Chapter 25 The Fast Fourier Transform

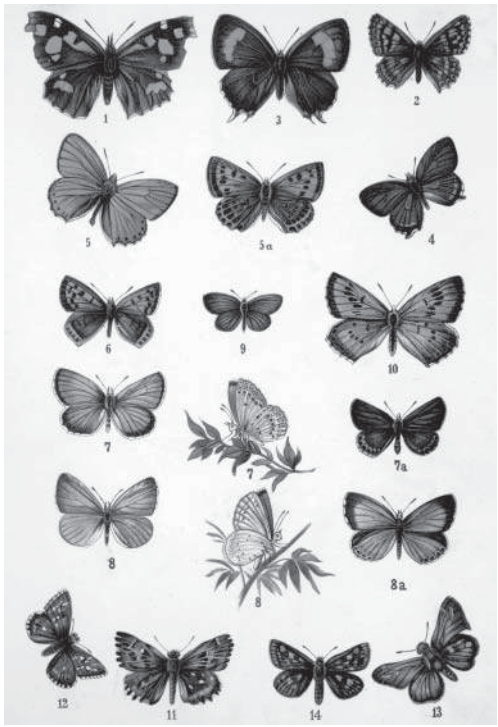

Plate VI from Europas bekannteste Schmetterlinge. Beschreibung der wichtigsten Arten und Anleitung zur Kenntnis und zum Sammeln der Schmetterlinge und Raupen (ca. 1895), F. Nemos, Oestergaard Verlag, Berlin, 18 chromolithographische Tafeln, 160 pp. Public domain image.

Contents

A common computation in many cryptographic systems is the multiplication of large integers. For instance, real-world uses of the RSA cryptosystem often involve 1024-bit, 2048-bit, or even 3072-bit keys; hence, RSA encryption and decryption with such keys involves the multiplication of large integers having these bit lengths. In fact, the U.S. National Institute for Standards and Technology (NIST) has argued that to achieve a high level of security for the RSA cryptosystem one should use 15,360-bit keys. Thus, from an algorithmic viewpoint, improving the running time for integer multiplication can result in faster and more secure cryptographic protocols involving large integers.

Unfortunately, a straightforward adaptation of the standard method for multiplying integers, as taught in elementary school, results in a method for multiplying two n-bit integers that runs in $O(n^2)$ time. This can be improved to an algorithm running in $O(n^{1.585})$ time, using the divide-and-conquer Karatsuba algorithm described in Section 11.2, which is okay for moderately large integers, but multiplying the large integers used in cryptographic computations could benefit from an even faster algorithm. Interestingly, the technique we discuss in this chapter, the Fast Fourier Transform (FFT), can be used to achieve a much faster algorithm for multiplying large integers. Moreover, it turns out that the FFT has many other applications as well, including fast methods for signal processing, image processing, scientific data analysis, and the pricing of financial options.

Suppose, then, that we want to multiply two n-bit integers, $P = a_{n-1} \ldots a_1 a_0$ and $Q = b_{n-1} \ldots b_1 b_0$. Rather than attack the problem of computing $R = P \cdot Q$ directly, however, let us reduce it to a related problem—polynomial multiplication. Construct two polynomials, $p(x)$ and $q(x)$, from P and Q as follows:

$$\begin{aligned} p(x) &= a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} \\ q(x) &= b_0 + b_1 x + a_2 x^2 + \cdots + b_{n-1} x^{n-1}, \end{aligned}$$

and note that $P = p(2)$ and $Q = q(2)$. Imagine, for the moment, that we had a fast algorithm to compute the polynomial

$$r(x) = p(x) \cdot q(x).$$

Then we could compute the integer product, $R = P \cdot Q$, by first computing the polynomial, $r(x)$, and then performing the evaluation $R \leftarrow r(2)$.

As one of the most surprising and ingenious results in algorithms, it turns out that we can compute an efficient representation for the product polynomial, $r(x)$, using $O(n \log n)$ arithmetic operations on reasonably sized numbers. Thus, using the above approach, we can design a method for multiplying two n-bit integers using $O(n \log n)$ such arithmetic operations. The FFT algorithm, which achieves this result, is based on an interesting use of the divide-and-conquer technique. We therefore devote this entire chapter to describing and analyzing this algorithm.

25.1 Convolution

A polynomial represented in ***coefficient form*** is described by a coefficient vector $\mathbf{a} = [a_0, a_1, \ldots, a_{n-1}]$ as follows:

$$p(x) = \sum_{i=0}^{n-1} a_i x^i.$$

The ***degree*** of such a polynomial is the largest index of a nonzero coefficient a_i. A coefficient vector of length n can represent polynomials of degree at most $n - 1$.

The coefficient representation is natural, in that it is simple and allows for several polynomial operations to be performed quickly. For example, given a second polynomial described using a coefficient vector $\mathbf{b} = [b_0, b_1, \ldots, b_{n-1}]$ as

$$q(x) = \sum_{i=0}^{n-1} b_i x^i,$$

we can easily add $p(x)$ and $q(x)$ component-wise to produce their sum,

$$p(x) + q(x) = \sum_{i=0}^{n-1} (a_i + b_i) x^i.$$

Likewise, the coefficient form for $p(x)$ allows us to evaluate $p(x)$ efficiently, by ***Horner's rule*** (Exercise C-1.14), as

$$p(x) = a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-2} + x a_{n-1}) \cdots)).$$

Thus, with the coefficient representation, we can add and evaluate degree-$(n-1)$ polynomials using $O(n)$ arithmetic operations.

Multiplying two polynomials $p(x)$ and $q(x)$, as defined above in coefficient form, is not straightforward, however. To see the difficulty, consider $p(x) \cdot q(x)$:

$$p(x) \cdot q(x) = a_0 b_0 + (a_0 b_1 + a_1 b_0)x + (a_0 b_2 + a_1 b_1 + a_2 b_0)x^2 + \cdots + a_{n-1} b_{n-1} x^{2n-2}.$$

That is,

$$p(x) \cdot q(x) = \sum_{i=0}^{2n-2} c_i x^i,$$

where

$$c_i = \sum_{j=0}^{i} a_j b_{i-j},$$

for $i = 0, 1, \ldots, 2n-2$. This equation defines a vector $\mathbf{c} = [c_0, c_1, \ldots, c_{2n-1}]$, which we call the ***convolution*** of the vectors $\mathbf{a}$ and $\mathbf{b}$. For symmetry reasons, we view the convolution as a vector of size $2n$, defining $c_{2n-1} = 0$. We denote the convolution of $\mathbf{a}$ and $\mathbf{b}$ as $\mathbf{a} * \mathbf{b}$.

Using the Interpolation Theorem for Polynomials

If we apply the definition of the convolution directly, then it will take us $\Theta(n^2)$ time to multiply the two polynomials p and q. The ***Fast Fourier Transform*** (***FFT***) algorithm allows us to perform this multiplication using $O(n \log n)$ arithmetic operations. The improvement of the FFT is based on an interesting observation. Namely, that another way of representing a degree-$(n-1)$ polynomial is by its value on n distinct inputs. Such a representation is unique, because of the following theorem.

Theorem 25.1 [The Interpolation Theorem for Polynomials]: *Given a set of n pairs,*

$$S = \{(x_0, y_0), (x_1, y_1), (x_2, y_2), \ldots, (x_{n-1}, y_{n-1})\},$$

such that the x_i's are all distinct, there is a unique degree-$(n-1)$ polynomial, $p(x)$, with $p(x_i) = y_i$, for $i = 0, 1, \ldots, n-1$.

Suppose, then, that we can represent a polynomial not by its coefficients, but instead by its value on a collection of different inputs. This theorem suggests an alternative method for multiplying two polynomials p and q. In particular, evaluate p and q for $2n$ different inputs $x_0, x_1, \ldots, x_{2n-1}$ and compute the representation of the product of p and q as the set

$$\{(x_0, p(x_0)q(x_0)),\ (x_1, p(x_1)q(x_1)),\ \ldots,\ (x_{2n-1}, p(x_{2n-1})q(x_{2n-1}))\}.$$

Such a computation would clearly require just $O(n)$ arithmetic operations, given the $2n$ input-output pairs for each of p and q.

The challenge, then, to effectively using this approach to multiply p and q is to come up quickly with $2n$ input-output pairs for p and q. Applying Horner's rule to $2n$ different inputs would require $\Theta(n^2)$ arithmetic operations, which is not asymptotically any faster than using the convolution directly. So Horner's rule is of no immediate help. Of course, we have full freedom in how we choose the set of $2n$ inputs for our polynomials. That is, we have full discretion to choose inputs that are easy to evaluate. For example, the evaluation,

$$p(0) = a_0,$$

is a simple case. But we have to choose a set of $2n$ easy inputs to evaluate p on, not just one. Fortunately, the mathematical concept we discuss next provides a convenient set of inputs that are collectively easier to use to evaluate a polynomial than applying Horner's rule $2n$ times.

25.2 Primitive Roots of Unity

One of the central ideas that allows for fast polynomial evaluation is the concept of primitive roots of unity.

Definition

A number, ω, is a ***primitive nth root of unity***, for $n \geq 2$, if it satisfies the following properties:

1. $\omega^n = 1$, that is, ω is an nth root of 1.
2. The numbers $1, \omega, \omega^2, \ldots, \omega^{n-1}$ are distinct.

Note that this definition implies that a primitive nth root of unity has a multiplicative inverse, $\omega^{-1} = \omega^{n-1}$, for

$$\omega^{-1}\omega = \omega^{n-1}\omega = \omega^n = 1.$$

Thus, we can speak in a well-defined fashion of negative exponents of ω, as well as positive ones.

A Complex Example of a Primitive Root of Unity

The notion of a primitive nth root of unity may, at first, seem like a strange definition with few examples. But it actually has several important instances.

One important example is the complex number

$$\omega = e^{2\pi i/n} = \cos(2\pi/n) + \mathbf{i}\sin(2\pi/n),$$

which is a primitive nth root of unity when we perform all our arithmetic in the complex number system, where $\mathbf{i} = \sqrt{-1}$.

An Integer Example of a Primitive Root of Unity

As another example, suppose that we have a prime number, $p = cn + 1$, for some positive integer, c. Choose x to be a multiplicative generator for the positive numbers in the finite field, Z_t, which is defined on the set of integers, $\{0, 1, 2, \ldots, t-1\}$ (see Section 24.1). That is, choose x so that the numbers, $x, x^2, \ldots, x^{t-1}$, are all distinct, modulo t. By Fermat's Little Theorem (19.5), $x^{t-1} \bmod t = 1$. Thus, the number,

$$\omega = x^c \bmod t,$$

is a primitive nth root of unity in Z_t.

Properties of Primitive Roots of Unity

Primitive nth roots of unity have a number of important properties, including the following three ones.

Lemma 25.2 (Cancellation Property): *If ω is an nth root of unity, then, for any integer $k \neq 0$, with $-n < k < n$,*

$$\sum_{j=0}^{n-1} \omega^{kj} = 0.$$

Proof: Since $\omega^k \neq 1$,

$$\sum_{j=0}^{n-1} \omega^{kj} = \frac{(\omega^k)^n - 1}{\omega^k - 1} = \frac{(\omega^n)^k - 1}{\omega^k - 1} = \frac{1^k - 1}{\omega^k - 1} = \frac{1 - 1}{\omega^k - 1} = 0.$$

■

Lemma 25.3 (Reduction Property): *If ω is a primitive $(2n)$th root of unity, then ω^2 is a primitive nth root of unity.*

Proof: If $1, \omega, \omega^2, \ldots, \omega^{2n-1}$ are distinct, then $1, \omega^2, (\omega^2)^2, \ldots, (\omega^2)^{n-1}$ are also distinct. ■

Lemma 25.4 (Reflective Property): *If ω is a primitive nth root of unity and n is even, then*

$$\omega^{n/2} = -1.$$

Proof: By the cancellation property, for $k = n/2$,

$$\begin{aligned} 0 &= \sum_{j=0}^{n-1} \omega^{(n/2)j} \\ &= \omega^0 + \omega^{n/2} + \omega^n + \omega^{3n/2} + \cdots + \omega^{(n/2)(n-2)} + \omega^{(n/2)(n-1)} \\ &= \omega^0 + \omega^{n/2} + \omega^0 + \omega^{n/2} + \cdots + \omega^0 + \omega^{n/2} \\ &= (n/2)(1 + \omega^{n/2}). \end{aligned}$$

Thus, $0 = 1 + \omega^{n/2}$. ■

An interesting corollary to the reflective property, which motivates its name, is the fact that if ω is a primitive nth root of unity and $n \geq 2$ is even, then

$$\omega^{k+n/2} = -\omega^k.$$

25.3 The Discrete Fourier Transform

Let us now return to the problem of evaluating a polynomial defined by a coefficient vector $\mathbf{a}$ as

$$p(x) = \sum_{i=0}^{n-1} a_i x^i,$$

for a carefully chosen set of input values. The technique we discuss in this section, called the ***Discrete Fourier Transform*** (DFT), is to evaluate $p(x)$ at the nth roots of unity, $\omega^0, \omega^1, \omega^2, \ldots, \omega^{n-1}$. Admittedly, this gives us just n input-output pairs, but we can "pad" our coefficient representation for p with 0's by setting $a_i = 0$, for $n \leq i \leq 2n-1$. This padding would let us view p as a degree-$(2n-1)$ polynomial, which would in turn let us use the primitive $(2n)$th roots of unity as inputs for a DFT for p. Thus, if we need more input-output values for p, let us assume that the coefficient vector for p has already been padded with as many 0's as necessary.

Formally, the Discrete Fourier Transform for the polynomial p represented by the coefficient vector $\mathbf{a}$ is defined as the vector $\mathbf{y}$ of values

$$y_j = p(\omega^j),$$

where ω is a primitive nth root of unity. That is,

$$y_j = \sum_{i=0}^{n-1} a_i \omega^{ij}.$$

In the language of matrices, we can alternatively think of the vector $\mathbf{y}$ of y_j values and the vector $\mathbf{a}$ as column vectors and say that

$$\mathbf{y} = F\mathbf{a},$$

where F is an $n \times n$ matrix such that $F[i,j] = \omega^{ij}$.

The Inverse Discrete Fourier Transform

Interestingly, the matrix F has an inverse, F^{-1}, so that $F^{-1}(F(\mathbf{a})) = \mathbf{a}$ for all $\mathbf{a}$. The matrix F^{-1} allows us to define an ***inverse Discrete Fourier Transform***. If we are given a vector $\mathbf{y}$ of the values of a degree-$(n-1)$ polynomial p at the nth roots of unity, $\omega^0, \omega^1, \ldots, \omega^{n-1}$, then we can recover a coefficient vector for p by computing

$$\mathbf{a} = F^{-1}\mathbf{y}.$$

Moreover, the matrix F^{-1} has a simple form, in that $F^{-1}[i,j] = \omega^{-ij}/n$. Thus, we can recover the coefficient a_i as

$$a_i = \sum_{j=0}^{n-1} y_j \omega^{-ij}/n.$$

The following lemma justifies this claim, and is the basis of why we refer to F and F^{-1} as "transforms."

Lemma 25.5: *For any vector* $\mathbf{a}$, $F^{-1} \cdot F\mathbf{a} = \mathbf{a}$.

Proof: Let $A = F^{-1} \cdot F$. It is enough to show that $A[i,j] = 1$ if $i = j$, and $A[i,j] = 0$ if $i \neq j$. That is, $A = I$, where I is the ***identity matrix***. By the definitions of F^{-1}, F, and matrix multiplication,

$$A[i,j] = \frac{1}{n}\sum_{k=0}^{n-1} \omega^{-ik}\omega^{kj}.$$

If $i = j$, then this equation reduces to

$$A[i,i] = \frac{1}{n}\sum_{k=0}^{n-1} \omega^{0} = \frac{1}{n} \cdot n = 1.$$

So, consider the case when $i \neq j$, and let $m = j - i$. Then the ijth entry of A can be written as

$$A[i,j] = \frac{1}{n}\sum_{k=0}^{n-1} \omega^{mk},$$

where $-n < m < n$ and $m \neq 0$. By the cancellation property for a primitive nth root of unity, the right-hand side of the above equation reduces to 0; hence,

$$A[i,j] = 0,$$

for $i \neq j$. ■

Given the DFT and the inverse DFT, we can now define our approach to multiplying two polynomials p and q.

To use the discrete Fourier transform and its inverse to compute the convolution of two coefficient vectors, $\mathbf{a}$ and $\mathbf{b}$, we apply the following steps, which we illustrate in a schematic diagram, as shown in Figure 25.1.

1. Pad $\mathbf{a}$ and $\mathbf{b}$ each with n 0's and view them as column vectors to define
$$\mathbf{a}' = [a_0, a_1, \ldots, a_{n-1}, 0, 0, \ldots, 0]^T$$
$$\mathbf{b}' = [b_0, b_1, \ldots, b_{n-1}, 0, 0, \ldots, 0]^T.$$
2. Compute the Discrete Fourier Transforms $\mathbf{y} = F\mathbf{a}'$ and $\mathbf{z} = F\mathbf{b}'$.
3. Multiply the vectors $\mathbf{y}$ and $\mathbf{z}$ component-wise, defining the simple product $\mathbf{y} \cdot \mathbf{z} = F\mathbf{a}' \cdot F\mathbf{b}'$, where
$$(\mathbf{y} \cdot \mathbf{z})[i] = (F\mathbf{a}' \cdot F\mathbf{b}')[i] = F\mathbf{a}'[i] \cdot F\mathbf{b}'[i] = y_i \cdot z_i,$$
for $i = 1, 2, \ldots, 2n-1$.
4. Compute the inverse Discrete Fourier Transform of this simple product. That is, compute $\mathbf{c} = F^{-1}(F\mathbf{a}' \cdot F\mathbf{b}')$.

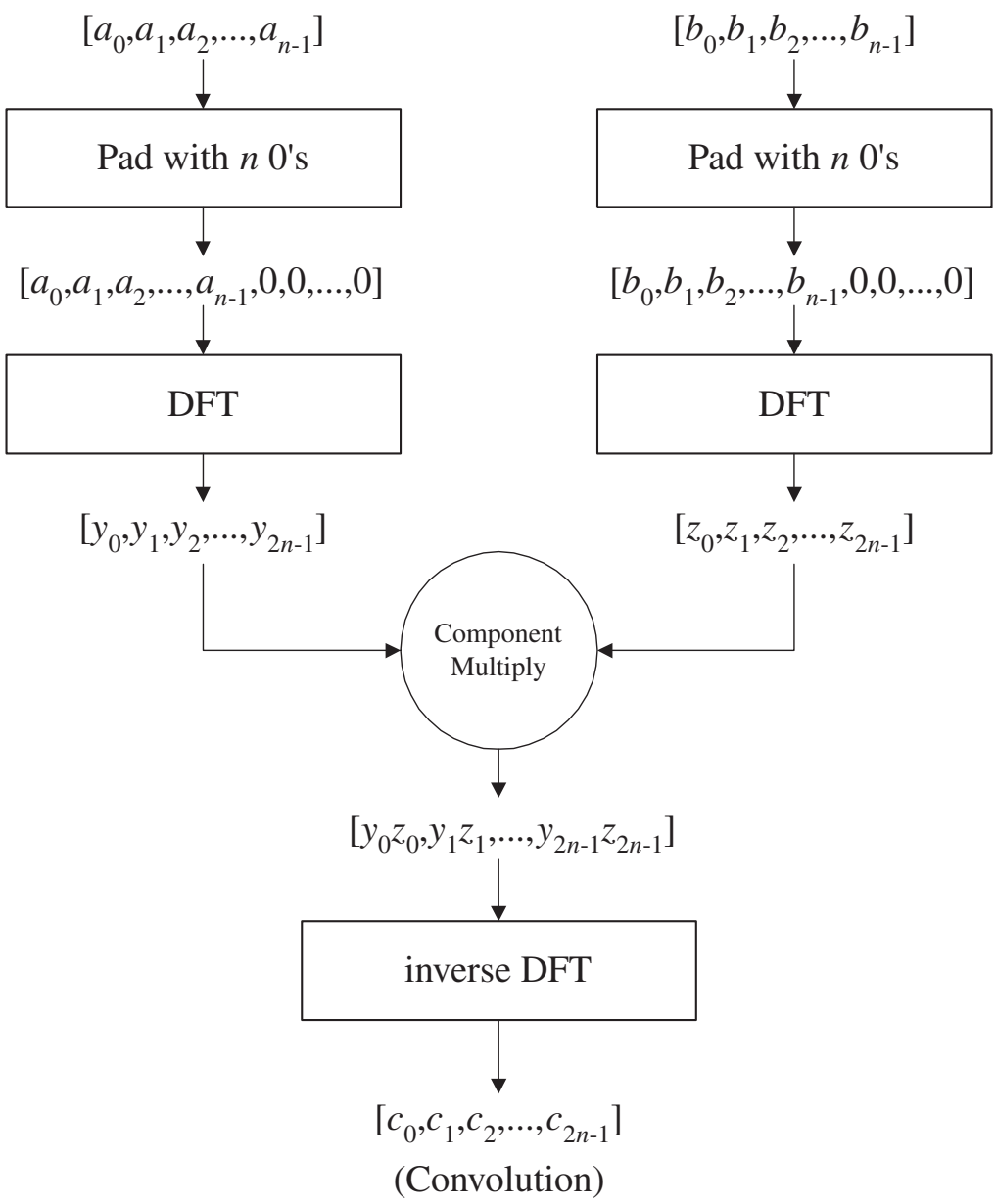

Figure 25.1: An illustration of the Convolution Theorem, to compute $\mathbf{c} = \mathbf{a} * \mathbf{b}$.

The reason the above approach works is because of the following.

Theorem 25.6 [The Convolution Theorem]: *Suppose we are given two n-length vectors $\mathbf{a}$ and $\mathbf{b}$ padded with 0's to $2n$-length vectors $\mathbf{a}'$ and $\mathbf{b}'$, respectively. Then $\mathbf{a} * \mathbf{b} = F^{-1}(F\mathbf{a}' \cdot F\mathbf{b}')$.*

Proof: We will show that $F(\mathbf{a} * \mathbf{b}) = F\mathbf{a}' \cdot F\mathbf{b}'$. So, consider $A = F\mathbf{a}' \cdot F\mathbf{b}'$. Since the second halves of $\mathbf{a}'$ and $\mathbf{b}'$ are padded with 0's,

$$\begin{aligned} A[i] &= \left(\sum_{j=0}^{n-1} a_j \omega^{ij}\right) \cdot \left(\sum_{k=0}^{n-1} b_k \omega^{ik}\right) \\ &= \sum_{j=0}^{n-1}\sum_{k=0}^{n-1} a_j b_k \omega^{i(j+k)}, \end{aligned}$$

for $i = 0, 1, \ldots, 2n-1$. Consider, next, $B = F(\mathbf{a} * \mathbf{b})$. By the definition of convolution and the DFT,

$$B[i] = \sum_{l=0}^{2n-1}\sum_{j=0}^{2n-1} a_j b_{l-j} \omega^{il}.$$

Substituting k for $l - j$, and changing the order of the summations, we get

$$B[i] = \sum_{j=0}^{2n-1}\sum_{k=-j}^{2n-1-j} a_j b_k \omega^{i(j+k)}.$$

Since b_k is undefined for $k < 0$, we can start the second summation above at $k = 0$. In addition, since $a_j = 0$ for $j > n-1$, we can lower the upper limit in the first summation above to $n-1$. But once we have made this substitution, note that the upper limit on the second summation above is always at least n. Thus, since $b_k = 0$ for $k > n-1$, we may lower the upper limit on the second summation to $n-1$. Therefore,

$$B[i] = \sum_{j=0}^{n-1}\sum_{k=0}^{n-1} a_j b_k \omega^{i(j+k)},$$

which proves the theorem. ∎

We now have a method for computing the multiplication of two polynomials that involves computing two DFTs, doing a simple linear-time component-wise multiplication, and computing an inverse DFT. Thus, if we can find a fast algorithm for computing the DFT and its inverse, then we will have a fast algorithm for multiplying two polynomials. We describe such a fast algorithm, which is known as the "Fast Fourier Transform," next.

25.4 The Fast Fourier Transform Algorithm

The ***Fast Fourier Transform*** (FFT) algorithm computes a Discrete Fourier Transform (DFT) of an n-length vector in $O(n \log n)$ time. In the FFT algorithm, we apply the divide-and-conquer approach to polynomial evaluation by observing that if n is even, we can divide a degree-$(n-1)$ polynomial

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$$

into two degree-$(n/2-1)$ polynomials

$$\begin{aligned} p^{\text{even}}(x) &= a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-2} x^{n/2-1} \\ p^{\text{odd}}(x) &= a_1 + a_3 x + a_5 x^2 + \cdots + a_{n-1} x^{n/2-1} \end{aligned}$$

and noting that we can combine these two polynomials into p using the equation

$$p(x) = p^{\text{even}}(x^2) + x p^{\text{odd}}(x^2).$$

The DFT evaluates $p(x)$ at each nth root of unity, $\omega^0, \omega^1, \omega^2, \ldots, \omega^{n-1}$. Note that, by the reduction property, the values $(\omega^2)^0, \omega^2, (\omega^2)^2, (\omega^2)^3, \ldots, (\omega^2)^{n-1}$ are $(n/2)$th roots of unity. Thus, we can evaluate each of $p^{\text{even}}(x)$ and $p^{\text{odd}}(x)$ at these values, and we can reuse those same computations in evaluating $p(x)$. This observation is used in Algorithm 25.2 (FFT), which takes as input an n-length coefficient vector $\mathbf{a}$ and a primitive nth root of unity ω, where n is a power of 2.

Algorithm $\mathsf{FFT}(\mathbf{a}, \omega)$:

Input: An n-length coefficient vector $\mathbf{a} = [a_0, a_1, \ldots, a_{n-1}]$ and a primitive nth root of unity ω, where n is a power of 2

Output: A vector $\mathbf{y}$ of values of the polynomial for $\mathbf{a}$ at the nth roots of unity

if $n = 1$ **then**
  return $\mathbf{y} = \mathbf{a}$.
$x \leftarrow \omega^0$ // x will store powers of ω, so initially $x = 1$.
// Divide Step, which separates even and odd indices
$\mathbf{a}^{\text{even}} \leftarrow [a_0, a_2, a_4, \ldots, a_{n-2}]$
$\mathbf{a}^{\text{odd}} \leftarrow [a_1, a_3, a_5, \ldots, a_{n-1}]$
// Recursive Calls, with ω^2 as $(n/2)$th root of unity, by the reduction property
$\mathbf{y}^{\text{even}} \leftarrow \mathsf{FFT}(\mathbf{a}^{\text{even}}, \omega^2)$
$\mathbf{y}^{\text{odd}} \leftarrow \mathsf{FFT}(\mathbf{a}^{\text{odd}}, \omega^2)$
// Combine Step, using $x = \omega^i$
for $i \leftarrow 0$ to $n/2 - 1$ **do**
  $y_i \leftarrow y_i^{\text{even}} + x \cdot y_i^{\text{odd}}$
  $y_{i+n/2} \leftarrow y_i^{\text{even}} - x \cdot y_i^{\text{odd}}$ // Uses reflective property
  $x \leftarrow x \cdot \omega$
return $\mathbf{y}$

Algorithm 25.2: Recursive FFT algorithm.

The Correctness of the FFT Algorithm

The pseudocode description in Algorithm 25.2 for the FFT algorithm is deceptively simple, so let us say a few words about why it works correctly. First, note that the base case of the recursion, when $n = 1$, correctly returns a vector $\mathbf{y}$ with the one entry, $y_0 = a_0$, which is the leading and only term in the polynomial $p(x)$ in this case.

In the general case, when $n \geq 2$, we separate $\mathbf{a}$ into its even and odd instances, $\mathbf{a}^{\text{even}}$ and $\mathbf{a}^{\text{odd}}$, and recursively call the FFT using ω^2 as the $(n/2)$th root of unity. As we have already mentioned, the reduction property of a primitive nth root of unity, allows us to use ω^2 in this way. Thus, we may inductively assume that

$$\begin{aligned} y_i^{\text{even}} &= p^{\text{even}}(\omega^{2i}) \\ y_i^{\text{odd}} &= p^{\text{odd}}(\omega^{2i}). \end{aligned}$$

Let us therefore consider the for-loop that combines the values from the recursive calls. Note that in the i iteration of the loop, $x = \omega^i$. Thus, when we perform the assignment statement

$y_i \leftarrow y_i^{\text{even}} + x y_i^{\text{odd}}$,

we have just set

$$\begin{aligned} y_i &= p^{\text{even}}((\omega^2)^i) + \omega^i \cdot p^{\text{odd}}((\omega^2)^i) \\ &= p^{\text{even}}((\omega^i)^2) + \omega^i \cdot p^{\text{odd}}((\omega^i)^2) \\ &= p(\omega^i), \end{aligned}$$

and we do this for each index $i = 0, 1, \ldots, n/2 - 1$. Similarly, when we perform the assignment statement

$y_{i+n/2} \leftarrow y_i^{\text{even}} - x y_i^{\text{odd}}$,

we have just set

$$y_{i+n/2} = p^{\text{even}}((\omega^2)^i) - \omega^i \cdot p^{\text{odd}}((\omega^2)^i).$$

Since ω^2 is a primitive $(n/2)$th root of unity, $(\omega^2)^{n/2} = 1$. Moreover, since ω is itself a primitive nth root of unity,

$$\omega^{i+n/2} = -\omega^i,$$

by the reflection property. Thus, we can rewrite the above identity for $y_{i+n/2}$ as

$$\begin{aligned} y_{i+n/2} &= p^{\text{even}}((\omega^2)^{i+(n/2)}) - \omega^i \cdot p^{\text{odd}}((\omega^2)^{i+(n/2)}) \\ &= p^{\text{even}}((\omega^{i+(n/2)})^2) + \omega^{i+n/2} \cdot p^{\text{odd}}((\omega^{i+(n/2)})^2) \\ &= p(\omega^{i+n/2}), \end{aligned}$$

and this will hold for each $i = 0, 1, \ldots, n/2 - 1$. Thus, the vector $\mathbf{y}$ returned by the FFT algorithm will store the values of $p(x)$ at each of the nth roots of unity.

Analyzing the FFT Algorithm

The FFT algorithm follows the divide-and-conquer paradigm, dividing the original problem of size n into two subproblems of size $n/2$, which are solved recursively. We assume that each arithmetic operation performed by algorithms takes $O(1)$ time. The divide step as well as the combine step for merging the recursive solutions, each take $O(n)$ time. Thus, we can characterize the running time $T(n)$ of the FFT algorithm using the recurrence equation

$$T(n) = 2T(n/2) + bn,$$

for some constant $b > 0$. By the Master Theorem (11.4), $T(n)$ is $O(n \log n)$. Therefore, we can summarize our discussion as follows.

Theorem 25.7: *Given an n-length coefficient vector* **a** *defining a polynomial $p(x)$, and a primitive nth root of unity, ω, the* FFT *algorithm evaluates $p(x)$ at each of the nth roots of unity, ω^i, for $i = 0, 1, \ldots, n-1$, using $O(n \log n)$ arithmetic operations.*

There is also an inverse FFT algorithm, which computes the inverse DFT in $O(n \log n)$ time. The details of this algorithm are similar to those for the FFT algorithm and are left as an exercise (R-25.1). Combining these two algorithms in our approach to multiplying two polynomials $p(x)$ and $q(x)$, given their n-length coefficient vectors, we have an algorithm for computing this product using $O(n \log n)$ arithmetic operations, in the field used to define the primitive roots of unity used by the FFT algorithm.

Multiplying Big Integers

Let us revisit the problem discussed in the introduction, of multiplying two n-bit integers. Namely, suppose we are given two big integers P and J that use at most $n \geq 64$ bits each, where n is a power of 2, and we are interested in computing $R = P \cdot Q$. As mentioned earlier, we construct two polynomials, $p(x)$ and $q(x)$, from P and Q as follows:

$$\begin{aligned} p(x) &= a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} \\ q(x) &= b_0 + b_1 x + a_2 x^2 + \cdots + b_{n-1} x^{n-1}, \end{aligned}$$

and note that $P = p(2)$ and $Q = q(2)$. Then we use the FFT algorithm to compute a coefficient representation of the degree-$2n$ polynomial,

$$r(x) = p(x) \cdot q(x).$$

This takes $O(n \log n)$ arithmetic operations, say, in Z_t, for a prime number, t, that can be represented using $O(\log n)$ bits. This is likely to be on the order of the word size of our computer, since it takes $O(\log n)$ bits just to represent the number n. Finally, given this representation for $r(x)$, we need to compute $r(2)$ and assign this to R.

A Divide-and-Conquer Algorithm for Evaluating $r(2)$

We can compute $r(2)$ efficiently via yet another divide-and-conquer algorithm, by noting that if $r_1(x)$ is a polynomial defined by the first n (lower-order) coefficients of $r(x)$ and $r_2(x)$ is a polynomial defined by the second n (higher-order) coefficients of $r(x)$, then

$$r(x) = r_1(x) + r_2(x) \cdot x^n.$$

Thus, we can evaluate $r(2)$ as follows:

if $n = 1$ **then**
  return $r(2)$
recursively compute $R_1 \leftarrow r_1(2)$
recursively compute $R_2 \leftarrow r_2(2)$
Let $R_2' \leftarrow R_2 \cdot 2^n$
return $R_1 + R_2'$

Note that doing the multiplication of R_2 and 2^n is not as difficult as the integer multiplication problem we are trying to solve. In particular, since we are computing $r(2)$ in binary, we can multiply $r_1(2)$ by 2^n by a left shift of the bits of $r_2(2)$ by n places. Thus, we can do the final multiplication by 2^n and addition of the resulting $O(n)$-bit numbers in $O(n)$ time. Therefore, this divide-and-conquer evaluation algorithm can be characterized by the recurrence equation,

$$T(n) = 2T(n/2) + bn,$$

for some constant $b \geq 1$; hence, the evaluation of $r(2)$ can be done in $O(n \log n)$ time. This gives us the following.

Theorem 25.8: *Given two n-bit integers P and Q, we can compute the product $R = P \cdot Q$ using $O(n \log n)$ arithmetic operations.*

Here, the arithmetic operations are done in the number system that is used to define the primitive nth roots of unity required by the FFT algorithm. For instance, if we can do all the arithmetic in Z_t, for a prime $t = cn + 1$, for a small integer constant, c, such that t can be stored in a single word on our computer, then we can perform each arithmetic operation in $O(1)$ time in the RAM model.

In some cases, we cannot assume that arithmetic involving reasonably sized words can be done in constant time, however. In such scenarios, we must pay constant time for every bit operation. In this model it is still possible to use the FFT to multiply two n-bit integers, but the details are somewhat more complicated and the running time increases to $O(n \log n \log \log n)$. We omit the details for this approach here.

Implementing the FFT Algorithm to Avoid Repeated Array Allocation

The pseudocode for the recursive FFT algorithm calls for the allocation of several new arrays, including $\mathbf{a}^{\text{even}}$, $\mathbf{a}^{\text{odd}}$, $\mathbf{y}^{\text{even}}$, $\mathbf{y}^{\text{odd}}$, and $\mathbf{y}$. Allocating all of these arrays with each recursive call could prove to be a costly amount of extra work. If it can be avoided, saving this additional allocation of arrays could significantly improve the constant factors in the running time of the FFT algorithm.

Fortunately, the structure of FFT allows us to avoid this repeated array allocation. Instead of allocating many arrays, we can use a single array, A, for the input coefficients and use a single array, Y, for the answers. The main idea that allows for this usage is that we can think of the arrays A and Y as partitioned into subarrays, each one associated with a different recursive call. We can identify these subarrays using just two variables, base, which identifies the base address of the subarray, and n, which identifies the size of the subarray. Thus, we can avoid the overhead associated with allocating lots of small arrays with each recursive call.

Having decided that we will not allocate new arrays during the FFT recursive calls, we must deal with the fact that the FFT algorithm involves performing separate computations on even and odd indices of the input array. In the pseudocode of Algorithm 25.2, we use new arrays $\mathbf{a}^{\text{even}}$ and $\mathbf{a}^{\text{odd}}$, but now we must use subarrays in A for these vectors. Our solution for this memory management problem is to take the current n-cell subarray in A, and divide it into two subarrays of size $n/2$. One of the subarrays will have the same base as A, while the other has base base $+ n/2$. We move the elements at even indices in A to the lower half and we move elements at odd indices in A to the upper half. In doing so, we define an interesting permutation known as the ***inverse shuffle***. This permutation gets its name from its resemblance to the inverse of the permutation we would get by cutting the array A in half and shuffling it perfectly as if it were a deck of cards. Repeating it recursively on each half gives rise to a structure known as the ***butterfly network***, because of its symmetry. (See Figure 25.3.)

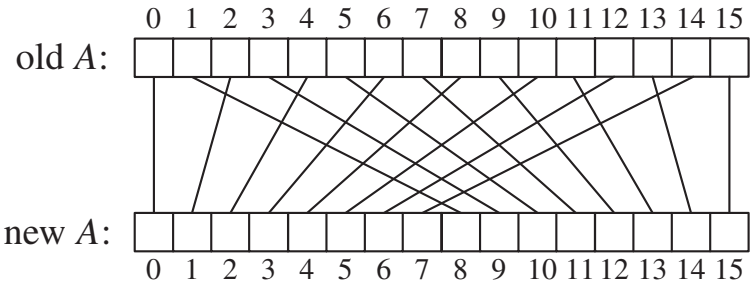

Figure 25.3: An illustration of the inverse shuffle permutation.

Avoiding Recursion

Another constant-time improvement we can make to the running of the FFT algorithm is to avoid recursion. The main challenge in such an implementation is that we have to figure out a way of performing all the inverse shuffles in the input array A. Rather than performing each inverse shuffle with each iteration, we instead perform all the inverse shuffles in advance, assuming that n, the size of the input array, is a power of two.

In order to figure out the net effect of the permutation we would get by repeated and recursive inverse shuffle operations, let us consider how the inverse shuffles move data around with each recursive call. In the first recursive call, of course, we perform an inverse shuffle on the entire array A. Note how this permutation operates at the bit level of the indices in A. It brings all elements at addresses that have a 0 as their least significant bit to the bottom half of A. Likewise, it brings all elements at addresses that have a 1 as their least significant bit to the top half of A. That is, if an element starts out at an address with b as its least significant bit, then it ends up at an address with b as its most significant bit. The least significant bit in an address is the determiner of which half of A an element winds up in. In the next level of recursion, we repeat the inverse shuffle on each half of A. Viewed again at the bit level, for $b = 0, 1$, these recursive inverse shuffles take elements originally at addresses with b as their second least significant bit, and move them to addresses that have b as their second most significant bit. Likewise, for $b = 0, 1$, the ith levels of recursion move elements originally at address with b as their ith least significant bit, to addresses with b as their ith most significant bit. Thus, if an element starts out at an address with binary representation $[b_{l-1} \ldots b_2 b_1 b_0]$, then it ends up at an address with binary representation $[b_0 b_1 b_2 \ldots b_{l-1}]$, where $l = \log_2 n$. That is, we can perform all the inverse shuffles in advance just by moving elements in A to the address that is the bit reversal of their starting address in A. To perform this permutation, we build a permutation array, reverse, in the multiply method, and then use this inside the FFT method to permute the elements in the input array A according to this permutation.

25.5 Exercises

Reinforcement

R-25.1 Describe the inverse FFT algorithm, which computes the inverse DFT in $O(n \log n)$ time. That is, show how to reverse the roles of $\mathbf{a}$ and $\mathbf{y}$ and change the assignments so that, for each output index, we have

$$a_i = \frac{1}{n} \sum_{j=1}^{n-1} y_j \omega^{-ij}.$$

R-25.2 Write the complex nth roots of unity for $n = 4$ and $n = 8$ in the form $a + b\mathbf{i}$.

R-25.3 What is the bit-reversal permutation, reverse, for $n = 16$?

R-25.4 Show that 5 is a multiplicative generator of the positive numbers in Z_{17}.

R-25.5 Use the FFT and inverse FFT to compute the convolution of $\mathbf{a} = [1, 2, 3, 4]$ and $\mathbf{b} = [4, 3, 2, 1]$, using arithmetic in Z_{17}. Use the fact that 5 is a generator for the positive elements of Z_{17}, and show the output of each component as in Figure 25.1.

R-25.6 Use the convolution theorem to compute the product of the polynomials $p(x) = 3x^2 + 4x + 2$ and $q(x) = 2x^3 + 3x^2 + 5x + 3$, using arithmetic in Z_{17}. You may use the fact that 5 is a generator for the positive elements of Z_{17}.

R-25.7 Compute the discrete Fourier transform of the vector $[5, 4, 3, 2]$ using arithmetic modulo $17 = 2^4 + 1$. Use the fact that 5 is a generator for the positive elements in Z_{17}.

R-25.8 Compute the product of the binary numbers $(01101000)_2$ and $(10001011)_2$ using the algorithm given in the book.

R-25.9 What is the exact number of recursive calls made to compute the convolution of the vectors $[6, 2, 3, 5, 2, 5, 8, 3, 2, 6]$ and $[4, 2, 3, 2, 7, 3, 3, 9]$, using recursive definitions of the FFT and inverse FFT algorithms?

Creativity

C-25.1 Prove the following more general form of the reduction property of primitive roots of unity: For any integer $c > 0$, if ω is a primitive (cn)th root of unity, then ω^c is a primitive nth root of unity.

C-25.2 Prove that $\omega = 2^{4b/m}$ is a primitive mth root of unity when multiplication is taken modulo $(2^{2b} + 1)$, for any integer $b > 0$ that is a multiple of m.

C-25.3 Given degree-n polynomials $p(x)$ and $q(x)$, describe a method for multiplying the derivatives of $p(x)$ and $q(x)$, that is, $p'(x) \cdot q'(x)$, using $O(n \log n)$ arithmetic operations.

C-25.4 Describe a version of the FFT that works when n is a power of 3 by dividing the input vector into three subvectors, recursing on each one, and then merging the subproblem solutions. Derive a recurrence equation for the running time of this algorithm and solve this recurrence using the Master Theorem.

C-25.5 Describe a method for computing the coefficients of the polynomial,

$$P(x) = (x+1)^n,$$

in $O(n)$ time.

Applications

A-25.1 In ***Shamir secret sharing***, an administrator, Bob, chooses a secret number, s, in a finite field, Z_p, for some prime number, p. He then chooses $n-1$ more random numbers, $a_1, a_2, \ldots, a_{n-1}$, in Z_p, and uses them to define the polynomial,

$$p(x) = s + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}.$$

Then, for each of $n < p$ friends, he chooses a distinct value, x_i, and distributes $p(x_i)$ to friend number i. Argue why it is impossible for a group of $(n-1)$ friends to learn the secret s, but if all n friends cooperate, they can learn s. Also, describe a method for Bob to compute all the values, $p(x_1), p(x_2), \ldots, p(x_n)$, using $O(n^2)$ arithmetic operations in Z_p.

A-25.2 Consider the Shamir secret sharing problem from the previous exercise, but now design an algorithm for computing all the values, $p(x_1), p(x_2), \ldots, p(x_n)$, using $O(n \log^2 n)$ arithmetic operations. You may use "as a black box" an algorithm, PolyDivide, which takes two polynomials, $p(x)$ and $q(x)$, given in coefficient form, with each of them having degree at most $(n-1)$, and returns the remainder polynomial, $r(x)$,

$$r(x) = p(x) \bmod q(x),$$

in coefficient form, using $O(n \log n)$ arithmetic operations. In addition, you may use the fact that, for any point, x_i,

$$p(x_i) = p(x) \bmod (x - x_i).$$

Finally, you may use the fact that if we let

$$q_{i,j}(x) = \prod_{k=i}^{j} (x - x_k),$$

then $p(x) = p(x) \bmod q_{1,n}(x)$, and

$$p(x) \bmod q_{i,j}(x) = (p(x) \bmod q_{k,l}(x)\,) \bmod q_{i,j}(x),$$

for $k \leq i \leq j \leq l$.

A-25.3 In some numerical computing applications, a desired computation is to find a polynomial that goes through a given set of points on a line, which, without loss of generality, we can assume is the x-axis. So suppose you are given a set of real numbers

$$X = \{x_0, x_1, \ldots, x_{n-1}\}.$$

Note that, by the Interpolation Theorem for Polynomials, there is a unique degree-$(n-1)$ polynomial $p(x)$, such that

$$p(x_i) = 0, \quad \text{for } i = 0, 1, \ldots, n-1,$$

and these are the only 0-values for the polynomial. Design a divide-and-conquer algorithm that can construct a coefficient-form representation of this polynomial, $p(x)$, using $O(n \log^2 n)$ arithmetic operations.

A-25.4 Suppose you have a software method, Conv, that can perform the convolution of two length-n integer vectors, A and B, using the FFT algorithm described in this chapter. Suppose further that you have been asked to build a system that can take an n-bit binary "text" string, T, and an m-bit binary "pattern" string, P, for $m \leq n$, and determine all the places in T where P appears as a substring. Show that in $O(n)$ time, plus the time needed for calls to the Conv method, you can solve this pattern matching problem by making two calls to the Conv function.

Hint: Note that the kth position in the convolution of two bit strings, A and B, counts the number of 1's that match among the first $k-1$ places in A with the last $k-1$ places in the reversal of B.

A-25.5 Consider a generalization of the pattern matching problem from the previous exercise, where we allow the pattern P and text T to be strings defined over an arbitrary alphabet, Σ. Show that you can still find all occurrences of P in T using two calls to the Conv method. In this case, your algorithm should run in $O(n \log |\Sigma|)$ time, plus the time needed for the calls to the Conv method.

A-25.6 Consider a further generalization of the pattern matching problem from the previous exercise, where we allow the pattern, P, to contain instances of a special "wild card" or "don't care" symbol, `*`, which matches any character in the alphabet, Σ. For example, with

$$P = \texttt{ab**c}$$

and

$$T = \texttt{babdfcabghci},$$

P matches T in positions 2 and 7. Show that, even in this case, you can still find all occurrences of P in T using two calls to the Conv method. In this case, your algorithm should run in $O(n \log |\Sigma|)$ time, plus the time needed for the calls to the Conv method. (Also, note that this problem cannot be solved using the efficient algorithms from Chapter 23.)

A-25.7 Suppose you are given a set, S, of n distinct number pairs, (x, y), such as in the Shamir secret sharing scheme described in Exercise A-25.1. Furthermore, assume that you have a software method, LinSolve, for solving a system of n linear equations with n unknowns. Describe how to produce a coefficient-form representation of the unique degree-$(n-1)$ polynomial that satisfies $y = p(x)$, for each (x, y) in S. Your algorithm should run in $O(n^2)$ time plus the time taken by the LinSolve method.

A-25.8 In financial and scientific data analysis applications, such as in spotting trends in stocks, we are often interested in making sense of noisy or highly fluctuating data. One method to achieve this goal is to take an average of recent values, as shown in Figure 25.4. For instance, in using a ***weighted moving average***, one begins by specifying a sequence of m weights, $W = (w_0, w_1, \ldots, w_{m-1})$, with

$$\sum_{i=0}^{m-1} w_i \;=\; 1.$$

Typically, one chooses the weights so that $w_i > w_{i+1}$, for $i = 1, \ldots, n-1$, so as to give greater emphasis to recent data. Then, given a sequence of $n \geq m$ data values, $X = (x_0, x_1, \ldots, x_{n-1})$, the ith value of the weighted moving average is computed as

$$A_i = w_0 a_i + w_1 a_{i-1} + w_2 a_{i-2} + \cdots .$$

For example, if $W = (0.5, 0.3, 0.2)$ and the three most recent data values were 136, 150, 200, then the current weighted moving average would be

$$A_i = 0.5(136) + 0.3(150) + 0.2(200) = 153,$$

which is closer to 136 than it is to 200. Given the sequences W and X, as specified above, describe an efficient method for computing all the A_i values, for $i = 0, 1, \ldots, n-1$, using $O(n \log n)$ arithmetic operations.

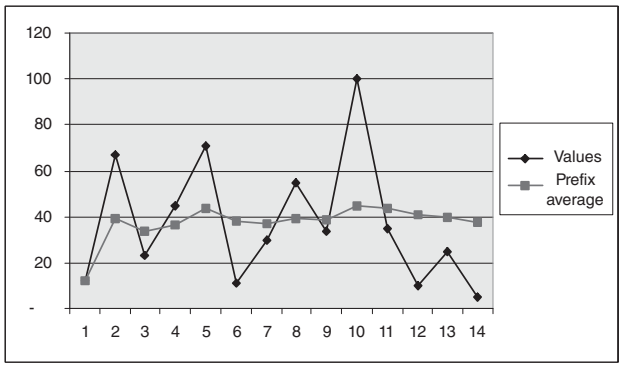

Figure 25.4: Smoothing data by taking an average of recent values.

Chapter Notes

The Fast Fourier Transform (FFT) appears in a paper by Cooley and Tukey [49]. It is also discussed in books by Aho, Hopcroft, and Ullman [8], Baase [18], and Yap [219], all of which were influential in the discussion given above. The fast integer multiplication method, running in $O(n \log n \log \log n)$ time, is due to Schönhage and Strassen [186]. For information on additional applications of the FFT, the interested reader is referred to books by Brigham [39] and Elliott and Rao [65], and the chapter by Emiris and Pan [66]. The connection between string matching and convolution begins with work by Fischer and Paterson [70]. Shamir describes a polynomial-based way to share a secret in [194].