Corrigé

Transformée de Fourier rapide

1. Manipulation de polynômes

1.1 Représentation par les coefficients

Les trois premières questions, élémentaires, vont nous permettre de réviser quelque peu les fonctions qui agissent sur les itérateurs.

Question 1. La fonction `reversed` permet d'inverser l'ordre des valeurs d'un itérable.

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

Cette fonction effectue n additions et n multiplications ; elle est de coût linéaire.

Question 2. La fonction `zip` renvoie un itérateur calculant les couples d'éléments de même position dans deux itérables.

```
def addition(p, q):
    r = []
    for (a, b) in zip(p, q):
        r.append(a+b)
    return r
```

Cette fonction effectue n additions ; elle est de coût linéaire.

Question 3. La fonction `enumerate` prend en argument un itérable et crée un itérateur renvoyant les couples formés de l'indice et de l'élément associé dans cet itérable.

```
def produit(p, q):
    r = [0] * (len(p) + len(q) - 1)
    for (j, a) in enumerate(p):
        for (k, b) in enumerate(q):
            r[j+k] += a * b
    return r
```

Cette fonction effectue n^2 additions et n^2 multiplications ; elle est de coût quadratique.

1.2 Représentation par échantillonnage

Question 4. Le calcul de la représentation par échantillonnage nécessite le calcul de n évaluations : les valeurs de $p(x_0), p(x_1), \dots, p(x_{n-1})$. La conversion par l'algorithme naïf est donc de coût quadratique.

Question 5. Considérons la matrice de Vandermonde V associée aux valeurs $(x_0, x_1, \dots, x_{n-1})$:

$$V = \begin{pmatrix} 1 & x_0 & x_0^2 & \dots\dots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \dots\dots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \dots\dots & x_2^{n-1} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \dots\dots & x_{n-1}^{n-1} \end{pmatrix}$$

Nous avons $V\vec{p} = \vec{y}$, donc le calcul de $\vec{p}$ à partir de $\vec{y}$ se ramène à la résolution d'un système linéaire de n équations à n inconnues. L'algorithme du pivot de Gauss étudié en première année permet de résoudre de tels systèmes pour un coût en $O(n^3)$. Cependant, les valeurs des x_j étant fixées, il est plus intéressant de calculer une fois pour toute la matrice V^{-1} ; il reste alors à calculer le produit $\vec{p} = V^{-1}\vec{y}$, ce qui peut être réalisé pour un coût en $O(n^2)$.

2. Transformée de Fourier rapide

2.1 Diviser pour régner

Question 6. Chacun des deux appels récursifs des points 1. et 2. a un coût égal à $C(n/2)$; l'opération décrite dans le point 3. est de coût linéaire donc $C(n)$ vérifie la relation :

$$C(n) = 2C(n/2) + O(n).$$

On résout cette relation en posant $n = 2^k$ et $c_k = C(2^k)$. Alors $c_k = 2c_{k-1} + O(2^k)$, soit encore : $\frac{c_k}{2^k} = \frac{c_{k-1}}{2^{k-1}} + O(1)$. Par télescopage on en déduit que $\frac{c_k}{2^k} = O(k)$, soit $c_k = O(k2^k)$. Puisque $n = 2^k$ ceci conduit à : $C(n) = O(n \log n)$.

Question 7. Soit X un ensemble contractant. Puisque X^2 est contractant et de cardinal $n/2$, il est évident que n doit être une puissance de 2. Posons $n = 2^k$; alors X^{2^k} est de cardinal 1 donc il s'écrit : $X^{2^k} = \{z\}$. Considérons alors un élément $x \in X$. On a $x^2 \in X^2$ et plus généralement $x^{2^k} \in X^{2^k}$ donc $x^{2^k} = x^n = z$. Puisque $\operatorname{card} X = n$, X est exactement l'ensemble des racines n-ième de z.
Réciproquement, on démontre sans peine par récurrence que l'ensemble des racines 2^k-ième d'un nombre complexe $z \neq 0$ est contractant.

2.2 Transformée de Fourier discrète

Question 8.

```
import cmath as cm

def fft(p):
    n = len(p)
    if n == 1:
        return p
    q, r = fft(p[0::2]), fft(p[1::2])
    s = [0] * len(p)
    omega = cm.rect(1, 2*cm.pi / n)
    x = 1
    for j in range(n//2):
        s[j] = q[j] + x * r[j]
        s[j+n//2] = q[j] - x * r[j]
        x = x * omega
    return s
```

On notera que cette fonction utilise la relation : $\omega_n^{n/2+j} = -\omega_n^j$, ce qui permet d'économiser $n/2$ multiplications. En effet,

$$p(\omega_n^j) = q(\omega_n^{2j}) + \omega_n^j r(\omega_n^{2j}) = q(\omega_{n/2}^j) + \omega_n^j r(\omega_{n/2}^j) \quad \text{et} \quad p(\omega_n^{n/2+j}) = p(-\omega_n^j) = q(\omega_n^{2j}) - \omega_n^j r(\omega_n^{2j}) = q(\omega_{n/2}^j) - \omega_n^j r(\omega_{n/2}^j)$$

2.3 Inverser la FFT

Question 9. Calculons les coefficients α_{ij} de la matrice $V\overline{V}$: $\alpha_{ij} = \sum_{k=0}^{n-1} \omega_n^{ik}\omega_n^{-kj} = \sum_{k=0}^{n-1} (\omega_n^{i-j})^k$.

Si $i \neq j$ la raison de cette somme géométrique est différente de 1 et $\alpha_{ij} = \frac{1-(\omega_n^n)^{i-j}}{1-\omega_n^{i-j}} = 0$; si $i = j$ la raison est égale à 1 et $\alpha_{ii} = n$. On a donc $V\overline{V} = nI$, ce qui montre que $V^{-1} = \frac{1}{n}\overline{V}$.

Question 10. Si on remplace ω_n par son conjugué dans l'algorithme précédent on obtient un algorithme qui calcule le produit d'un vecteur par la matrice $\overline{\mathrm{V}}$; il faut encore multiplier le résultat par $\frac{1}{n}$ pour obtenir le résultat souhaité, ce qu'on réalise récursivement en se souvenant que n est une puissance de 2 :

```
def ifft(p):
    n = len(p)
    if n == 1:
        return p
    q, r = ifft(p[0::2]), ifft(p[1::2])
    s = [0] * len(p)
    omega = cm.rect(1, -2*cm.pi / n)
    x = 1
    for j in range(n//2):
        s[j] = (q[j] + x * r[j]) / 2
        s[j+n//2] = (q[j] - x * r[j]) / 2
        x = x * omega
    return s
```

2.4 Multiplication rapide de deux polynômes

Question 11. Si p et q sont deux polynômes de degrés d_1 et d_2 le produit sera de degré $d_1 + d_2$ donc il faut commencer par calculer l'entier n égal à la plus petite des puissances de 2 qui dépasse strictement $d_1 + d_2$ et échantillonner à l'aide de cet entier.

```
def fastproduct(p, q):
    n = 1
    while n < len(p)+len(q)-1:
        n *= 2
    p += [0] * (n-len(p))
    q += [0] * (n-len(q))
    r = [x * y for (x, y) in zip(fft(p), fft(q))]
    return ifft(r)
```

♦ ♦
♦