TP Caml 6: Multiplication de polynômes

lionel.rieg@ens-lyon.fr

13 & 20 janvier 2009

Dans ce TP, nous allons nous intéresser à la multiplication de polynômes, qui peut s'adapter à celle des grands entiers. Pour permettre aux polynômes d'avoir des degrés arbitraires, on va utiliser un type « extensible », les listes, qui contiendront les coefficients du polynôme et qu'on va ordonner par degrés croissants. Le type des éléments dépendra des coefficients des polynômes, int, float voire num. On prend ici le type int pour simplifier (sauf indication contraire) mais tout resterait valable avec float ou num.

► **Question 1** Écrire les polynômes 0, X et $4X^3 - 2X^2 + 1$ avec cette représentation. Programmer ensuite le calcul du degré.

1 Multiplication naïve

On s'intéresse ici à la multiplication selon la formule théorique : $P \cdot Q = \sum_{n \geqslant 0} \left(\sum_{j+k=n} p_j \cdot q_k \right) X^n = \sum_{j \geqslant 0} p_j X^j \cdot Q$.

► **Question 2** Écrire des fonctions d'addition et de soustraction de polynômes ainsi que la multiplication par un monôme

```
monomial_mult : int  list  →  int  →  int  →  int  list
```

telle que monomial_mult p a n réalise la multiplication du polynôme p par le monôme aX^n.

► **Question 3** Écrire l'algorithme de multiplication naïve. Quelle est sa complexité ?

2 Algorithme de Karatsuba

Pour diminuer la complexité de la multiplication de polynômes, on va utiliser un paradigme très classique de programmation : *diviser pour régner*. Pour cela, on va décomposer les polynômes à multiplier P et Q de degrés strictement inférieurs à $2n$ en

$$P = P_1 + P_2 \cdot X^n \qquad \text{et} \qquad Q = Q_1 + Q_2 \cdot X^n$$

avec les degrés de P_1, P_2, Q_1 et Q_2 strictement inférieurs à n.

► **Question 4** Écrire une formule qui réduit la multiplication des polynômes P et Q de degrés strictement inférieurs à $2n$ en multiplications de polynômes de degrés strictement inférieurs à n.

► **Question 5** Programmer un algorithme récursif de multiplication qui utilise la formule précédente. Quelle est sa complexité ?

On peut raffiner cette méthode avec la remarque suivante de Karatsuba : le terme intermédiaire de $P \cdot Q$ s'écrit

$$P_1 \cdot Q_2 + P_2 \cdot Q_1 = (P_1 + P_2) \cdot (Q_1 + Q_2) - P_1 Q_1 - P_2 Q_2$$

On échange donc deux multiplications et une addition contre une multiplication et quatre additions.

► **Question 6** Écrire une fonction qui réalise la multiplication de polynômes à la Karatsuba.

► **Question 7** Trouver la formule de récurrence qui définit la complexité de la multiplication de Karatsuba. Quelle est sa solution ?

3 Algorithme de Schönhage-Strassen

Dans cette partie, on va étudier une technique qui est asymptotiquement plus rapide que celle de Karatsuba et qui se base sur la transformée de Fourier rapide. Elle n'est utilisée que pour les grandes valeurs de n, qui doit être une puissance de 2. Le principe de cette technique est de remarquer que l'évaluation d'un produit ou d'une somme de polynômes se fait facilement à partir des valeurs des polynômes à multiplier ou sommer. De plus, l'interpolation de Lagrange permet de revenir des valeurs aux coefficients.

Cette idée peut se voir comme un changement de représentation : on passe d'une représentation par coefficients à une représentation par valeurs. C'est un compromis car certaines opérations comme déterminer le degré ou la valuation, triviales avec les coefficients, ne le sont plus avec des valeurs. Ce qui rend cette idée viable, c'est le fait que l'évaluation et l'interpolation sont des opérations pas trop coûteuses (par la transformée de Fourier) comme nous allons le voir.

3.1 (Multi-)Évaluation

L'évaluation va se faire dans $\mathbb{C}$, choix qui se justifiera dans la suite. Pour représenter les nombres complexes, on prend une représentation cartésienne avec le type suivant :

```
type complex = { Re : float ; Im : float };;
```

► **Question 8** Écrire les fonctions d'arithmétique usuelle sur les complexes : addition, soustraction, multiplication, division et conjugaison. Programmer aussi les complexes 0 et 1 ainsi que des fonctions de conversion entre entiers et complexes.

► **Question 9** Écrire un algorithme d'évaluation de polynômes. Il sera de type

```
eval : int list → complex → complex
```

Quelle est sa complexité en nombres d'additions et de multiplications ?

L'évaluation d'un polynôme peut se faire de façon incrémentale par le schéma de Horner qui consiste à remarquer que le polynôme $P = a_nX^n + a_{n-1}X^{n-1} + \cdots + a_1X + a_0$ peut se récrire $P = (\ldots((a_nX) + a_{n-1})X + \cdots + a_1)X + a_0$. Il ne faut donc que $2n$ opérations.

► **Question 10** Améliorer l'algorithme précédent à l'aide du schéma de Horner. Pourrait-on faire mieux que $\Theta(n)$ opérations ?

Maintenant que l'on sait évaluer un polynôme en un point, on peut utiliser cet algorithme n fois sur P et Q pour obtenir les évaluations qui permettront de reconstruire le produit $P \cdot Q$ par interpolation. Malheureusement, on obtient ainsi un algorithme en $\Theta(n^2)$, qui n'est pas meilleur que la méthode naïve. Il nous faut donc économiser des calculs lors des évaluations futures en réutilisant ceux effectués auparavant, et ce par un choix judicieux des points d'évaluation. Par exemple, si $b = -a$, alors les termes de degrés pairs de $P(a)$ et $P(b)$ sont les mêmes et ceux de degrés impairs sont opposés. En fin de compte, les candidats naturels comme points d'évaluation sont les racines (complexes) n^{e} de l'unité. Dans la suite, on note ω_n une racine primitive n^{e} de l'unité, par exemple $e^{\frac{2i\pi}{n}}$.

► **Question 11** Écrire une fonction de découpage d'un polynôme en ses parties paire et impaire : son résultat sur un polynôme P sera le couple (P_{pair}, P_{impair}) tel que $P(X) = P_{pair}(X^2) + X \cdot P_{impair}(X^2)$.

Évaluer P en ω_n^k et $-\omega_n^k = \omega_n^{k+\frac{n}{2}}$ se ramène à évaluer P_{pair} et P_{impair} en ω_n^{2k} plus quelques multiplications et additions/soustractions (si l'on suppose déjà connus les ω_n^k pour $k \in [\![0, n-1]\!]$). On est donc naturellement porté à utiliser une solution récursive et cela justifie d'utiliser des racines de l'unité car elles forment un groupe multiplicatif : on connaît déjà les $\omega_n^{2k} = \omega_{n/2}^k$ car ce sont des ω_n^k.

► **Question 12** Écrire une fonction qui prend en argument les ω_n^k pour $k \in [\![0, n-1]\!]$ et les valeurs de P_{pair} et P_{impair} en les $\omega_n^{2k} = \omega_{n/2}^k$ pour $k \in \left[\!\left[0, \frac{n}{2} - 1\right]\!\right]$ et construit les valeurs de P en les ω_n^k pour $k \in [\![0, n-1]\!]$.

► **Question 13** Écrire un algorithme récursif qui réalise l'évaluation d'un polynôme (à coefficients complexes) en les n puissances successives d'une racine primitive n^{e} de l'unité ω_n qui seront données en argument. (c'est la transformée de Fourier du polynôme)

► **Question 14** Écrire un algorithme qui calcule la liste des puissances successives ordonnées par ordre croissant d'un nombre complexe donné en argument. Il renverra la liste $[1; \omega; \omega^2; \ldots; \omega^{n-1}]$ si on lui passe ω et n. Il sera donc de type

```
compute_powers : complex → int → complex list
```

A-t-on intérêt ici à utiliser l'exponentiation rapide ?

3.2 Interpolation

► **Question 15** Expliquer pourquoi l'interpolation est une opération aussi simple que l'évaluation (penser que l'évaluation est une opération linéaire et possède donc une matrice associée).

► **Question 16** Calculer le résultat de la composée d'une évaluation en les ω_n^k suivie d'une autre en les ω_n^{-k} ($k \in [\![0, n-1]\!]$) pour un polynôme de degré inférieur à n. On considérera la première évaluation comme les coefficients du polynôme pour la seconde.

► **Question 17** À l'aide de la question précédente, écrire un algorithme d'interpolation de polynômes en les ω_n^k pour $k \in [\![0, n-1]\!]$.

► **Question 18** Combiner toutes les fonctions écrites jusqu'à présent et écrire la multiplication par FFT.

► **Question 19** Quelle est la complexité de l'algorithme de multiplication de Schönhage-Strassen ?

4 Solutions

▸ **Question 1** On décide qu'une liste vide n'est pas à un polynôme pour avoir unicité de la représentation du polynôme nul.

```
let zero = [0];;
let x = [0; 1];;
let poly = [1; 0; -2 ;4];;

let deg p = list_length p - 1;;
```

On prend ici la convention que le polynôme nul est de degré zéro. Cela simplifie la suite lors de traitements particuliers pour les polynômes constants mais il faut garder à l'esprit que ce n'est pas l'habitude et que l'on a plus dans ce cas les relations qui expriment le degré ou la valuation d'un produit comme somme de ceux des termes.

▸ **Question 2** L'addition et la soustraction sont des opérations qui peuvent faire apparaître des zéros dans les coefficients dominants (ce sont d'ailleurs les seules dans un corps). La représentation du polynôme est alors impropre car il utilise d'avantage de place que nécessaire mais surtout la fonction deg ne fonctionne plus. D'où l'intérêt d'une fonction de normalisation :

```
let normalize p =
  let rec normalize_aux = function
    | [] -> []
    | 0 :: q -> let q2 = normalize_aux q in if q2 = [] then [] else 0 :: q2
    | t :: q -> t :: normalize_aux q
  in
    match p with
      | [] -> failwith "normalize: empty polynomial"
      | a_0 :: q -> a_0 :: normalize_aux q;;
```

La fonction auxiliaire map_op généralise map2 à deux listes de taille différentes. Ne pas oublier dans l'addition et la soustraction les normalisations finales. La multiplication par un monôme se décompose en deux étapes : la multiplication des coefficients de la liste par un scalaire puis un ajout de zéros en tête de liste.

```
let rec map_op op p1 p2 =
  match p1, p2 with
    | [],p -> p
    | p,[] -> p
    | a1 :: p1',a2 :: p2' -> (op a1 a2) :: map_op op p1' p2';;

let add p1 p2 = normalize (map_op (prefix +) p1 p2);;
let subtract p1 p2 = add p1 (map (fun a -> - a) p2);;

let rec add_zeros p = function
  | 0 -> p
  | n -> add_zeros (0 :: p) (n - 1);;

let monomial_mult p a_n n =
  add_zeros (map (fun c -> a_n * c) p) n;;
```

▸ **Question 3** On développe Q en somme de monômes et on applique la formule donnée au début de cette partie. Il faut ajouter pour cela un paramètre n pour connaître le degré correspondant à chaque monôme de Q.

```
let rec term_rec_naive_mult p_1 n acc = function
  | [] -> acc
  | t :: q ->
      term_rec_naive_mult p_1 (n + 1) (add acc (monomial_mult p_1 t n)) q;;

let naive_mult p_1 p_2 =
  if p_1 = [] || p_2 = []
  then failwith "naive_mult: empty polynomial"
  else term_rec_naive_mult p_1 0 zero p_2;;
```

Notons p et q les degrés respectifs des polynômes P et Q. La complexité se détermine facilement en remarquant que tous les produits entre un coefficient de P et un de Q interviennent dans le calcul des coefficients de $P \cdot Q$ ($p_i \cdot q_j$ intervient dans le coefficient de degré $i + j$). Il y a ainsi pq produits à faire. Enfin, il faut réduire ces pq nombres en les $p + q + 1$ coefficients de $P \cdot Q$ à l'aide d'additions. Chaque addition combine deux valeurs en une seule donc fait décroître leur nombre de 1. Il faut donc $pq - p - q - 1$ additions pour réduire les pq produits en $p + q + 1$ termes. Le coût total est donc $2pq - p - q - 1 = \Theta(pq)$.

▸ **Question 4** Il suffit de développer le produit $(P_1 + X^n P_2) \cdot (Q_1 + X^n Q_2)$:

$$(P_1 + X^n P_2) \cdot (Q_1 + X^n Q_2) = P_1 Q_1 + X^n \cdot (P_1 Q_2 + P_2 Q_2) + X^{2n} \cdot P_2 Q_2$$

On se ramène ainsi aux quatre multiplications $P_1 Q_1$, $P_1 Q_2$, $P_2 Q_1$ et $P_2 Q_2$ entre polynômes de degrés strictement inférieurs à n, plus deux multiplications par X^n et X^{2n} qui ne sont que des ajouts de zéros en tête de liste.

▸ **Question 5** On sépare les deux étapes de l'algorithme : d'abord la découpe des polynômes (dans laquelle il ne faut pas oublier de donner n en argument car ce n'est pas forcément le milieu du polynôme : n doit être le même pour P et Q)

```
let cut l n =
  let rec term_rec_cut acc l n =
    match n, l with
      | 0, _ → rev acc, l
      | _, [ ] → rev acc, [ ]
      | n, t :: q → term_rec_cut (t :: acc) q (n − 1)
  in
    term_rec_cut [ ] l n;;
```

puis la multiplication proprement dite avec les appels récursifs et leur combinaison

```
let rec split_mult p q =
  match p,q with
    | [ ],_ | _,[ ] | [0],_ | _,[0] → zero
    | [a ],_ → map (prefix ∗ a) q
    | _,[b ] → map (prefix ∗ b) p
    | _ →
        let n = (1 + max (deg p) (deg q)) / 2 in
        let p_1, p_2 = cut p n in
        let q_1, q_2 = cut q n in
        let pq_1 = split_mult p_1 q_1 in
        let pq_3 = split_mult p_2 q_2 in
        let pq_2 = add ( split_mult p_1 q_2) ( split_mult p_2 q_1) in
          add (add pq_1 (add_zeros pq_2 n)) (add_zeros pq_3 (2 ∗ n ));;
```

La relation de récurrence qui exprime la complexité de cet algorithme est $C(n) = 4C(n/2) + O(n)$ et elle se résout en $C(n) = O(n^2)$. Voir la question 7 pour une méthode de résolution.

▸ **Question 6**

```
let rec karatsuba p q =
  match p,q with
    | [ ],_ | _,[ ] | [0],_ | _,[0] → zero
    | [a ],_ → map (prefix ∗ a) q
    | _,[b ] → map (prefix ∗ b) p
    | _ →
        let n = (1 + max (deg p) (deg q)) / 2 in
        let p_1, p_2 = cut p n in
        let q_1, q_2 = cut q n in
        let pq_1 = karatsuba p_1 q_1 in
        let pq_3 = karatsuba p_2 q_2 in
        let pq_2 =
          subtract (karatsuba (add p_1 p_2) (add q_1 q_2)) (add pq_1 pq_3) in
          add (add pq_1 (add_zeros pq_2 n)) (add_zeros pq_3 (2 ∗ n ));;
```

► **Question 7** Notons $C(n)$ la multiplication entre polynômes de degrés strictement inférieurs à n. En plus des trois appels récursifs, il y a des opérations linéaires : deux calculs de degrés, deux découpes en $n/2$ puis des additions : deux de taille $n/2$, une de taille n, une de taille $3n/2$ et une de taille $2n$. On obtient donc la relation de récurrence suivante :

$$C(n) = 3 \cdot C(n/2) + 15n/2$$

Une méthode de résolution est de poser $\alpha_l = \frac{C(2^l)}{3^l}$ qui vérifie $\alpha_l = \alpha_{l-1} + \frac{15}{2}\left(\frac{2}{3}\right)^l$. D'où on tire, puisque $\alpha_0 = C(1) = 1$,

$$\alpha_l = \frac{15}{2}\sum_{k=1}^{l}\left(\frac{2}{3}\right)^k + \alpha_0 = 3\frac{15}{2}\left(1 - \left(\frac{2}{3}\right)^{l+1}\right) \qquad \text{puis} \qquad C(2^l) = \frac{15}{2}(3^{l+1} - 2^{l+1}) = O(3^l) = O(2^{l\frac{\ln 3}{\ln 2}}) = O(n^{\frac{\ln 3}{\ln 2}})$$

La complexité de la multiplication de Karatsuba est donc $O(n^{\frac{\ln 3}{\ln 2}}) \approx O(n^{1.585})$.

► **Question 8** Pour utiliser les opérateurs de façon infixe, on utilise la construction **let prefix**. Dans ce cas, l'opérateur ainsi défini ne peut pas contenir de lettres ou de chiffres, seulement des symboles. (et encore, pas tous !)

```
let conj a = { Re = a.Re ; Im = -. a.Im };;
let prefix +: a b = { Re = a.Re +. b.Re ; Im = a.Im +. b.Im };;
let prefix -: a b = { Re = a.Re -. b.Re ; Im = a.Im -. b.Im };;
let prefix *: a b = { Re = (a.Re *. b.Re) -. (a.Im *. b.Im) ;
                      Im = (a.Re *. b.Im) +. (a.Im *. b.Re) };;
let prefix /: a b =
  let norm2_b = (b.Re *. b.Re) +. (b.Im *. b.Im) in
    { Re = ((a.Re *. b.Re) +. (a.Im *. b.Im)) /. norm2_b ;
      Im = ((a.Im *. b.Re) -. (a.Re *. b.Im)) /. norm2_b };;

let zero = { Re = 0. ; Im = 0. };;
let one = { Re = 1. ; Im = 0. };;
```

En ce qui concerne les fonctions de conversion, si passer des entiers aux complexes ne pose pas de problèmes, le contraire n'est pas vrai : il faut faire attention aux erreurs d'arrondi. Comme la fonction Caml de conversion des flottants en entiers réalise une troncature, la technique usuelle consiste à ajouter 0.5 avant la troncature. Mais dans les nombres négatifs, cette troncature fonctionne dans l'autre sens et il faut donc retrancher 0.5 et non l'ajouter.

```
let complex_of_int n = { Re = float_of_int n ; Im = 0. };;
let int_of_complex c =
  int_of_float (if c.Re >= 0. then (c.Re +. 0.5) else (c.Re -. 0.5));;
```

► **Question 9** Pour le calcul des puissances, on utilise l'exponentiation rapide qui réduit la complexité de $O(n^2)$ à $O(n \log n)$, un gain appréciable avant d'avoir la solution linéaire de la question suivante.

```
let rec fast_exp c n =
  if n = 0 then one
  else
    if n mod 2 = 0
    then fast_exp (c *: c) (n / 2)
    else c *: (fast_exp (c *: c) (n / 2));;

let eval p c =
  let rec eval_aux acc k c = function
    | [] -> acc
    | t :: q -> eval_aux ((t *: (fast_exp c k)) +: acc) (k + 1) c q
  in
    eval_aux zero 0 c p;;
```

► **Question 10** Il n'est pas possible de faire mieux que $\Theta(n)$ opérations car il faut lire tous les coefficients du polynôme.

```
let horner p c =
  let rev_p = rev p in
    it_list (fun val a_i → (val *: c) +: a_i) zero rev_p;;
```

▸ **Question 11**

```
let rec split p =
  list_it (fun elt (l1, l2) → (elt :: l2, l1)) p ([], []);;
```

Pour d'autres versions possibles et des détails sur celle-là, voir le TP n° 2 sur les circuits. Noter cependant la différence entre it_list et list_it : it_list commence par le début de la liste et est terminale récursive alors que list_it ne l'est pas et commence par la fin de la liste.

▸ **Question 12** C'est dans cette question que ce justifie la contrainte sur n d'être une puissance de 2. En effet, pour avoir $\omega_n^{k+\frac{n}{2}} = -\omega_n^k$, il faut que n soit pair. Comme on veut appliquer l'algorithme récursivement sur $n/2$, il faut que n soit une puissance de 2.

```
let combine omegas eval_even eval_odd =
  let half_omegas = fst (cut omegas (list_length omegas / 2)) in (* w^2j *)
  let odd_eval = map2 (prefix *:) eval_odd half_omegas in (* w^j * P_i(w^2j) *)
  let fst_list = map2 (prefix +:) eval_even odd_eval in (* P_p + w^j * P_i *)
  let snd_list = map2 (prefix -:) eval_even odd_eval in (* P_p - w^j * P_i *)
    fst_list @ snd_list ;;
```

▸ **Question 13**

```
let rec eval_constant p n =
  match p with
    | [a] → if n = 1 then p else a :: eval_constant p (n - 1)
    | _ → failwith "eval_constant : invalid polynomial";;

let rec fft n omegas = function
  | [] → failwith "fft : empty polynomial"
  | [a] as p → eval_constant p n
  | p →
      let p_even, p_odd = split p in
      let squared_omegas = fst (split omegas) in
      let fft_even = fft (n / 2) squared_omegas p_even in
      let fft_odd = fft (n / 2) squared_omegas p_odd in
        combine omegas fft_even fft_odd ;;
```

▸ **Question 14**

```
let compute_powers omega n =
  let rec term_rec_aux w w_k n acc =
    if n = 0 then acc else
      let w_k1 = w *: w_k in
        term_rec_aux w w_k1 (n - 1) (w_k1 :: acc)
  in
    rev (term_rec_aux omega one (n - 1) [one]);;
```

▸ **Question 15** On peut voir l'évaluation comme la multiplication d'une matrice de Vandermonde et du vecteur colonne des coefficients. Ainsi, l'interpolation est simplement la multiplication du vecteur des évaluations par la matrice inverse.

$$\begin{pmatrix} 1 & & \dots & 1 \\ 1 & \omega_n & \dots & \omega_n^{n-1} \\ \vdots & & & \vdots \\ 1 & \omega_n^{n-1} & \dots & \omega_n^{(n-1)^2} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} \widehat{a_0} \\ \widehat{a_1} \\ \vdots \\ \widehat{a_{n-1}} \end{pmatrix} \iff \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & \dots & 1 \\ 1 & \omega_n & \dots & \omega_n^{n-1} \\ \vdots & \vdots & & \vdots \\ 1 & \omega_n^{n-1} & \dots & \omega_n^{(n-1)^2} \end{pmatrix}^{-1} \begin{pmatrix} \widehat{a_0} \\ \widehat{a_1} \\ \vdots \\ \widehat{a_{n-1}} \end{pmatrix}$$

▸ **Question 16** On note $P = \sum_{i=0}^{n-1} p_i X^i$ polynôme de degré au plus $n-1$. La première évaluation fournit la liste des $P(\omega_n^j) = \sum_{i=0}^{n-1} p_i \omega_n^{ij}$ donc le polynôme

$$P_2 = \sum_{k=0}^{n-1} P(\omega_n^k) \cdot X^k = \sum_{k=0}^{n-1} \left(\sum_{i=0}^{n-1} p_i \omega_n^{ik} \right) X^k$$

En l'évaluant en ω_n^{-j}, on trouve

$$P_2(\omega_n^{-j}) = \sum_{k=0}^{n-1} \sum_{i=0}^{n-1} p_i \omega_n^{ik} \omega_n^{-jk} = \sum_{k=0}^{n-1} \sum_{i=0}^{n-1} p_i \omega_n^{k(i-j)} = \sum_{i=0}^{n-1} p_i \sum_{k=0}^{n-1} \omega_n^{k(i-j)} = np_j + \sum_{i \neq j} p_i \underbrace{\sum_{k=0}^{n-1} \omega_n^{k(i-j)}}_{=0} = np_j$$

car pour $j \neq 0$, la somme des puissances j^{e} des racines n^{e} de l'unité est nulle : $\sum_{\omega^n=1} \omega^j = \sum_{k=0}^{n-1} \omega_n^{kj} = 0$. Cela peut se voir par exemple avec les relations coefficients/racines pour le polynôme $X^n - 1$ ou plus simplement comme la somme des termes d'une suite géométrique (ω_n^j est une racine n^{e} de l'unité) :

$$\text{pour } \omega_n^j \neq 1 (\textit{ i.e. } 1 \leqslant j \leqslant n-1), \quad \sum_{k=0}^{n-1} \omega_n^{kj} = \frac{1 - (\omega_n^j)^n}{1 - \omega_n^j} = 0$$

▸ **Question 17** D'après la question précédente, évaluer en les ω_n^k puis en les ω_n^{-k} revient à multiplier par n les coefficients. L'interpolation en les ω_n^k est donc simplement une évaluation en les ω_n^{-k} suivie d'une division par n. On peut résumer cela matriciellement :

$$\begin{pmatrix} 1 & 1 & \dots & 1 \\ 1 & \omega_n & \dots & \omega_n^{n-1} \\ \vdots & \vdots & & \vdots \\ 1 & \omega_n^{n-1} & \dots & \omega_n^{(n-1)^2} \end{pmatrix}^{-1} = \frac{1}{n} \begin{pmatrix} 1 & 1 & \dots & 1 \\ 1 & \omega_n^{-1} & \dots & \omega_n^{1-n} \\ \vdots & \vdots & & \vdots \\ 1 & \omega_n^{1-n} & \dots & \omega_n^{(1-n)^2} \end{pmatrix}$$

```
let ifft n omegas p =
  let cmplx_n = complex_of_int n in
    map (fun c → c /: cmplx_n) ( fft n (one :: (rev ( tl omegas))) p );;
```

▸ **Question 18**

```
let fft_mult p q =
  let log_n = (log ( float_of_int (deg p + deg q + 1))) /. log 2. in
  let n = 1 lsl ( int_of_float ( ceil log_n)) in
  let pi_twice = 2. ∗. acos (−. 1.) in
  let w = { Re = cos (pi_twice /. float_of_int n) ;
            Im = sin (pi_twice /. float_of_int n) } in
  let omegas = compute_powers w n in
  let cmplx_p = map complex_of_int p in
  let cmplx_q = map complex_of_int q in
  let eval_p = fft n omegas cmplx_p in
  let eval_q = fft n omegas cmplx_q in
  let eval_pq = map2 (prefix ∗:) eval_p eval_q in
  let cmplx_pq = ifft n omegas eval_pq in
    normalize (map int_of_complex cmplx_pq);;
```

▸ **Question 19** De par la nature récursive de l'algorithme, on trouve la relation de récurrence suivante pour la complexité de l'évaluation par transformée de Fourier :

$$\begin{cases} C(n) = 2C(n/2) + \frac{3}{2}n \\ C(1) = 1 \end{cases}$$

qui se résout (voir la question 7) en $C(n) = \frac{3}{2} n \log n$. (on rappelle que n doit être une puissance de deux)

L'algorithme ne comportant en outre que des opérations linéaires (calcul des ω_n^k, multiplication composante par composante, divisions par n), la complexité globale de la multiplication de Schönhage-Strassen est $O(n \log n)$.