Échangeurs de polynômes

Dans tout le problème on considère exclusivement des polynômes à coefficients réels qui s'annulent en 0. Un tel polynôme s'écrit donc $P = X(a_0 + a_1X + \ldots + a_{m-1}X^{m-1})$; il sera représenté en Python par une liste de flottants `[`$a_0, \ldots, a_{m-1}$`]`, où les nombres a_0 et a_{m-1} ne sont pas nécessairement non nuls ; ainsi un polynôme donné peut être représenté par plusieurs listes de longueurs distinctes.

I Comparaison de deux polynômes en zéro

Q1 Écrire une fonction `valuation(P)` qui renvoie le degré k du monôme (non nul) de `P` de plus petit degré. Cette fonction renvoie 0 si `P` est le polynôme nul.
Par exemple, `valuation([1.5, 0, 3.2, 0.0])` → 1 ;
`valuation([0.0, 0.0])` → 0 ;
`valuation([0.0, 1.5, 0, 3.2, 0.0])` → 2.

Q2 Écrire une fonction `difference(P, Q)` qui prend en argument deux polynômes (dont les longueurs peuvent être différentes) et qui renvoie le polynôme `P-Q`.

On remarque que si le polynôme $P - Q = X(a_0 + a_1X + \ldots + a_{m-1}X^{m-1})$ est de valuation k, alors on a $P(x) - Q(x) \underset{x\to 0}{\sim} a_{k-1}x^k$, donc $P(x) - Q(x)$ est du signe de $a_{k-1}x^k$ au voisinage de 0^+ et au voisinage de 0^-.

Q3 Écrire une fonction `compareNeg(P, Q)` qui prend en argument deux polynômes `P` et `Q` et qui renvoie 1 (resp. -1) si $P(x) > Q(x)$ (resp. $P(x) < Q(x)$) quand x est au voisinage de 0^- et qui renvoie 0 si $P = Q$.

II Permutation de n polynômes

Q4 Écrire une fonction `tri(T)` qui prend en argument une liste `T` de polynômes distincts et qui renvoie la liste des mêmes polynômes triée dans l'ordre décroissant en utilisant la fonction `compareNeg`, c'est à dire qu'elle renvoie $[P_0, ..., P_{n-1}]$ de telle sorte que :

$$P_0(x) > P_1(x) > \cdots > P_{n-1}(x) \text{ au voisinage de } 0^-.$$

On représente une permutation des entiers de 0 à $n-1$ par une liste `pi` formée des entiers de 0 à $n-1$ (une et une seule fois chacun, dans un certain ordre). On dira qu'une telle permutation permute les polynômes $P_0, \ldots, P_{n-1}$ lorsque l'on a simultanément :

- $P_0(x) > P_1(x) > \cdots > P_{n-1}(x)$ au voisinage de 0^-.
- $P_{\texttt{pi[0]}}(x) > P_{\texttt{pi[1]}}(x) > \cdots > P_{\texttt{pi[}n-1\texttt{]}}(x)$ au voisinage de 0^+.

Q5 Trouver la permutation `pi0` qui permute les polynômes 0, $-X^2$ et X^3.
Trouver la permutation `pi1` qui permute les polynômes $-X$, $-X^2$ et X^2.

Q6 Écrire une fonction `verifiePermute(T, pi)` qui renvoie `True` si la permutation `pi` permute les polynômes contenus dans la liste `T` (non triée) et qui renvoie `False` sinon.
Par exemple, `verifiePermute([[0], [0, -1], [0, 0, 1]], [1, 0, 2])` → `True`

III Échangeur de n polynômes

On dira qu'une permutation `pi` des entiers de 0 à $n-1$ est un échangeur s'il existe n polynômes $P_0, \dots, P_{n-1}$ telle que `pi` permute ces polynômes. On admet la caractérisation suivante :
une permutation `pi` des entiers de 0 à $n-1$ est un échangeur si et seulement si, il n'existe aucun quadruplet d'entiers (a, b, c, d) tels que $n-1 \geq a > b > c > d \geq 0$ et :

$$\texttt{pi}[b] > \texttt{pi}[d] > \texttt{pi}[a] > \texttt{pi}[c] \text{ ou } \texttt{pi}[c] > \texttt{pi}[a] > \texttt{pi}[d] > \texttt{pi}[b] \quad (\mathcal{C}_n).$$

Ainsi toute permutation des entiers de 0 à 2 est un échangeur, mais ce n'est pas vrai pour les permutations des entiers de 0 à 3.

Q7 Écrire une fonction `estEchangeurAux(pi, d)` qui renvoie `True` s'il n'existe aucun triplet d'entiers (a, b, c) tels que le quadruplet (a, b, c, d) vérifie la condition $(\mathcal{C}_n)$ ci-dessus, et qui renvoie `False` sinon.

Q8 Écrire une fonction `estEchangeur(pi)` qui renvoie `True` si la permutation `pi` est un échangeur et qui renvoie `False` sinon.

On remarque que, pour que la permutation `pi` des entiers de 0 à $n-1$ ($n \geq 5$) vérifie la condition $(\mathcal{C}_n)$, il faut que cette condition soit en particulier vérifiée par tout quadruplet d'entiers (a, b, c, d) formés d'éléments distincts de `pi[0]`. Or cela revient à dire que la permutation `pi2` des entiers de 0 à $n-2$ vérifie la condition $(\mathcal{C}_{n-1})$, si l'on définit `pi2` par :

$$\forall i \in [\![1, n-1]\!], \quad \texttt{pi2}[i-1] = \begin{cases} \texttt{pi}[i] & \text{si } \texttt{pi}[i] < \texttt{pi}[0] \\ \texttt{pi}[i] - 1 & \text{si } \texttt{pi}[i] > \texttt{pi}[0] \end{cases} \quad (\mathcal{D})$$

Par exemple, si `pi` vaut `[3, 5, 1, 2, 0, 4]`, alors `pi2` vaut `[4, 1, 2, 0, 3]`.

Q9 Écrire une fonction `decaler(pi2, v)`, où `pi2` est une permutation des entiers de 0 à $n-2$ et `v` est un entier de 0 à $n-1$. Cette fonction renvoie la permutation `pi` telle que `pi[0]` vaut `v` et telle que `pi` et `pi2` vérifient la relation $\mathcal{D}$.

On peut ainsi construire les échangeurs des entiers de 0 à $n-1$ à partir des échangeurs des entiers de 0 à $n-2$, à condition de vérifier que les permutations ainsi obtenues sont bien des échangeurs.

Q10 Écrire une fonction récursive `listeEchangeurs(n)` qui renvoie la liste de tous les échangeurs des entiers de 0 à $n-1$ (pour $n \geq 3$).

IV Pour ceux qui ont déjà tout fini

On admet sans démonstration que la relation de récurrence suivante permet de compter le nombre a_n de permutations des entiers 0 à $n-1$ qui sont des échangeurs :

$$a_1 = 1, \quad \forall n \geq 2, a_n = a_{n-1} + \sum_{i=1}^{n-1} a_i a_{n-i}$$

Q11 Écrire une fonction `nombreEchangeurs(n)` qui renvoie le nombre d'échangeurs a_n.
On demande que cette fonction ait une complexité en $\mathcal{O}(n^2)$.

Q12 Écrire une fonction `evaluation(P, x)` qui prend en arguments un polynôme `P` (représenté comme dans l'introduction) et un flottant `x` et qui renvoie la valeur de `P(x)`.

Q13 Donner, en fonction de `len(P)`, le nombre de produits effectués par la fonction précédente - où une puissance i compte pour $i-1$ produits.
Si ce n'est pas déjà le cas, écrire une version de la fonction `evaluation(P, x)` qui utilise moins de `len(P)` produits.

Corrigé

Q1

```
def valuation(P):
    for i in range(len(P)):
        if P[i] != 0:
            return i+1
    return 0

P = [1.5, 0, 3.2, 0.0]; assert valuation(P) == 1
P = [0.0, 0.0]; assert valuation(P) == 0
P = [0.0, 1.5, 0, 3.2, 0.0]; assert valuation(P) == 2
```

Q2

```
def minmax(P, Q):
    if len(P) > len(Q):
        return len(P), len(Q)
    else:
        return len(Q), len(P)

def difference(P, Q):
    M, m = minmax(P, Q)
    R = [0 for _ in range(M)]
    for i in range(m): # P-Q sur la partie commune
        R[i] = P[i]-Q[i]
    for i in range(m, M): # P-Q est complété
        if len(P) == M:
            R[i] = P[i]
        else:
            R[i] = -Q[i]
    return R

P = [1.5, 0, 3.2, 0.0]
Q = [0.0, 1.5, 0, 3.2, 0.0, 4.2]
assert difference(P,Q) == [1.5, -1.5, 3.2, -3.2, -0.0, -4.2]
assert difference(Q,P) == [-1.5, 1.5, -3.2, 3.2, 0.0, 4.2]
```

Q3

```
def compareNeg(P, Q):
    D = difference(P, Q)
    k = valuation(D)
    if k == 0:
        return 0
    else:
        if (D[k-1] > 0 and k%2 == 0) or (D[k-1] < 0 and k%2 == 1):
            return 1
        else:
            return -1

# ou de façon astucieuse
def compareNeg(P, Q):
    D = difference(P, Q)
    k = valuation(D)
    if k == 0:
        return 0
    else:
        return int((D[k-1]//abs(D[k-1]))*(-1)**k)

P = [1.5, 0, 3.2, 0.0]; Q = [0.0, 1.5, 0, 3.2, 0.0, 4.2]
assert compareNeg(P,Q) == -1; assert compareNeg(Q,P) == 1
```

Q4

```
def insere(M, L):
    for i in range(len(M)):
        if compareNeg(M[i], L) in {-1,0}: # tri décroissant
            return M[: i] + [L] + M[i: ]
    return M + [L]

def tri(T):
    '''Tri par insertion (ordre décroissant)'''
    M = []
    for L in T:
        M = insere(M, L)
    return M

T = [[1], [0, 1], [0, 0, 1], [0, 0, 0, 1]]
# X**2 > X**4 > X**3 > X (pour X<0, proche de 0)
assert tri(T) == [[0, 1], [0, 0, 0, 1], [0, 0, 1], [1]]
T = []; assert tri(T) == []

# ou
def partition(T):
    L1, L2 = [], []
    for i in range(1,len(T)):
        if compareNeg(T[i], T[0]) in {0, 1}: # T[i] >= T[0]
            L1.append(T[i])
        else:
            L2.append(T[i])
    return [L1, L2]

def tri(T):
    '''Tri rapide (ordre décroissant)'''
    if len(T) == 0 or len (T) == 1:
        return T
    else:
        L1, L2 = partition(T)
        return tri(L1) + [T[0]] + tri(L2)

T = [[1], [0, 1], [0, 0, 1], [0, 0, 0, 1]]
# X**2 > X**4 > X**3 > X (pour X<0, proche de 0)
assert tri(T) == [[0, 1], [0, 0, 0, 1], [0, 0, 1], [1]]
T = []; assert tri(T) == []
```

Q5

- Au voisinage de 0^-, on a $0 > x^3 > -x^2$, donc $P_0 = 0, P_1 = X^3$ et $P_2 = -X^2$.
  Au voisinage de 0^+, on a $x^3 > 0 > -x^2$, donc `pi0[0]`$= 1$, `pi0[1]`$= 0$ et `pi0[2]`$= 2$.
  Ainsi, `pi0 == [1, 0, 2]`.
- Au voisinage de 0^-, on a $-x > x^2 > -x^2$, donc $P_0 = -X, P_1 = X^2$ et $P_2 = -X^2$.
  Au voisinage de 0^+, on a $x^2 > -x^2 > -x$, donc `pi1[0]`$= 1$, `pi1[1]`$= 2$ et `pi1[2]`$= 0$.
  Ainsi, `pi1 == [1, 2, 0]`.

Q6

```
def comparePos(P, Q):
    D = difference(P, Q)
    k = valuation(D)
    if k == 0:
        return 0
    else:
        return int((D[k-1]//abs(D[k-1])))


```

```python
def verifiePermute(T, pi):
    TT = tri(T)
    for i in range(len(TT)-1):
        if comparePos(TT[pi[i]], TT[pi[i+1]]) != 1:
            return False
    return True

assert verifiePermute([[0], [0, -1], [0, 0, 1]], [1, 0, 2]) == True
assert verifiePermute([[-1], [0, 1], [0, -1]], [1, 2, 0]) == True
assert verifiePermute([[-1], [0, 1], [0, -1]], [2, 1, 0]) == False
```

Q7

```python
def estEchangeurAux(pi, d):
    n = len(pi)
    for a in range(d+1, n):
        for b in range(d+1, a):
            for c in range(d+1, b):
                if pi[b] > pi[d] > pi[a] > pi[c] or pi[c] > pi[a] > pi[d] > pi[b]:
                    return False
    return True

assert estEchangeurAux([1, 0, 2, 3], 3) == True

# ou
def estEchangeurAux(pi, d):
    n = len(pi)
    for c in range(d+1, n):
        for b in range(c+1, n):
            for a in range(b+1, n):
                if pi[b] > pi[d] > pi[a] > pi[c] or pi[c] > pi[a] > pi[d] > pi[b]:
                    return False
    return True

assert estEchangeurAux([1, 0, 2, 3], 3) == True
```

Q8

```python
def estEchangeur(pi):
    n = len(pi)
    for d in range(n):
        if not estEchangeurAux(pi, d):
            return False
    return True

assert estEchangeur([1, 0, 2, 3]) == True
assert estEchangeur([1, 3, 0, 2]) == False
assert estEchangeur([2, 0, 3, 1]) == False
```

Q9

```python
def decaler(pi2, v):
    pi = [v]
    for i in range(len(pi2)):
        if pi2[i] < v :
            pi.append(pi2[i])
        else:
            pi.append(pi2[i] + 1)
    return pi

v = 3; pi2 = [4, 1, 2, 0, 3]; assert decaler(pi2, v) == [3, 5, 1, 2, 0, 4]
```

Q10

```
174 def listeEchangeurs(n):
175     if n == 1:
176         return [[0]]
177     else:
178         liste = []
179         for pi2 in listeEchangeurs(n-1):
180             for v in range(n):
181                 pi = decaler(pi2, v)
182                 if estEchangeur(pi):
183                     liste.append(pi)
184         return liste
185 
186 assert listeEchangeurs(4) == [[0, 1, 2, 3], [1, 0, 2, 3], [2, 0, 1, 3],
    [3, 0, 1, 2], [0, 2, 1, 3], [1, 2, 0, 3], [2, 1, 0, 3], [3, 1, 0, 2],
     [0, 3, 1, 2], [2, 3, 0, 1], [3, 2, 0, 1], [0, 1, 3, 2], [1, 0, 3,
    2], [3, 0, 2, 1], [0, 2, 3, 1], [1, 2, 3, 0], [2, 1, 3, 0], [3, 1, 2,
     0], [0, 3, 2, 1], [1, 3, 2, 0], [2, 3, 1, 0], [3, 2, 1, 0]]
```

Q11

```
189 def somme(a, k):
190     somme = 0
191     for i in range(1, k+1):
192         somme = somme + a[i] * a[k+1-i]
193     return somme
194 
195 def nombreEchangeurs(n):
196     a = [0, 1] # a0 = 0 arbitrairement et a1 = 1
197     for k in range(1, n):
198         a.append(a[k] + somme(a, k))
199     return a[n]
200 
201 assert nombreEchangeurs(1) == 1
202 assert nombreEchangeurs(2) == 2
203 assert nombreEchangeurs(3) == 6
204 assert nombreEchangeurs(4) == 22
```

Q12

```
207 def evaluation(P, x):
208     p = len(P)
209     eval = 0
210     for i in range(p):
211         eval = eval + P[i] * x**i
212     return x * eval
213 
214 assert evaluation([1, 3],2) == 14 # P=X+3X**2 évalué en 2
```

Q13 Soit $p = \texttt{len(P)}$.
La fonction précédente utilise $p + \sum_{i=0}^{p-1} i + 1 = p + 1 + \frac{(p-1)p}{2} = \frac{p(p+1)}{2} + 1$ multiplications.
On peut réduire ce total à p multiplications avec l'algorithme de Hörner :

```
217 def evaluation(P, x):
218     '''algorithme de Hörner'''
219     eval = P[len(P)-1]
220     for j in range(len(P)-1, 0, -1) :
221         eval = x * eval + P[j-1]
222     return x * eval
223 
224 assert evaluation([1, 3],2) == 14 # P=X+3X**2 évalué en 2
```