Travaux pratiques n°4

Polynômes
Permutations d'un ensemble fini

F. Cuvellier, J. Tanoh

M.P.S.I. 1, 2018–2019

1 Polynômes réels

1.1 Définitions et notations

Un *polynôme réel* est une suite de nombres réels, indexée par l'ensemble des nombres entiers naturels, dont l'ensemble des indices des éléments non nuls est fini. Pour des raisons qui n'apparaîtront pas ici, un polynôme réel $(a_n)_{n\in\mathbf{N}}$ se note d'ordinaire

$$a_0 + a_1X + a_2X^2 + \cdots + a_nX^n + \cdots$$

ou $\sum_{n=0}^{\infty} a_nX^n$. De façon générale, l'élément d'indice n du polynôme s'appelle le *coefficient* d'indice n du polynôme.
Si le polynôme $f = \sum_{n=0}^{\infty} a_nX^n$ est non nul (c'est-à-dire si l'un des nombres a_n est non nul), le plus grand des indices des éléments non nuls s'appelle le *degré* du polynôme ; on le note $\deg(f)$. On convient que le degré du polynôme nul est $-\infty$.
Si le polynôme $\sum_{n=0}^{\infty} a_nX^n$ est de degré plus petit que N, on le note aussi

$$a_0 + a_1X + a_2X^2 + \cdots + a_NX^N.$$

Étant donnés des polynômes $f = \sum_{n=0}^{\infty} a_nX^n$ et $g = \sum_{n=0}^{\infty} b_nX^n$, on appelle *somme* de f et de g le polynôme

$$\sum_{n=0}^{\infty}(a_n + b_n)X^n,$$

que l'on note $f + g$; on appelle *produit* de f et de g le polynôme

$$\sum_{n=0}^{\infty}\Big(\sum_{h=0}^{n} a_hb_{n-h}\Big)X^n,$$

que l'on note fg.
Enfin, si g est un polynôme non nul, de degré n, il existe un unique couple (q, r) de polynômes réels tel que r soit de degré strictement plus petit que n et que

$$f = gq + r.$$

Le polynôme q s'appelle le *quotient* et r le *reste* de la *division euclidienne* de f par g.

1.2 Algorithmes

On convient de représenter un polynôme réel $f = \sum_{n=0}^{\infty} a_n X^n$ de degré d sous la forme d'un tableau `f` de longueur strictement plus grande que d, telle que `f[0]` soit a_0, `f[1]` a_1, etc. Plusieurs tableaux peuvent donc représenter un polynôme non nul donné, mais un seul d'entre eux a un dernier élément non nul. On l'appellera (ici) le *représentant normalisé* du polynôme. Le polynôme nul peut être représenté par le tableau vide, aussi bien que par tout tableau de zéros. On convient que le tableau vide est le représentant normalisé du polynôme nul.
Les complexités des algorithmes suivants s'exprimeront en fonction des longueurs des tableaux des polynômes, sans tenir compte de la taille des coefficients.

Exercice 1.

1. Donner un algorithme pour transformer un tableau représentant un polynôme en le représentant normalisé. On pourra convenir, ou non, qu'il existe deux fonctions qui modifient un tableau T : l'une qui lui adjoint un nombre x donné en entrée (en Python, `T.append(x)`), l'autre qui en supprime le dernier élément et le retourne en sortie (en Python, `T.pop()`, ici sans argument).

2. Calculer sa complexité, dans le cas le pire et dans le cas optimal.

Pour l'implémentation en Python de cet algorithme, on pourra proposer deux fonctions, une qui retourne le tableau du représentant normalisé, une autre qui modifie le tableau initial sans retourner de valeur.

Exercice 2. Soit $f = \sum_{n=0}^{\infty} a_n X^n$ un polynôme. Pour tout nombre réel x, on note $f(x)$ le nombre réel $\sum_{n=0}^{\infty} a_n x^n$. Il se calcule à l'aide de la formule suivante :

$$f(x) = a_0 + x(a_1 + x(a_2 + x(\ldots)))$$

(méthode de Horner).

1. Expliciter un algorithme de calcul de $f(x)$ fondé sur cette formule.

2. Combien d'additions et combien de multiplications ce calcul nécessite-t-il ?

Exercice 3.

1. Donner un algorithme qui calcule la somme de deux polynômes.

2. Calculer sa complexité, dans le cas le pire et dans le cas optimal.

Exercice 4.

1. Donner un algorithme qui calcule le produit d'un polynôme et d'un monôme (αX^k).

2. Donner un algorithme qui calcule le produit de deux polynômes.

3. Calculer sa complexité, dans le cas le pire et dans le cas optimal.

Exercice 5.

1. Donner un algorithme qui calcule le quotient et le reste de la division euclidienne d'un polynôme par un polynôme non nul.

2. Calculer sa complexité, dans le cas le pire et dans le cas optimal.

1.3 Programmation en Python

Un polynôme est représenté par un tableau de nombres, entiers ou flottants.

Exercice 6. Implémenter ces algorithmes en Python.

Exercice 7.

1. Implémenter en Python l'algorithme d'exponentiation rapide, associé à la multiplication des polynômes.
2. L'appliquer au calcul des coefficients binomiaux. (Celui d'indice (n, k) est le coefficient du terme de degré k du polynôme $(X + 1)^n$.)

Exercice 8.

1. Définir une fonction qui transforme un tableau `[a, b, c, ...]` de nombres en une chaîne de caractères `a+bX+cX^2+...`, en respectant l'usage habituel de noter `-` au lieu de `-+`, `X^k` au lieu de `1X^k` et de ne pas faire apparaître les monômes de coefficient nul, sauf pour le polynôme nul. Par exemple, le tableau `[2, 3, -1, 0, -3]` devra donner la chaîne de caractères `2+3X-X^2-3X^4`.
2. Définir une fonction qui transforme une chaîne de caractères de la forme précédente en le représentant normalisé du polynôme correspondant.

2 Permutations

2.1 Définitions et théorèmes

Une *permutation* d'un ensemble E est une application bijective de E dans lui-même.
La composée de deux permutations de E est une permutation de E. La réciproque de toute permutation de E est une permutation de E.
On dit qu'un élément x de E est *fixe* pour une permutation σ de E si $\sigma(x) = x$. On dit qu'une permutation σ de E est un *cycle* si l'ensemble des éléments de E que σ ne fixe pas est une partie non vide *finie* S de E, de cardinal r (strictement plus grand que 1), qui possède un élément a tel que les r éléments de S soient a, $\sigma(a)$, $\sigma^2(a)$, …, $\sigma^{r-1}(a)$ (dans ce cas, $\sigma^r(a) = a$). La partie S s'appelle le *support* de σ, et le cardinal de S la *longueur* du cycle. En pratique, un cycle de longueur r se note sous la forme d'un r-uplet $(a, \sigma(a), \sigma^2(a), \ldots, \sigma^{r-1}(a))$ (d'une des r façons possibles).
Ici, nous ne considérons que le cas où l'ensemble E est fini, et plus précisément, celui où il est un intervalle $[1, n]$ de l'ensemble $\mathbf{N}$ des nombres entiers naturels.
On décrit généralement une permutation σ de l'intervalle $[1, n]$ de $\mathbf{N}$ comme une matrice à deux lignes et n colonnes :

$$\begin{pmatrix} 1 & 2 & \cdots & n \\ \sigma(1) & \sigma(2) & \cdots & \sigma(n) \end{pmatrix}.$$

L'ordre des colonnes n'a pas d'importance, mais si dans une colonne donnée, k est l'élément de la première ligne, $\sigma(k)$ doit être celui de la seconde ligne. (C'est ainsi que la matrice obtenue en échangeant les deux lignes précédentes représente la permutation σ^{-1}.)
Toute permutation est la composée de cycles dont les supports sont deux à deux disjoints. Par exemple, la permutation

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 1 & 3 & 2 & 6 & 5 & 7 & 4 \end{pmatrix}$$

est la composée des cycles $(2, 3)$ et $(4, 6, 7)$.
Des cycles de supports disjoints commutent, si bien que la décomposition d'une permutation en produit de cycles n'est pas unique, mais à l'ordre près, les cycles qui interviennent dans la décomposition sont bien déterminés par la permutation.
Il existe toutefois une façon canonique de noter une permutation à partir de sa décomposition en produit de cycles : noter un point fixe sous la forme d'un 1-uplet ; noter un cycle de longueur r sous la forme d'un

r-uplet dont le premier élément est le plus petit du support ; enfin, écrire les k-uplets obtenus de sorte que la suite formée par tous les premiers éléments soit *décroissante*. Par exemple, la permutation précédente s'écrit $(5, 4, 6, 7, 2, 3, 1)$.

2.2 Algorithmes

Exercice 9. Donner un algorithme qui vérifie si les éléments d'un n-uplet sont deux à deux distincts.

Exercice 10. Donner un algorithme qui vérifie si une matrice à deux lignes représente une permutation.

Exercice 11.

1. Écrire un algorithme qui transforme une permutation représentée par une matrice de deux lignes en la décomposition en produit de cycles écrite sous forme canonique.
2. Calculer sa complexité, dans le cas le pire et dans le cas optimal.

Exercice 12.

1. Écrire un algorithme qui transforme un n-uplet formé des n nombres 1, 2, ..., n (dans un ordre arbitraire) en la matrice de deux lignes représentant la permutation dont le n-uplet est la forme canonique.
2. Calculer sa complexité, dans le cas le pire et dans le cas optimal.

Exercice 13. Écrire un algorithme qui calcule la composée de deux permutations données.

2.3 Programmation en Python

Exercice 14. Implémenter les algorithmes précédents en Python. Une matrice à deux lignes sera représentée sous la forme d'un tableau de deux tableaux de même longueur.