ITC – TD n°6

Autour des polynômes

Ce TD présente quelques aspects de la représentation en mémoire et du calcul de polynômes ; cela sera l'occasion de pratiquer quelques algorithmes simples.

On rappelle qu'un polynôme P de degré N est une expression de la forme

$$P[X] = \sum_{k=0}^{N} a_k X^k$$

où X est un symbole appelé indéterminée du polynôme. On évalue un polynôme en x_0 en calculant la valeur $\sum_{k=0}^{N} a_k x_0^k$.

I. Travail préliminaire

Récupérer sur cahier-de-prépa le module `module_polynomes.py` et vérifier qu'il s'importe correctement en début de code avec l'instruction

```
import module_polynomes as mod
```

Une fonction d'affichage élégant des polynômes est alors fournie :

```
In [1]: mod.affiche([1, -1, 2, -4, 0, 8])
Out[1]: 8X^5 - 4X^3 + 2X^2 - X^1 + 1
```

II. Représentation en mémoire

Informatiquement, la méthode intuitive pour représenter un polynôme de degré N est d'utiliser une liste de taille N, dont le i^{e} élément est le coefficient a_i du degré i. Par exemple, le polynôme $X^4 + 5X^3 - 2$ est représenté par la liste `[-2, 0, 0, 5, 1]`. Le polynôme nul est représenté par la liste vide. On manipule alors ces listes-polynômes à l'aide de fonctions dédiées pour diverses opérations.

1 – Écrire une fonction `somme` qui prend en paramètres deux listes-polynômes et renvoie une nouvelle liste-polynôme représentant la somme des précédents. On fera attention à enlever les éventuels éléments nuls en fin de liste, si les monômes de haut degré s'annulent. Tester cette fonction à l'aide de l'instruction `mod.tester_somme(somme)` .

2 – Écrire une fonction `produit` qui prend en paramètres deux listes-polynômes et renvoie une nouvelle liste-polynôme représentant le produit des précédents. On pourra remarquer que le produit des monômes de degrés i_1 et i_2 dans les polynômes de départ contribue au degré $i_1 + i_2$ dans le produit. Tester cette fonction à l'aide de l'instruction `mod.tester_produit(produit)` .

III. Évaluation d'un polynôme en une valeur

Dans toute la suite, on interdit l'usage de l'opérateur puissance `x**n` .

3 – Écrire une fonction `évaluation` qui prend en paramètres une liste-polynôme et un nombre x, et calcule monôme par monôme l'évaluation du polynôme en x. Évaluer sa complexité en fonction du degré N du polynôme.

Pour accélérer le calcul, une première amélioration consiste à calculer les puissances dans les monômes à l'aide de l'algorithme d'exponentiation rapide. On rappelle le principe : on peut calculer une puissance de x connaissant une puissance plus petite puisque :

$$x^p = \begin{cases} 1 & \text{si } p = 1 \\ (x^{p//2})^2 & \text{si } p \text{ pair} \\ x(x^{p//2})^2 & \text{si } p \text{ impair} \end{cases}$$

4 – Écrire une fonction `expo_rapide(x, n)` qui calcule x^n en utilisant l'exponentiation rapide. Quelle est sa complexité ?

5 – L'utiliser pour écrire une fonction `évaluation_er` de même signature que `évaluation` qui évalue un polynômes en x. Évaluer un majorant raisonnable de sa complexité en fonction de N.

Pour améliorer encore la rapidité et éviter les dépassements de mémoire (par exemple, pour des entiers sur 64 bits, l'évaluation de $X^{10} - 99X^9$ en $x = 100$ posera ce problème), on utilise la méthode de Ruffini-Horner. L'idée est d'écrire le polynôme sous forme factorisée terme à terme :

$$a_0 + a_1 X + a_2 X^2 + a_3 X^3 \ldots = ((\ldots(((a_n X + a_{n-1}) X + a_{n-2}) X + \ldots)) X + a_1) X + a_0$$

On n'a alors plus qu'à évaluer N produits, et les problèmes présentés précédemment disparaissent.

6 – Écire une fonction `évaluation_rh` de même signature que `évaluation` et qui évalue un polynôme en suivant cette méthode. Évaluer sa complexité.

Application : on peut se servir de cette fonction pour convertir un nombre binaire en entier ; en effet, l'écriture dans une base donnée s'écrit comme une polynôme, donc pour un nombre binaire :

$$11011001 \leftrightarrow X^7 + X^6 + X^4 + X^3 + 1$$

En évaluant alors ce polynôme en $x = 2$, on obtient la valeur décimale associée.

7 – Écrire une fonction `bin2dec` qui prend en paramètre une **chaîne de caractères** représentant un nombre binaire (par exemple `'11011001'`) et renvoie sa conversion en entier (dans notre exemple, 217).

IV. Racines d'un polynôme

On peut construire un polynôme de coefficient 1 au plus haut degré à partir de la liste de ses racines :

$$P[X] = \prod_{k=1}^{N} (X - x_k)$$

8 – Écrire une fonction `construction_racines` qui prend en paramètres une liste de nombres et renvoie un polynôme défini par la formule ci-dessus.

Pour évaluer une racine d'une fonction, on peut utiliser la méthode de Newton en itérant la formule suivante :

$$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})}$$

Il faut donc initialiser la suite avec une estimation initiale du zéro, et on arrête les itérations quand $|x_n - x_{n-1}| < \varepsilon$, où ε est une tolérance choisie (typiquement `1e-12`). Cette méthode présente deux difficultés : elle marche bien si la fonction est suffisamment régulière, et si on peut évaluer le nombre dérivé sans trop d'erreurs numériques. Avec des fonctions polynômiales, ces problèmes ne sont pas vraiment.

9 – Écrire une fonction `dérivée` qui prend en paramètre une liste-polynôme et renvoie une liste représentant le polynôme dérivé.

10 – Écrire une fonction `zéro` qui prend en paramètres une liste-polynôme, une valeur initiale x et un seuil de tolérance, et renvoie un zéro du polynôme obtenu par la méthode de Newton.