TP 10 : Polynômes

Mots clés : *opérations, évaluation, Hörner, exponentiation rapide*

Table des matières

I Représentation de polynôme

On rappelle qu'un polynôme est entièrement déterminé par ses coefficients : ces coefficients sont les composantes du polynôme dans la base canonique $(1, X, X^2, \dots, X^n)$ de $\mathbb{K}_n[X]$. En informatique, on peut donc représenter un polynôme par une liste. Par exemple : [1,3,4] représente le polynôme $1 + 3X + 4X^2$.

❒ Q1. Quel est le polynôme représenté par la liste $[1] * 7$?

❒ Q2. Ecrire une fonction `evaluation(L, x)` qui, pour un polynôme P donné sous forme de liste L et une valeur x, renvoie la valeur de $P(x)$.

```
>>> evaluation([1, 0, 2, 2, -1], 0)
1
>>> evaluation([1, 0, 2, 2, -1], 1)
4
```

❒ Q3. Déterminer le nombre d'opérations de `evaluation(L, x)` en fonction de la taille de `L`.En déduire la complexité de cette fonction.

❒ Q4. Ecrire une fonction `reduit(L)` qui, pour une liste `L`, renvoie la même liste mais en éliminant tous les 0 à droite « en trop ».

```
>>> reduit([1, 0, 2, 2, -1, 0, 0, 0])
[1, 0, 2, 2, -1]
```

❒ Q5. Montrer que la fonction précédente se termine en identifiant un variant de boucle.

❒ Q6. Ecrire une fonction `deg(L)` qui, pour une liste `L`, renvoie le degré du polynôme représenté.

```
>>> deg([1, 0, 2, 2, -1, 0, 0, 0])
4
```

❒ Q7. Ecrire une fonction `ajout_zero(L, n)` qui renvoie la liste `L` complétée par `n` zéros.

```
>>> ajout_zero([1, 0, 2, 2, -1], 3)
[1, 0, 2, 2, -1, 0, 0, 0]
```

❒ Q8. Ecrire une fonction `somme(L1, L2)` qui renvoie une liste représentant la somme des polynômes représentés par `L1` et `L2`. On pourra utiliser la fonction `ajout_zero`.

```
>>> somme([1, 0, 2, 2, -1], [1, 2, 3])
[2, 2, 5, 2, -1]
```

❒ Q9. Ecrire une fonction `produit(L1, L2)` qui renvoie une liste représentant le produit des polynômes représentés par `L1` et `L2`. On pourra utiliser la fonction `ajout_zero` pour obtenir des listes de taille `len(L1)+len(L2)-1`.

On rappelle que si $P = \sum_{k=0}^{n} a_k X^k$ et $Q = \sum_{j=0}^{m} b_j X^j$ alors

$$PQ = \sum_{k=0}^{n+m} \left(\sum_{j=0}^{k} a_j b_{k-j} \right) X^k.$$

```
>>> produit([1, 0, 2, 2, -1], [1, 2, 3])
[1, 2, 5, 6, 9, 4, -3]
```

II Algorithme de Hörner

II.1 Présentation

Soit un polynôme quelconque

$$P(X) = a_n X^n + a_{n-1} X^{n-1} + \cdots + a_1 X + a_0.$$

Le schéma de Hörner propose une façon bien plus rapide (en terme de complexité) pour le calcul de $P(x_0)$ que la fonction `evaluation(L, x)`.

1. On multiplie le coefficient a_n par x_0
2. On ajoute au résultat a_{n-1}
3. On multiplie le résultat par x_0
4. On ajoute au résultat a_{n-2}
5. ... et ainsi de suite jusqu'à a_0

Cela revient à écrire :

$$P(x_0) = ((\cdots((a_n x_0 + a_{n-1})x_0 + a_{n-2})x_0 + \cdots)x_0 + a_1)x_0 + a_0.$$

Calculer $P(2)$ où $P(X) = 4X^3 - 7X^2 + 3X - 5$ via l'algorithme de Hörner "à la main".

II.2 Algorithme

❒ Q10. Ecrire la fonction `horner(P, x)` qui prend en argument une liste représentant un polynôme et `x` et qui renvoie la valeur de `P(x)` via l'algorithme de Hörner.

```
>>> P = [3, -1, 2, -3]
>>> horner(P, 0)
-3    # Pourquoi pouvait-on s'y attendre ?
>>> horner(P, -1)
-9
>>> horner(P, 2)
21
```

❒ Q11. Déterminer la complexité de la fonction `horner(P, x)` en fonction de la taille de `P`.

II.3 Comparaison avec l'algorithme naturel

En reprenant la fonction `evaluation(L,x)` de la partie 1 qui renvoie la valeur de $P(x)$ où P est le polynôme décrit par la liste de ses coefficients (rangés par ordre croissant) L. On compare le temps d'exécution des deux programmes précédents :

Essayez !

```
import time

start_time = time.time()  # enregistre le temps présent dans un variable
evaluation([1] * 1000, 10)    # exécute le programme évaluation
print(time.time() - start_time) # compare le temps présent avec le temps enregistré

start_time = time.time()
horner([1] * 1000, 10)
print(time.time() - start_time)
```

Puis :

```
start_time = time.time()
evaluation([1] * 10000, 10)
print(time.time() - start_time)

start_time = time.time()
horner([1] * 10000, 10)
print(time.time() - start_time)
```

III Exponentiation rapide

On s'intéresse au calcul d'une puissance x^n où x est un réel fixé et n est un entier naturel.

❒ Q12. Ecrire une fonction `puissance(x, n)` qui renvoie la valeur de x^n de manière itérative et sans utiliser la commande `**` de Python.

```
>>> puissance(2, 10)
1024
>>> puissance(5, 2)
25
```

❒ Q13. Déterminer le nombre d'opérations nécessaires à `puissance(x, n)` en fonction de `n`.

❒ Q14. Ecrire une version récursive de `puissance(x, n)`.

Il existe des moyens beaucoup plus rapides de calculer x^n. Par exemple si on écrit n en base 2 (chapitre à venir) :

$$n = \sum_{i=0}^{d} a_i 2^i, \qquad a_i \in \{0, 1\}$$

alors on constate que

$$x^n = x^{a_0} \left(x^2\right)^{a_1} \left(x^{2^2}\right)^{a_2} \cdots \left(x^{2^d}\right)^{a_d}.$$

Ainsi il suffit de d opérations pour calculer tous les x^{2^i} puis d opérations pour former le produit des $\left(x^{2^i}\right)^{a_i}$. Au total il suffit donc de $2d \approx \log(n)$, ce qui est bien mieux que le nombre d'opérations de `puissance(x, n)`.

En pratique, l'algorithme est le suivant et s'écrit naturellement de façon récursive :

- si $n = 1$ alors il n'y a plus rien à faire ;
- si n est pair alors il suffit de calculer x^2 à la puissance $n/2$: $\left(x^2\right)^{n/2} = x^n$;
- si n est impair ($n > 1$) alors il suffit de calculer x^2 à la puissance $(n-1)/2$ et multiplier le tout par x : $\left(x^2\right)^{(n-1)/2} x = x^n$.

❒ Q15. Ecrire la fonction récursive `expo_rapide(x, n)` qui renvoie la valeur de x^n via la méthode décrite ci-dessus.

❒ Q16. Comparer avec les outils `time` de la section précédente les fonctions `puissance(x, n)` et `expo_rapide(x, n)`.