DS 1 d'informatique

Corrigé

Polynômes

Partie I – Exemples

1. (a) À quel polynôme correspond la liste `[1]` ?

La liste `[1]` correspond au $\boxed{\text{polynôme constant 1.}}$

(b) À quel polynôme correspond la liste `[0, 1]` ?

La liste `[0, 1]` correspond à $\boxed{\text{l'indéterminée } X.}$

(c) À quel polynôme correspond la liste `[0, 0, 0, 1, 0, 0]` ?

La liste `[0, 0, 0, 1, 0, 0]` correspond au polynôme $\boxed{X^3.}$

2. (a) Donner le `polynôme` qui est la représentation normale de $X^3 + 2X^2 + 3X + 4$.

La représentation normale de $X^3 + 2X^2 + 3X + 4$ est la liste $\boxed{\texttt{[4, 3, 2, 1]}.}$

(b) Donner la représentation normale du polynôme nul.

La représentation normale du polynôme nul est $\boxed{\text{la liste vide } \texttt{[]}.}$

Partie II – Premières fonctions

3. Écrire une fonction `maxCoeffs(P)` qui prend en argument un polynôme `P` et qui renvoie le maximum des valeurs absolues des coefficients de P.

Les coefficients du polynôme nul étant tous nuls, si `P` représente le polynôme nul, on renvoie 0.

```
def maxCoeffs(P):
    if P == []:
        return 0

    maxi = abs(P[0])
    for coeff in P:
        if abs(coeff) > maxi
            maxi =  abs(coeff)
    return maxi
```

4. Écrire une fonction `estNul(P)` qui prend en argument un polynôme `P` et qui renvoie `True` si le polynôme est nul et `False` sinon.

```
def estNul(P):
    for coeff in P:
        if coeff != 0:
            return False
    return True
```

5. Écrire une fonction `degre(P)` qui renvoie le degré d'un polynôme `P`.

```
def degre(P):
    maxDeg = -1
    n = len(P)
    for k in range(n):
        if P[k] != 0:
            maxDeg = k
    if maxDeg = -1:
        return MOINS_INF
    else:
        return maxDeg
```

6. Écrire une fonction `formeNormale(P)` qui prend en argument un `polynôme` `P` et qui renvoie la représentation normale de `P`.

Attention ! La fonction `formeNormale(P)` *ne doit pas modifier le polynôme.*

```
def formeNormale(P):
    copyP = P.copy()
    for k in range(-1, -(n+1), -1):
        if copyP[k] == 0:
            copyP.pop()
        else:
            return copyP

    return copyP
```

Voici une autre réponse possible :

```
def formeNormale(P):
    n = deg(P)

    if deg(P) == MOINS_INF:
        return []

    resultat = []
    for k in range(n+1):
        resultat.append(P[k])

    return resultat
```

Partie III – Premières opérations

7. (a) Écrire une fonction `evaluation(P, alpha)` qui prend en argument un polynôme `P` et un nombre `alpha` et qui renvoie l'évaluation $P(\alpha)$ de P en α.

- Voici un code astucieux qui calcule les puissances de α au fur et à mesure.

```
def evaluation(P, alpha):
    S = 0
    # puissance représente les alpha**k
    puissance = 1

    for coeff in P:
        S = S + coeff * puissance
        # On passe à la puissance suivante
        puissance = alpha * puissance

    return S
```

- Voilà un code moins astucieux mais tout à fait valable (vue la question posée).

```
def evaluation(P, alpha):
    n = deg(P)

    if n == MOINS_INF:
        return 0

    S = 0
    for k in range(n+1):
        S = S + P[k] * alpha**k
    return S
```

(b) Quelle est la complexité de votre fonction ?

On considère un polynôme de degré N.

- Analysons d'abord le code astucieux.
On a N boucles et il y a un nombre constant d'opérations par boucles : la complexité de la fonction astucieuse est $\boxed{\mathrm{O}(N) \text{ opérations.}}$
- Passons au code moins astucieux.
Il y a aussi N boucles mais dans la i-ième boucle, on calcule α^i ce qui nécessite *a priori* i opérations. Au total, à une constante près, on a $\sum_{i=0}^{N} i = \frac{N(N+1)}{2}$ opérations. Ainsi, la complexité de la deuxième fonction est $\boxed{\mathrm{O}\left(N^2\right) \text{ opérations.}}$

8. Écrire une fonction `scalairisation(mu, P)` qui prend en argument un nombre `mu` et un `polynôme` `P` et qui renvoie un `polynôme` représentant la scalairisation μP.

- Voici une première réponse possible.

```
def scalairisation(mu, P):
    n = deg(P)

    if n == MOINS_INF:
        return []

    resultat = P.copy()
    for k in range(n+1):
        resultat[k] = mu * resultat[k]
    return resultat
```

- Voici une autre réponse possible.

```
def scalairisation(mu, P):
    return [mu*coeff for coeff in P]
```

9. Écrire une fonction `somme(P, Q)` qui prend en argument deux `polynômes` `P` et `Q` et qui renvoie un `polynôme` représentant leur somme $P + Q$.

Pour la fonction `somme(P, Q)` :

- il faut commencer par créer une liste de la bonne taille ;
- puis, on somme !

```
def somme(P, Q):
    # On a long_P = deg(P) + 1
    long_P = len(P)
    # On a long_Q = deg(Q) + 1
    long_Q = len(Q)
    N = max(long_P, long_Q)

    resultat = [0]*N

    for k in range(N):
        coeff = 0
        if k < long_P:
            coeff = coeff + P[k]
        if k < long_Q:
            coeff = coeff + Q[k]
        resultat[k] = coeff

    return resultat
```

Partie IV – Dérivation

10. Écrire une fonction `derive(P, k)` qui prend en argument un polynôme `P` et un entier naturel `k` et qui renvoie la dérivée k-ième $P^{(k)}$ de P.

On procède modulairement en définissant d'abord une fonction `derive_aux(P)` qui prend en argument un polynôme `P` qui renvoie la dérivée de P.

```
def derive_aux(P):
    n = len(P)
    return [k*P[k] for k in range(1, n)]
```

On peut alors écrire :

```
def derive(P, k):
    resultat = P
    for i in range(k):
        resultat = derive_aux(resultat)
    return resultat
```

Partie V – Produit

11. (a) Écrire une fonction `produitParX(P)` qui prend en argument un polynôme `P` et qui renvoie un polynôme représentant le produit XP.

```
def produitParX(P):
    n = len(P)
    resultat = [0]*(n+1)
    for k in range(n):
        resultat[k+1] = P[k]
    return resultat
```

(b) Écrire une fonction `produit(P, Q)` qui prend en argument deux polynômes `P` et `Q` et qui renvoie un polynôme représentant leur produit PQ.

Soient P et Q deux polynômes. Si on écrit

$$P = \sum_{k=0}^{n} a_k X^k$$

alors on a

$$PQ = \sum_{k=0}^{n} a_k X^k Q.$$

On procède modulairement et on utilise les fonctions définies précédemment.

```
def produit(P, Q):
    resultat = []
    produitXQ = Q

    for coeff in P:
        terme = scalairisation(coeff, produitXQ)
        resultat = somme(resultat, terme)
        produitXQ = produitParX(produitXQ)

    return resultat
```

Partie VI – Arithmétique polynomiale

Définition

Soient P et Q deux polynômes.

On dit que P divise Q quand il existe un polynôme R tel que $Q = PR$.

12. Écrire une fonction `divisionEuclidienne(A, B)` qui prend en argument deux polynômes `A` et `B` et qui renvoie

- si $B \neq 0$: un `tuple (Q, R)` de polynômes où `Q` est le quotient dans la division euclidienne de A par B et où `R` est le reste.
- si $B = 0$: `None`

On procède modulairement.

- On commence par définir une fonction `coeffDom(P)` qui prend en argument un polynôme `P` qui renvoie le coefficient dominant de P si $P \neq 0$ et qui renvoie `None` sinon.

```
def coeffDom(P):
    if estNul(P):
        return None

    return P[degre(P)]
```

- Puis, on définit une fonction `monomeX(k)` qui prend en argument un entier `k` qui renvoie la représentation normale de X^k.

```
def monomeX(k):
    resultat = [0]*k
    resultat.append(1)

    return resultat
```

On procède de façon récursive.

- Si $\deg(A) < \deg(B)$, alors le quotient est 0 et le reste est A.
- Sinon, on note

  $$\begin{cases} \text{coeff}_A := \text{le coefficient dominant de } A \\ \text{coeff}_B := \text{le coefficient dominant de } B. \end{cases}$$

  On pose

  $$k := \deg(A) - \deg(B) \qquad \text{et} \qquad \mu := \frac{\text{coeff}_A}{\text{coeff}_B}$$

  puis

  $$newA := A - \mu X^k B.$$

  On commence par faire la division euclidienne de $newA$ par B. Si on note (Q, R) le résultat de cette division euclidienne, alors le résultat de la division euclidienne de A par B est

  $$(Q + \mu X^k, R).$$

D'où le code suivant.

```
def divisionEuclidienne(A, B):
    if estNul(B):
        return None

    if degre(A) < degre(B):
        return ([], A.copy())

    mu = coeffDom(A)/coeffDom(B)
    k = deg(A) - deg(B)
    monome = monomeX(k)

    newA = A - scalairisation(mu, produit(monome, B))

    (Q, R) = divisionEuclidienne(newA, B)

    return (somme(Q, scalairisation(mu, monome)), R)
```

13. Écrire une fonction `valuation(P, alpha)` qui prend en argument un `polynôme` `P` et un nombre `alpha` et qui renvoie le plus grand entier k tel que $(X-\alpha)^k$ divise P.
On conviendra que si $P = 0$ alors la fonction renvoie $+\infty$.

On procède modulairement.

- On commence par définir une fonction `divise(P, Q)` qui renvoie `True` si le `polynôme` `P` divise le `polynôme` `Q`, et `False` sinon.
- Pour cela, on utilise le fait que, si $P \neq 0$, P divise Q si et seulement si le reste dans la division euclidienne de Q par P est nul.
- Si $P = 0$, le seul polynôme divisible par P est le polynôme nul.

```
def divise(P, Q):
    if estNul(P):
        return estNul(Q)

    (_, R) = divisionEuclidienne(Q, P)
    return estNul(R)
```

On peut alors écrire le code suivant. On remarque que, pour des raisons de degré, le plus grand entier k cherché est nécessairement $\leqslant \deg(P)$. On est ainsi assuré que la boucle `while` dans le code suivant va s'arrêter.

```
def valuation(P, alpha):
    if estNul(P):
        return PLUS_INF

    # atome représente (X-alpha)
    atome = [-alpha, X]

    # puissance représente les (X-alpha)**k
    puissance = [1]

    k = 0
    while divise(puissance, P):
        puissance = produit(puissance, atome)
        k = k+1

    return k-1
```

Partie VII – Recherche de racines

14. Écrire une fonction `racine(P, a, b, eps)` qui prend en argument un `polynôme` `P`, trois nombres flottants `a`, `b` et `eps` tels que

- `a` $<$ `b`,
- `P(a)` et `P(b)` sont de signes stricts contraires

et qui renvoie un nombre flottant `x` tel que

- `a` $\leqslant$ `x` $\leqslant$ `b`,
- `x` est `eps`-proche d'une racine de P, *ie* il existe une racine α de P vérifiant $|\mathtt{x} - \alpha| \leqslant \mathtt{eps}$.

On procèdera par dichotomie.

L'existence d'une racine de P dans $[a, b]$ est assurée par le théorème des valeurs intermédiaires et par le fait que $P(a)$ et $P(b)$ sont de signes contraires.

On procède par dichotomie.

```python
def racine(P, a, b, eps):
    # On se ramène au cas où P(a) < 0
    if P(a) > 0:
        moinsP = scalairisation(-1, P)
        return racine(moinsP, a, b, eps)

    # Cas général
    mini = a
    maxi = b
    while (maxi - mini) > eps:
        milieu = (mini + maxi)/2
        y = evaluation(P, milieu)
        if y > 0:
            maxi = milieu
        elif y < 0:
            mini = milieu
        else:
            return y

    return (mini + maxi)/2
```

15. (a) Soit $P \in \mathbb{R}[X]$ un polynôme unitaire de degré impair.

(i) Déterminer un réel $b > 0$, dépendant des coefficients de P, tel que $P(b) > 0$.

Il s'agit d'une question de mathématiques.
On écrit

$$P = X^n + c_{n-1}X^{n-1} + \cdots + c_1 X + c_0$$

où $\forall i$, $c_i \in \mathbb{R}$ et on pose $M := \max\limits_{0 \leqslant i \leqslant n-1} |c_i|$.
Soit $b > 1$. On a

$$\begin{aligned}
P(b) &\geqslant b^n - |c_{n-1}|\, b^{n-1} - |c_{n-2}|\, b^{n-2} - \cdots - |c_1|\, b - |c_0| \\
&\geqslant b^n - M\Big(b^{n-1} + \cdots + b + 1\Big) \\
&= b^n - M\frac{b^n - 1}{b - 1} && \text{car } b > 1 \text{ et donc } b \neq 1 \\
&> b^n - M\frac{b^n}{b-1} && \text{car } b - 1 > 0 \\
&= b^n\left(1 - \frac{M}{b-1}\right) = b^n \times \frac{b - 1 - M}{b - 1}.
\end{aligned}$$

Ainsi, si $b - 1 - M \geqslant 0$ *ie* si $b \geqslant M + 1$, on a $P(b) > 0$.
Comme on veut $b > 1$, on peut prendre

$$\boxed{b := \max(2, M+1).}$$

(ii) Déterminer un réel $a < 0$, dépendant des coefficients de P, tel que $P(a) < 0$.

On garde les notations de la question précédente.
Soit $a < -1$. On a

$$\begin{aligned}
P(a) &\leqslant a^n + |c_{n-1}|\, |a|^{n-1} - |c_{n-2}|\, |a|^{n-2} - \cdots - |c_1|\, |a| - |c_0| \\
&\leqslant a^n + M\Big(|a|^{n-1} + \cdots + |a| + 1\Big) \\
&= a^n + M\frac{|a|^n - 1}{|a| - 1} && \text{car } |a| > 1 \text{ et donc } |a| \neq 1 \\
&< a^n + M\frac{|a|^n}{|a| - 1} \\
&= -|a|^n + M\frac{|a|^n}{|a| - 1} && \text{car } n > 1 \text{ est impair} \\
&= |a|^n\left(\frac{M}{|a| - 1} - 1\right) = |a|^n \times \frac{M + 1 - |a|}{|a| - 1}
\end{aligned}$$

Ainsi, si $M + 1 - |a| \leqslant 0$ *ie* si $a \leqslant -M - 1$, on a $P(a) < 0$.
Comme on veut $a < -1$, on peut prendre

$$\boxed{a := \min(-2, -M-1).}$$

(b) Écrire une fonction `racineImpair(P, eps)` qui prend en argument un `polynôme` `P` de degré impair et un nombre flottant `eps`, et qui renvoie un nombre flottant `x` qui est `eps`-proche d'une racine de P.

On utilise la fonction `coeffDom(P)` définie dans la réponse à la question **12.**.
On utilise aussi la fonction `maxCoeffs(P)` de la question **3.**.

```
def racineImpair(P, eps):
    coeffDomP = coeffDom(P)
    Q = scalairisation(1/coeffDomP, P)
    M = maxCoeffs(Q)
    b = max(2, M+1)
    a = min(-2, -M-1)
    return racine(P, a, b, eps)
```