliser un arbre par une imbrication de listes. Ainsi, l'arbre ci-dessous :



Peut être modélisé par la liste de listes suivantes :

```
li = [1, [2, [[3], 4]], 5]
```

A chaque ouverture de crochet correspond un noeud (sans valeur), les feuilles stockent des entiers. Écrire une fonction récursive `flatten` qui retourne la liste des feuilles de l'arbre. Appelée sur l'exemple précédent, elle doit retourner la liste `[1, 2, 3, 4, 5]`.

5.2 Copie d'une liste de listes

Écrire une fonction récursive `deepcopy` qui créé une copie d'une liste de listes de profondeur quelconque (un arbre, donc). Sur l'exemple précédent, la fonction doit donc retourner `[1, [2, [[3], 4]], 5]`. Mais attention ! La liste retournée ne doit pas changer si on modifie un élement de la liste de départ. Par exemple, si on remplace la feuille de valeur `3` par la valeur `0` par l'instruction :

```
li[1][1][0][0] = 0
```

La liste de départ doit être `[1, [2, [[0], 4]], 5]` mais sa copie doit rester `[1, [2, [[3], 4]], 5]`.

6 Dénombrement *

6.1 Permutations

On veut écrire une fonction récursive `perms(li)` qui retourne la liste de toutes les permutations des éléments d'une liste `li`.

Exemple. `perm([1, 2, 3])` doit retourner :

```
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

Écrire tout d'abord une fonction récursive `ins(el, li)` qui retourne la liste de toutes les insertions possibles de l'élément `el` dans la liste `li`.

Exemple. `ins(4, [1, 2, 3])` doit retourner :

```
[[4, 1, 2, 3], [1, 4, 2, 3], [1, 2, 4, 3], [1, 2, 3, 4]]
```

En déduire la fonction `perms`.

6.2 Sous-listes

Écrire une fonction récursive `subs(li)` qui retourne la liste de toutes les sous-listes de la liste `li`.

Exemple. `subs([1, 2, 3])` doit retourner :

```
[[1, 2, 3], [1, 2], [1, 3], [1], [2, 3], [2], [3], []]
```

6.3 Énumération de toutes les possibilités

Déduire des deux questions précédentes une fonction `choices(li)` qui retourne la liste de toutes les sous-listes qu'on peut former avec les éléments de la liste `li`.

Exemple. `choices([1, 2, 3])` doit retourner :

```
[[1, 2, 3],
 [2, 1, 3],
 [2, 3, 1],
 [1, 3, 2],
 [3, 1, 2],
 [3, 2, 1],
 [1, 2],
 [2, 1],
 [1, 3],
 [3, 1],
 [1],
 [2, 3],
 [3, 2],
 [2],
 [3],
 []]
```