

Lecture #6 (Continued):

Sources: D.E. Knuth Vol #2; Books on Algebra; CLR, KT

0.0.1 Polynomials, Convolutions, Fourier Transforms

Let a_0, a_1, \dots, a_{n-1} be elements of an algebraic system S and let the variable x be regarded as an indeterminate (not specified but also comes from S). An expression of the form $A(x) = a_0 + a_1x + a_2x^2 + \dots + a_{n-1}x^{n-1}$ is called a *polynomial function* of x . It is usual to assume that S is a commutative ring with addition, and multiplication operations defined in the usual manner. Addition and multiplication are associative, and commutative, and multiplication is distributive over addition. There is an identity element for addition (we usually denote it by 0) and one for multiplication (we usually denote it by 1). There is additive inverse for each element and subtraction is the inverse of addition. When we consider division, we will assume that S is a field. The polynomial $A'(x) = a_0 + a_1x + a_2x^2 + \dots + a_{n-1}x^{n-1} + 0x^n + \dots + 0x^{n+m}$ is considered the same as the polynomial $A(x)$ [although formally it is a different expression.]. If all coefficients are equal to 0 the polynomial itself is referred to as the *zero polynomial* $0(x)$. The vector $a = [a_0, a_1, \dots, a_{n-1}]$ is called the vector of *coefficients* and the *degree* of the polynomial $A(x)$ is given by $\deg(A(x)) = \max\{j : a_j \neq 0\}$. An upper bound on $\deg(A(x))$ is called a *degree bound* for $A(x)$. If $A(x)$ and $B(x)$ are two polynomial functions of x each with degree bound n , then their *sum* is the polynomial function $C(x) = A(x) + B(x)$.

There are two well known ways of representing a polynomial function:

- (a) To specify $A(x)$ we specify the vector $a = [a_0, a_1, \dots, a_{n-1}]$ of its coefficients. This representation is called the *coefficient form* of representation.
- (b) Here we specify n point-value pairs; $\{(x_i, y_i) : 1 \leq i \leq n\}$ such that $[i \neq j \Rightarrow x_i \neq x_j]$ and $[y_i = A(x_i) : 1 \leq i \leq n]$. This representation is called the *point-value form* of representation.

There are several operations done on polynomials: (i) Evaluate $A(x)$ at x_0 ; (ii) Add two (or more) polynomials; (iii) Multiply two polynomials; (iv) Divide one polynomial by another; (v) find the greatest common divisor of two polynomials; (vi) factor a polynomial etc. These are similar to those for numbers. The way these are done under different representations is different and entails different amounts of work.

0.0.2 Coefficient Form

Evaluation:

1. Given a polynomial $A(x)$ in coefficient form there is nice method called Horner's Rule that is used to evaluate a polynomial in $O(n)$ where n is $\deg A(x)$. Here is a pseudo-code for Horner's Rule:

HORNER'S RULE

Input: $a = [a_0, a_1, \dots, a_{n-1}] \in R, x_0 \in R$ // Here R is a ring and $A(x) = \sum_{i=0}^{n-1} a_i x^i$

OUTPUT: $A(x_0)$

2. $y \leftarrow 0$
3. $i \leftarrow n$
4. **while** $i \geq 0$ **do**
5. $y \leftarrow a_i + x_0 y$
6. $i \leftarrow i - 1$
7. **return** y

It is easy to see that the complexity of this algorithm is $\Theta(n)$. That the result is correct is left to you to show by using loop invariants or induction.

Addition:

Given two polynomials $A(x)$ [by specifying vector a] and $B(x)$ [by specifying vector b] with same degree bound, computing $C(x) = A(x) + B(x)$ is easy since this is represented by the vector $c = a + b$. This also has a complexity $\Theta(n)$ assuming all additions take a single step.

Multiplication:

Given the same pair of polynomials in coefficient form (with same degree bound), multiplying them is a bit more difficult. If

$$C(x) = A(x)B(x)$$

then $\deg C(x) = \deg A(x) + \deg B(x)$. There are $2n - 1$ coefficients for $C(x)$ if there are n for each of $A(x)$ and $B(x)$. They are given by the formula:

$$c_k = \sum_{\substack{j,l: \\ j+l=k \\ j < n \\ l < n}} a_j b_l; k = 0, 1, \dots, 2n - 1$$

The vector c is called the *convolution* of the vectors a and b and this is often denoted by the equation

$$c = a * b$$

Since we must compute $O(n)$ such coefficients and each computation has complexity $O(n)$ the overall work is $O(n^2)$ if we do it this way. We can reduce this by using divide-and-conquer method to $O(n^{\lg 3})$ if we break the polynomial into two roughly equal parts at each stage (till the degree is very small). If we break it into k parts we can get an algorithm whose worst case complexity is reduced further to $O(n^{\log_k(2k-1)})$ and this as k tends to ∞ has a form $O(n^{1+\epsilon})$. [See Lecture notes #4 on fast integer multiplication.] Please note that if the

input data comes from an algebraic system S , we preserve this throughout this algorithm (even if it is a ring). Later, allowing the use of complex numbers, we can reduce the complexity to $O(n \lg n)$. This leads us to the closely related topic Discrete (or Finite) Fourier Transforms and the algorithm Fast Fourier Transform to calculate it.

Now let us also consider division.

Division:

Let S be such that if $\{u \in S, v \in S, v \neq 0\} \Rightarrow \exists w \in S$ such that $u = vw$. The most usual cases of S are: (i) \mathcal{Q} ; (ii) \mathcal{R} or \mathcal{C} ; (iii) Rational functions (ratio of polynomials); (iv) integers modulo p where p is a prime.

Lemma 1 *Given two polynomials $A(x)$ and $B(x) \neq 0(x)$ over a field S , there is a unique pair of polynomials $q(x)$ and $r(x)$ over S such that the equation*

$$A(x) = q(x)B(x) + r(x)$$

holds where $\deg r(x) < \deg B(x)$.

Proof. *Suppose*

$$A(x) = q_1(x)B(x) + r_1(x) = q_2(x)B(x) + r_2(x)$$

Then

$$[q_1(x) - q_2(x)]B(x) = r_2(x) - r_1(x)$$

which in turn implies that

$$\deg[q_1(x) - q_2(x)]B(x) = \deg[q_1(x) - q_2(x)] + \deg B(x) \geq \deg B(x) > \deg[r_2(x) - r_1(x)]$$

if $q_1(x) - q_2(x) \neq 0(x)$ and this is a contradiction. Hence, it follows that there is at most one such pair. To show existence of a pair, we use the usual algorithm.

■

The algorithm for polynomial division, has as an input two polynomials $A(x)$ and $B(x) \neq 0(x)$ with $\deg A(x) \geq \deg B(x)$ [If the last condition is not true, then $q(x) = 0(x)$ and $r(x) = A(x)$]. Its output are two polynomials $q(x)$ and $r(x)$ satisfying the relation

$$A(x) = q(x)B(x) + r(x)$$

with $\deg r(x) < \deg B(x)$. This is from D. Knuth vol 2.

INPUT: $A(x) = a_0 + a_1x + \dots + a_mx^m; B(x) = b_0 + b_1x + \dots + b_nx^n; m \geq n \geq 0; b_n \neq 0; a_i, b_j \in S$

OUTPUT: $q(x) = q_0 + q_1x + \dots + q_{m-n}x^{m-n}; r(x) = r_0 + r_1x + \dots + r_{n-1}x^{n-1}; q_k, r_l \in S$

Algorithm 2 *POLYNOMIAL-DIVISION* ($[a_0, a_1, \dots, a_m], [b_0, b_1, \dots, b_n]; m \geq n > 0; b_n \neq 0$

1. $k \leftarrow m - n$

2. **while** $k > -1$ **do**
3. $q_k \leftarrow \frac{a_{n+k}}{b_n}$
4. **for** $j \leftarrow n + k - 1$ **to** k **do**
5. $a_j \leftarrow a_j - q_k b_{j-k}$ // this replaces $A(x)$ by $A(x) - q_k x^k B(x)$, a polynomial of degree $< n + k$
6. $r \leftarrow [a_0, a_1, \dots, a_{n-1}]$
7. **return** q, r

Complexity of this algorithm is $\Theta(n(m - n + 1))$. The only explicit division is in step 3 and the divisor is always b_n . Hence if $b_n = 1$, we do not need S to be a field.

0.0.3 Point-Value Form

Addition:

Given two polynomials $A(x) = \{(x_i, y_i); 1 \leq i \leq n\}$ and $B(x) = \{(x_i, z_i); 1 \leq i \leq n\}$ with the same degree bound n , in point-value form, the point-value form of $C(x) = A(x) + B(x)$ is obtained as $\{(x_i, y_i + z_i); 1 \leq i \leq n\}$. Hence complexity of addition in this case is also $\Theta(n)$. [**Note that the points at which the two polynomials are evaluated are the same. This is also true for multiplication.**]

Multiplication:

Given two polynomials $A(x) = \{(x_i, y_i); 1 \leq i \leq 2n\}$ and $B(x) = \{(x_i, z_i); 1 \leq i \leq 2n\}$ with the same degree bound n , in point-value form, the point-value form of $C(x) = A(x) \bullet B(x)$ is obtained as $\{(x_i, y_i \bullet z_i); 1 \leq i \leq 2n\}$. Hence complexity of multiplication in this case is $\Theta(n)$. [**Note that each polynomial is evaluated at as many points as are required for the representation of the product.**]

Evaluation:

Given a polynomial in point-value form, in order to evaluate the polynomial at another point is a bit more time consuming. The simplest method seems to be to convert to coefficient form and then evaluate.

0.0.4 Transformations between the two forms

Given a polynomial in one form we may want to transform it to the other form. This is useful since some operations are easier in one than the other. We consider this now. We also need to make sure that there is a unique representation in each corresponding to representation in the other.

By using Horner's Rule, we can compute point-value representation of a polynomial given in coefficient form in $O(n^2)$ time – since we need to evaluate at n points and each takes $\Theta(n)$ time this follows. How about the converse? **Please note that we may choose any set of points as long as they are distinct for this purpose.**

Theorem 3 For any set of n point-value pairs $\{(x_i, y_i); 0 \leq i \leq n-1\}$ such that $\{[i \neq j; 0 \leq i, j \leq n-1] \Rightarrow [x_i \neq x_j]\}$, there is a unique polynomial $A(x)$ of degree bound $n-1$ with $y_i = A(x_i)$ for $0 \leq i \leq n-1$.

Proof. The set of equations $y_i = A(x_i)$ for $0 \leq i \leq n-1$ [in the coefficients $[a_0, a_1, \dots, a_{n-1}]$] is given by:

$$\begin{array}{|c|c|c|c|c|} \hline 1 & x_0 & x_0^2 & \bullet & x_0^{n-1} \\ \hline 1 & x_1 & x_1^2 & \bullet & x_1^{n-1} \\ \hline \bullet & \bullet & \bullet & \bullet & \bullet \\ \hline \bullet & \bullet & \bullet & \bullet & \bullet \\ \hline 1 & x_{n-1} & x_{n-1}^2 & \bullet & x_{n-1}^{n-1} \\ \hline \end{array} \times \begin{array}{|c|} \hline a_0 \\ \hline a_1 \\ \hline \bullet \\ \hline \bullet \\ \hline a_{n-1} \\ \hline \end{array} = \begin{array}{|c|} \hline y_0 \\ \hline y_1 \\ \hline \bullet \\ \hline \bullet \\ \hline y_{n-1} \\ \hline \end{array}$$

The matrix has a determinant equal to $\prod_{0 \leq i, j \leq n-1; i \neq j} [x_i - x_j]$ and since the x_i are distinct, this determinant is nonzero. (Show this) Hence the equations have a unique solution. The complexity of solving these equations (or equivalently inverting the matrix) is $O(n^3)$ using normal algorithms for this purpose. But there is another method whose complexity is $O(n^2)$ and it is described below. ■

Theorem 4 LaGrange: For any set of n point-value pairs $\{(x_i, y_i); 0 \leq i \leq n-1\}$ such that $\{[i \neq j; 0 \leq i, j \leq n-1] \Rightarrow [x_i \neq x_j]\}$, the corresponding polynomial $A(x)$ is given by

$$A(x) = \sum_{k=0}^{n-1} y_k \frac{\prod_{j \neq k} (x - x_j)}{\prod_{j \neq k} (x_k - x_j)}$$

Proof. Verify that $A(x_k) = y_k; 0 \leq k \leq n-1$ Now from uniqueness theorem above, the result follows. ■

The complexity of doing it this way is $O(n^2)$ if it is done as per exercise 30.1-5. This is described in what follows:

Exercise 30.1-2: Given $A(x)$, a polynomial of degree bound n , and a value x_0 , one other method (other than Horner's Rule) to compute $A(x_0)$ is to divide $A(x)$ by $(x - x_0)$ and get a quotient $q(x)$ and a remainder r . The degree bound on $q(x)$ is $n-1$ and r is constant. Thus

$$A(x) = q(x) \bullet (x - x_0) + r$$

Hence $A(x_0) = r$. Here it is assumed that $A(x)$ is given in coefficient form. Complexity of this process is also $O(n)$ since the polynomial is being divided by a polynomial whose degree is 1 and whose leading coefficient is also 1.

Computation of $\prod_{j \neq k} (x - x_j)$:

First we compute $\prod_{j=0}^{n-1} (x - x_j)$ by recursively computing $\prod_{j=n-1}^p (x - x_j)$ for $p = n-1, \dots, 0$. Suppose

$$\prod_{j=n-1}^p (x - x_j) = a_0 + a_1 x + \dots + a_{n-p} x^{n-p}$$

and we want

$$\begin{aligned} \prod_{j=n-1}^{p-1} (x - x_j) &= (x - x_{p-1}) \prod_{j=n-1}^p (x - x_j) \\ &= (x - x_p) [a_0 + a_1 x + \dots + a_{n-p} x^{n-p}] \\ &= -a_0 x_p + (a_0 - a_1 x_p) x + (a_1 - a_2 x_p) x^2 + \dots + (a_{n-p-1} - a_{n-p} x_p) x^{n-p} + a_{n-p} x^{n-p+1} \end{aligned}$$

It should be clear that to calculate the coefficients given $[a_0, a_1, \dots, a_{n-p}]$, the complexity is $\Theta(n)$ and hence to compute $\prod_{j=0}^{n-1} (x - x_j)$ has a complexity $\Theta(n^2)$. To compute $\prod_{j \neq k} (x - x_j)$ we divide $\prod_{j=0}^{n-1} (x - x_j)$ by $(x - x_k)$ and this takes for each k time complexity of $\Theta(n)$. So to compute all of them is $\Theta(n^2)$. Now to compute $\prod_{j \neq k} (x_k - x_j)$ we evaluate $\prod_{j \neq k} (x - x_j)$ for $x = x_k$ and the complexity of this operation is $O(n)$ for each value of k . Hence the whole process can be done in $O(n^2)$ time.

Advantages of point-value representation: (i) Addition is $O(n)$; (ii) Multiplication is also $O(n)$: but be careful to have enough point-value pairs for each so that we get as many pairs for the product as we need. We want the advantages of both representations by using the transformation between the two. If we are given point-value form to do an evaluation, convert to coefficient form and then evaluate the polynomial. If we want to multiply and the polynomials are given in coefficient form, convert to point-value form and then multiply. This can only work if these conversions have complexity $o(n^2)$. This leads us to Fourier transforms.

Complex Roots of Unity Consider the equation:

$$A(x) = 0$$

where $A(x)$ is a polynomial in x whose coefficients are complex numbers (numbers of the form $(a + bi)$ where $i = \sqrt{-1}$ with $a, b \in R$). There is a theorem, often attributed to C.F. Gauss, known as **the fundamental theorem of algebra**, that states there are n roots (solutions) to this equation each of which is a complex number if the degree of the polynomial is n . [What is the fundamental theorem of arithmetic?] The solutions to the special equation of this form

$$x^n = 1$$

are called n^{th} -**roots of unity**. Although this equation has all its coefficients as integers, some of its n roots may not be real numbers. An n^{th} root of unity z is **primitive** if

$$z^k \neq 1 \text{ for } k = 1, 2, \dots, n-1$$

If z is a primitive n^{th} -root of unity, then

$$\{z^k : k = 1, 2, \dots, n\}$$

are the n n^{th} - roots of unity. The primitive n^{th} -root of unity $e^{\frac{2\pi i}{n}}$ ($= \omega_n$) is called a **principal** n^{th} - root of unity. Recall

$$e^{ix} = \cos(x) + i \sin(x)$$

and hence $e^{\pi i} + 1 = \cos(\pi) + i \sin(\pi) + 1 = 0$ [The story goes that Euler used this equation to claim that God exists since the five most famous constants $0, 1, i, \pi, e$ are so connected by this beautiful formula and this could only happen ...]. It implies that $e^{2\pi i} = 1$. Thus, n complex n^{th} -roots of unity are $\omega_n^0, \omega_n^1, \dots, \omega_n^{n-1}$. Now some properties of these roots:

1. $\omega_n^{dk} = \omega_n^k$ for any integers $n \geq 0, k \geq 0$, and $d > 0$.
2. $\omega_n^{\frac{n}{2}} = \omega_2 = -1$; here we assume that n is an even integer
3. If $n > 0$ is an even integer, then the squares of the n complex n^{th} -roots of unity are the $\frac{n}{2}$ complex $(\frac{n}{2})^{\text{th}}$ -roots of unity. This is called Halving Lemma in your book. [See page 832 of your book]

Proof: By result #1 above, $(\omega_n^k)^2 = \omega_n^{2k}$ for any nonnegative integer k . If we square all complex n^{th} -roots of unity, then each $(\frac{n}{2})^{\text{th}}$ -root of unity is obtained exactly twice, since

$$(\omega_n^{k+\frac{n}{2}})^2 = (\omega_n^k)^2$$

since $\omega_n^n = 1$. Indeed, $\omega_n^{k+\frac{n}{2}} = \omega_n^k \omega_n^{\frac{n}{2}} = -\omega_n^k$ using #2. [This result is most important to obtaining the divide-and-conquer algorithm with complexity $O(n \lg n)$.]

4. For any integer $n \geq 1$, nonzero integer k not divisible by n ,

$$\begin{aligned} \sum_{j=0}^{n-1} (\omega_n^k)^j &= \frac{(\omega_n^k)^n - 1}{(\omega_n^k) - 1} \\ &= \frac{(\omega_n^n)^k - 1}{(\omega_n^k) - 1} \\ &= \frac{1 - 1}{(\omega_n^k) - 1} \\ &= 0 \end{aligned}$$

since $(\omega_n^k) \neq 1$ if k is not divisible by n .

0.0.5 DFT (Discrete Fourier Transform)

Let $[x_0, x_1, \dots, x_{n-1}] = x$ be a finite sequence of complex numbers. The $\text{DFT}_n(x)$ is defined by another n -vector $[y_0, y_1, \dots, y_{n-1}] = y$ as follows:

$$y_k = \sum_{j=0}^{n-1} x_j \omega_n^{kj}; 0 \leq k \leq n-1$$

where $e^{\frac{2\pi i}{n}} = \omega_n$ is a principal n^{th} -root of unity. This transformation is denoted by

$$y = \text{DFT}_n(x)$$

The inverse of this transformation recovers x from y and is given by the formula:

$$x_k = \frac{1}{n} \sum_{j=0}^{n-1} y_j \omega_n^{-kj}; 0 \leq k \leq n-1 \quad (1)$$

We think of x as in time domain and y as in frequency domain.

Properties of this transformation:

1. It is an invertible linear transformation.

$$y = Fx$$

where the $(i, j)^{th}$ element $F_{i,j}$ of the $n \times n$ matrix F is given by

$$F_{i,j} = \omega_n^{ij}; 0 \leq i, j \leq n-1$$

and its inverse is given by

$$F^{-1} = \frac{1}{n} F^*$$

where F^* is the conjugate of F . If we let $U = \frac{1}{\sqrt{n}} F$, then $U^{-1} = U^*$ and $|\det(U)| = 1$.

- 2.

$$x = F^{-1}y$$

0.0.6 FFT (Fast Fourier Transform Algorithm)

This explains how to compute $\text{DFT}_n(x)$ by an algorithm whose worst case complexity is $O(n \lg n)$. Please note that $\text{DFT}_n(x)$ is also a set of n polynomials in ω_n and hence can be evaluated in $O(n^2)$ time using exact arithmetic. Let us think of evaluating polynomial $A(x)$ given by its coefficients; $[a_0, a_1, \dots, a_{n-1}]$. We divide this polynomial into two parts (but not the usual way) – in this section we assume that n is a power of 2.

$$\begin{aligned} A^{[E]}(x) &= a_0 + a_2x + a_4x^2 + \dots + a_{n-2}x^{\frac{n}{2}-1} \\ A^{[O]}(x) &= a_1 + a_3x + a_5x^2 + \dots + a_{n-1}x^{\frac{n}{2}-1} \\ A(x) &= A^{[E]}(x^2) + xA^{[O]}(x^2) \end{aligned}$$

This is odd-even splitting. Thus, the problem of evaluating $A(x)$ at $\omega_n^0, \omega_n^1, \dots, \omega_n^{n-1}$ reduces to:

- (a) Evaluating $A^{[E]}(x)$ and $A^{[O]}(x)$ at $(\omega_n^0)^2, (\omega_n^1)^2, \dots, (\omega_n^{n-1})^2$. But these are polynomials whose degree bound is $\frac{n}{2}$. Moreover, among the set of values $(\omega_n^0)^2, (\omega_n^1)^2, \dots, (\omega_n^{n-1})^2$ there are at most $\frac{n}{2}$ distinct values and so this problem is similar to the original problem but of half size.

(b) Combine the results of #1 using the formula:

$$A(x) = A^{[E]}(x^2) + xA^{[O]}(x^2)$$

This involves $\Theta(1)$ time per each value of x and hence is $\Theta(n)$ overall.

Here is an algorithm from your book (page 835):

RECURSIVE-FFT(a)

1. $n \leftarrow \text{length}[a] // a = [a_0, a_1, \dots, a_{n-1}]$
2. **if** $n = 1$
3. **then return** a
4. $\omega_n \leftarrow e^{\frac{2\pi i}{n}}$
5. $\omega \leftarrow 1$
6. $a^{[0]} \leftarrow [a_0, a_2, \dots, a_{n-2}]$
7. $a^{[1]} \leftarrow [a_1, a_3, \dots, a_{n-1}]$
8. $y^{[0]} \leftarrow \text{RECURSIVE-FFT}(a^{[0]})$
9. $y^{[1]} \leftarrow \text{RECURSIVE-FFT}(a^{[1]})$
10. **for** $k \leftarrow 0$ **to** $\frac{n}{2} - 1$
11. **do** $y_k \leftarrow y_k^{[0]} + \omega y_k^{[1]}$
12. $y_{k+\frac{n}{2}} \leftarrow y_k^{[0]} - \omega y_k^{[1]}$
13. $\omega \leftarrow \omega \omega_n$
14. **return** y

Line-by-Line description:

Lines 2-3: Basis Case: $y_0 = a_0 \omega_1^0 = a_0$; since in this case ($n = 1$) DFT is the element itself.

Lines 6-7: define coefficient vectors for $A^{[0]}$ and $A^{[1]}$.

Lines 4,5, and 13: Guarantee that the value of ω is updated from iteration to iteration (as k changes) properly so that when we come to lines 11-12, $\omega = \omega_n^k$.

Lines 8-9: Compute DFT $_{\frac{n}{2}}$; for $0 \leq k \leq \frac{n}{2} - 1$, by these lines we have

$$\begin{aligned} y_k^{[0]} &= A^{[0]}(\omega_{\frac{n}{2}}^k) = A^{[0]}(\omega_n^{2k}) \\ y_k^{[1]} &= A^{[1]}(\omega_{\frac{n}{2}}^k) = A^{[1]}(\omega_n^{2k}) \end{aligned}$$

Lines 11-12: Combine the results of $\text{DFT}_{\frac{n}{2}}$ calculations to yield for $0 \leq k \leq \frac{n}{2} - 1$

$$\begin{aligned} y_k &= y_k^{[0]} + \omega_n^k y_k^{[1]} \\ &= A^{[0]}(\omega_n^{2k}) + \omega_n^k A^{[1]}(\omega_n^{2k}) \\ &= A(\omega_n^k) \end{aligned}$$

The last of these follows from

$$A(x) = A^{[E]}(x^2) + xA^{[O]}(x^2)$$

Also, we have:

$$\begin{aligned} y_{\frac{n}{2}+k} &= y_k^{[0]} - \omega_n^k y_k^{[1]} \\ &= A^{[0]}(\omega_n^{2k}) + \omega_n^{k+\frac{n}{2}} A^{[1]}(\omega_n^{2k}) \\ &= A^{[0]}(\omega_n^{2k+n}) + \omega_n^{k+\frac{n}{2}} A^{[1]}(\omega_n^{2k+n}) \\ &= A(\omega_n^{k+\frac{n}{2}}) \end{aligned}$$

Thus the vector y returned by algorithm is correct.

Complexity:

The recurrence relation for this algorithm:

$$t(n) = 2t\left(\frac{n}{2}\right) + cn$$

Since the only loop in the algorithm is $O(n)$, additional work beyond recursive calls is $O(n)$. Thus, given a polynomial in coefficient form, we can (for suitably selected values for $\{x_i : 0 \leq i \leq n-1\}$), we can compute point-value form in $O(n \lg n)$ time using FFT. **Please note much more complex operations are being performed at each step if your original data are, say, integers.**

Now we turn to conversion of point-value form to coefficient form in $O(n \lg n)$ time. So suppose we are given $[y_0, y_1, \dots, y_{n-1}]$ and we want the coefficient form $[a_0, a_1, \dots, a_{n-1}]$. We know that these satisfy the equations:

1	1	1	•••	1	×	a_0	=	y_0
1	ω_n	ω_n^2	•••	ω_n^{n-1}		a_1		y_1
•	•	•	•••	•		•		•
•	•	•	•••	•		•		•
1	ω_n^{n-1}	$\omega_n^{2(n-1)}$	•••	$\omega_n^{(n-1)(n-1)}$		a_{n-1}		y_{n-1}

The element in the position (k, j) of the above Vandermonde matrix V_n containing the appropriate powers of ω_n , is ω_n^{kj} ; $0 \leq k \leq n-1$; $0 \leq j \leq n-1$. Hence

$$a = V_n^{-1}y$$

Theorem 5 The entry in the position (j, k) of V_n^{-1} is $\frac{1}{n}\omega_n^{-kj}$ for $0 \leq j \leq n-1; 0 \leq k \leq n-1$.

Proof. With this matrix V_n^{-1} we show that $V_n^{-1}V_n = I_n$.

$$\begin{aligned} (V_n^{-1}V_n)_{j,j'} &= \sum_{k=0}^{n-1} \left(\frac{1}{n}\omega_n^{-kj}\right)(\omega_n^{kj'}) \\ &= \sum_{k=0}^{n-1} \frac{1}{n}\omega_n^{k(j'-j)} \end{aligned}$$

This is 1 if $j' = j$ and 0 otherwise. Please note that $-(n-1) \leq j' - j \leq n-1$ and so if $j' \neq j$ then $j' - j$ is not divisible by n and hence by summation lemma, the result follows. ■

Hence

$$a_j = \frac{1}{n} \sum_{k=0}^{n-1} y_k \omega_n^{-kj}; 0 \leq j \leq n-1$$

This is similar to the equations:

$$y_k = \sum_{j=0}^{n-1} a_j \omega_n^{kj}; 0 \leq k \leq n-1$$

with roles of a and y reversed and ω_n^{-kj} instead of ω_n^{kj} . Thus in the above FFT we replace a by y (and conversely), and also ω_n by ω_n^{-1} we can compute this also in $O(n \lg n)$ time.

This gives us the following scheme for multiplying two polynomials of degree bound n given in coefficient form in $O(n \lg n)$ time as follows.

1. Convert to point-value form with $2n$ points for each polynomial. This (using FFT) has complexity $O(n \lg n)$.
2. Point wise multiply the values of the two polynomials at each point. This has complexity $O(n)$ and is point-value form of the product.
3. Convert to coefficient form using FFT (in its inverse form). This has complexity $O(n \lg n)$

Imbedded in all this, is a way to compute convolutions (these are the coefficients the product polynomial) and multiplying two integers.