Corrigé : Échangeurs de polynômes (X mp 2010)

Partie I. Permutation de n polynômes

Question 1. On évalue $P(x)$ à l'aide du schéma de Hörner :

```
def evalue(p, v):
    s = 0
    for a in reversed(p):
        s = v * s + a
    return v * s
```

Question 2.

```
def valuation(p):
    for (k, a) in enumerate(p):
        if a != 0:
            return k + 1
    return 0
```

Question 3. Puisqu'un polynôme peut admettre plusieurs représentations, on choisit de retourner la représentation de $P_1 - P_2$ sous la forme d'un tableau dont la taille est égale à la plus grande des deux tailles des tableaux représentant P_1 et P_2 :

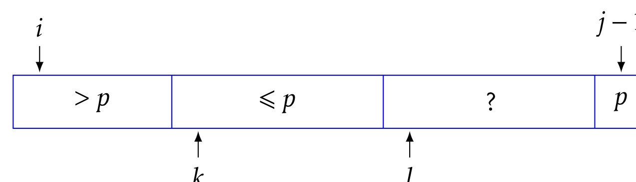
```
def difference(p1, p2):
    n = max(len(p1), len(p2))
    p = [0] * n
    for (k, a) in enumerate(p1):
        p[k] += a
    for (k, a) in enumerate(p2):
        p[k] -= a
    return p
```

Question 4. Si $k \neq 0$ désigne la valuation de $P_1 - P_2$ et a le coefficient de x^k alors $P_1(x) - P_2(x) \sim ax^k$ donc la position relative des deux polynômes pour de petites valeurs négatives de x dépend de la parité de k et du signe de a :

- on a $P_1(x) < P_2(x)$ lorsque $a < 0$ et k pair ou lorsque $a > 0$ et k impair ;
- on a $P_1(x) > P_2(x)$ lorsque $a < 0$ et k impair ou lorsque $a > 0$ et k pair.

```
def compare_neg(p1, p2):
    p = difference(p1, p2)
    k = valuation(p)
    if k == 0:
        return 0
    a = p[k-1]
    if a < 0 and k % 2 == 0 or a > 0 and k % 2 == 1:
        return -1
    else:
        return 1
```

Question 5. Aucune méthode de tri n'est imposée, je choisis d'utiliser *quicksort*. La fonction de segmentation maintient l'invariant représenté ci-dessous lors de l'énumération puis permute le pivot p avec l'élément d'indice k.



```
def segmente(t, i, j):
    p = t[j-1]
    k = i
    for l in range(i, j-1):
        if compare_neg(p, t[l]) < 0:
            t[k], t[l] = t[l], t[k]
            k += 1
    t[k], t[j-1] = t[j-1], t[k]
    return k
```

Une fois la portion $t[i:j]$ segmentée, le tri se poursuit récursivement dans $t[i:k]$ et dans $t[k+1:j]$:

```
def tri(t, *args):
    if len(args) == 0:
        i, j = 0, len(t)
    else:
        i, j = args
    if j > i + 1:
        k = segmente(t, i, j)
        tri(t, i, k)
        tri(t, k+1, j)
```

Question 6. On commence par définir la fonction `compare_pos`, qui cette fois ne dépend que du signe de a :

```
def compare_pos(p1, p2):
    p = difference(p1, p2)
    k = valuation(p)
    if k == 0:
        return 0
    a = p[k-1]
    if a < 0:
        return -1
    else:
        return 1
```

Il s'agit ensuite de vérifier les inégalités $P_{\pi(i)}(x) < P_{\pi(i+1)}(x)$ pour x positif assez petit :

```
def verifier_permute(pi, t):
    for i in range(len(pi)-1):
        if compare_pos(t[pi[i]], t[pi[i+1]]) >= 0:
            return False
    return True
```

Partie II. Échangeurs de *n* polynômes

Question 7. On applique une démarche naïve de coût $O(n^3)$:

```
def est_echangeur_aux(pi, d):
    n = len(pi)
    for c in range(d+1, n-2):
        for b in range(c+1, n-1):
            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
```

Question 8. Il aurait été tout à fait possible d'imbriquer les quatre énumérations des entiers a, b, c et d dans une seule fonction ; la raison pour laquelle l'énumération de d est dissociée de l'énumération des trois autres variables a, b et c n'apparaîtra que dans la dernière question.

```
def est_echangeur(pi):
    for d in range(len(pi)-3):
        if not est_echangeur_aux(pi, d):
            return False
    return True
```

Question 9. Utiliser une version récursive sans mémoïsation conduirait à un coût excessif, aussi je choisit une version itérative consistant à remplir un tableau avec les valeurs de a déjà calculées. On convient arbitrairement de poser $a(0) = 0$ pour que les indices du tableau a coïncident avec les valeurs de n.

```
def nombre_echangeurs(n):
    a = [0]*(n+1)
    a[1] = 1
    for k in range(2, n+1):
        for i in range(1, k):
            a[k] += a[i] * a[k-i]
        a[k] += a[k-1]
    return a[n]
```

Question 10.

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

Question 11. J'ai choisi de rédiger une fonction récursive. Une fois obtenu les échangeurs de rang $n-1$, on décale chacun d'eux à l'aide de la fonction précédente puis on détermine ceux d'entre-eux qui sont des échangeurs.

```
def enumerer_echangeurs(n):
    if n == 1:
        return [[1]]
    t = enumerer_echangeurs(n-1)
    e = []
    for v in range(1, n+1):
        for i in range(len(t)):
            u = decaler(t[i], v)
            if est_echangeur(u):
                e.append(u)
    return e
```

Cependant, le lecteur attentif remarquera qu'il ne nous est pas demandé d'utiliser la fonction `est_echangeur` (de coût $O(n^4)$) mais d'utiliser la fonction `est_echangeur_aux` (de coût $O(n^3)$). En effet, si $t[i]$ est un échangeur et u un décalage de celui-ci, et s'il existe a, b, c, d vérifiant l'une des conditions (1) dans u alors nécessairement $d = 1$. La version ci-dessous est donc légèrement plus efficace que la précédente :

```
def enumerer_echangeurs(n):
    if n == 1:
        return [[1]]
    t = enumerer_echangeurs(n-1)
    e = []
    for v in range(1, n+1):
        for i in range(len(t)):
            u = decaler(t[i], v)
            if est_echangeur_aux(u, 0):
                e.append(u)
    return e
```

Cela n'en reste pas moins une fonction de coût très important (vraisemblablement exponentiel).