Feuille 1: Introduction à la récursivité sur une représentation récursive des polynômes
CPP 2A, Semestre 2 – 2018-2019

Dans cette séance, on travaille sur une représentation récursive des polynômes à une variable. L'objectif *n'est pas ici de programmer efficacement* sur les polynômes (il y a d'autres représentations plus efficaces). Le but est (re)trouver des algorithmes simples de calculs sur les polynômes à l'aide de la récursivité. Dans la feuille 2, on verra que ce type de représentation récursive est commode pour d'autres objets que les polynômes.

Définition 1 *Un* polynôme de Horner sur la variable X *est:*

- *soit le uplet vide* `()` *qui représente le polynôme nul ;*
- *soit un couple* `(a, q)` *où* `a` *est un coefficient (entier ou flottant) et où* `q` *est lui-même un* polynôme de Horner sur la variable X*: un tel couple représente le polynôme "*$\texttt{a} + X.\texttt{q}$*".*

Par exemple, `(0, (2, (0, (7, ())))))` représente le polynôme $0 + X.(2 + X.(0 + X.(7 + X.0)))$, c'est-à-dire $2X + 7X^3$.

Les traitements sur ces polynômes sont programmés par récursivité sur leur structure. Par exemple, la fonction `evalpoly` ci-contre évalue le polynôme `p` sur le nombre `x` (elle réalise la fonction polynomiale associée à `p`).

```
def evalpoly(p, x):
    if p==():
        return 0
    (a, q) = p
    return a+x*evalpoly(q, x)
```

Exercice 1 Écrire une fonction Python pour chacun des calculs ci-dessous (en estimant à chaque fois le nombre d'opérations qu'elle effectue sur les coefficients)

1. multiplication d'un polynôme par un coefficient ;
2. multiplication d'un polynôme par un monôme X^n (avec n entier naturel) ;
3. addition de deux polynômes ;
4. multiplication de deux polynômes ;
5. dérivé d'un polynôme : dans un premier temps, on pourra se limiter à une méthode naïve en $\Theta(n^2)$ additions de coefficients ; on cherchera ensuite une méthode linéaire en introduisant une fonction récursive intermédiaire qui étant donné un polynôme P et un entier naturel n calcule le polynôme $\frac{1}{X^{n-1}}.\frac{\mathrm{d}(X^n.P)}{\mathrm{d}X}$;
6. calcul du degré d'un polynôme non nul : on renverra `None` comme degré du polynôme nul (au lieu de la convention mathématique $-\infty$) ;

7. calcul du normalisé d'un polynôme : c'est un polynôme qui représente la même fonction polynomiale mais sans coefficients nuls inutiles ;

8. calcul de la puissance P^n d'un polynôme P pour n entier naturel non nul en utilisant $\mathcal{O}(\log(n))$ multiplications intermédiaires de polynômes (dans un premier temps, on pourra utiliser $n-1$ multiplications).

9. division suivant les puissances croissantes : étant donnés un entier naturel n et deux polynômes P_1 et P_2 tels que le terme constant de P_2 est non nul, cette fonction doit retourner l'unique couple (Q, R) qui satisfait

$$P_1 = Q.P_2 + X^n.R \quad \text{et} \quad \deg(Q) < n$$

Indication : commencer par considérer les cas $n = 0$ et $n = 1$ avant de généraliser.

En Python, il est en fait aussi simple de représenter un polynôme directement par une liste, où chaque élément d'indice i dans la liste représente le coefficient associé au monôme X^i. Par exemple, $2X + 7X^3$ est représenté par $[0, 2, 0, 7]$.

Exercice 2 Considérons la fonction récursive `evalpoly` sur les polynômes de Horner.

1. Étant donné un polynôme $a_0+a_1.X+a_2X^2+\ldots+a_{n-1}.X^{n-1}$, indiquer les calculs de successifs faits par `evalpoly` pour un `x` donné. Quel est le nombre d'additions et de multiplications en fonction de n ?

2. Écrire une telle fonction d'évaluation pour la représentation des polynômes par des listes Python : on s'appliquera à reproduire les calculs sur les coefficients faits par la fonction récursive (mêmes calculs, dans le même ordre), mais sans utiliser de récursivité (juste une iteration).

Exercice 3 Implémenter les calculs 1 à 7 de l'exercice 1 en utilisant la représentation des polynômes par des listes, sans utiliser de récursivité (juste des itérations).