23. Convolution and the FFT

Mathematics lies at the heart of the study of algorithms in several ways: mathematics gives us a rich framework for studying and analyzing algorithms, and conversely algorithms tell us the most efficient ways to solve mathematical problems. If you look at the names attached to many of the most important mathematical algorithms we use today (e.g., *Gaussian* elimination, *Euclid's* algorithm, and *Newton's* method), you will see that many of the most important mathematical algorithms in use today actually date back several centuries, to a time in which the lack of automated computing devices gave mathematicians a strong incentive to develop faster methods for performing lengthy manual calculations. Since mathematics is where computer science began, it seems perhaps fitting that we end our book with a topic area on algorithms for mathematical problems.

This chapter is relatively short. It revolves around one central mathematical idea, *convolution*, and a famous divide-and-conquer algorithm called the *Fast Fourier Transform* (FFT) for performing convolution efficiently. As we discuss numerous applications of convolution at the beginning of the chapter, it will become clear why the discovery of the FFT ranks as one of the most significant advances in modern computation. Following this chapter, we discuss algorithms for problems involving matrices and linear systems in the next chapter, and then number theoretic algorithms in the final chapter.

We begin the chapter with a description of convolution and a survey of its diverse applications: fast arithmetic on polynomials and large integers, polynomial evaluation and interpolation, pattern matching with wildcard characters, signal processing, solving certain types of counting problems, finding the probability distribution of a sum of two random variables, and calculations involving special types of matrices. After this, we describe the operation of the FFT algorithm itself. We discuss how the FFT gives us an efficient means of performing the *Discrete Fourier Transform* (DFT), and how this in turn allows us to perform fast convolution.

23.1 Convolution and its Applications

Let $a = a_0 \ldots a_{n-1}$ and $b = b_0 \ldots b_{n-1}$ be any two length-n vectors (we use zero-based indexing throughout the chapter since it is the most natural convention here).

Addition of a and b is of course a well-defined operation, producing a vector whose ith component is $a_i + b_i$. We can also "multiply" a and b by taking their dot product to obtain the single number $\sum_{i=0}^{n-1} a_i b_i$. Convolution gives us another natural and simple way to "multiply" two vectors. The convolution of a and b, denoted $a * b$, is a vector $c_0 \ldots c_{2n-2}$ where c_i is the dot product of $a_0 \ldots a_i$ and $b_i \ldots b_0$:

$$
\begin{aligned}
c_0 &= a_0 b_0 \\
c_1 &= a_0 b_1 + a_1 b_0 \\
c_2 &= a_0 b_2 + a_1 b_1 + a_2 b_0 \\
&\vdots \\
c_{n-1} &= a_0 b_{n-1} + a_1 b_{n-2} + \ldots + a_{n-2} b_1 + a_{n-1} b_0 \\
&\vdots \\
c_{2n-4} &= a_{n-3} b_{n-1} + a_{n-2} b_{n-2} + a_{n-1} b_{n-3} \\
c_{2n-3} &= a_{n-2} b_{n-1} + a_{n-1} b_{n-2} \\
c_{2n-2} &= a_{n-1} b_{n-1}
\end{aligned}
$$

Over the course of this chapter, we will see several other ways to picture the convolution operation. For example, in Figure 23.1(a) we show how $a * b$ can be computed using the standard "long multiplication" algorithm you probably learned in elementary school for multiplying large integers.

It takes $\Theta(n^2)$ time to compute $x * y$ by a direct application of the definition above. The main result of this chapter is that the FFT will give us a means of convolving two length-n vectors in only $O(n \log n)$ time. We will learn the operation of the FFT shortly, right after we discuss several applications of convolution in order to gain more insight and motivation into the importance and versatility of this operation.

Convolution is well-defined for vectors of different lengths. To convolve a short vector a with a longer vector b, this is equivalent to padding a with zeros until it has the same length as b, then applying the formula above. We also note that convolution satisfies most of the same familiar properties we have with normal multiplication: for any vectors a, b, and c, we have $a * b = b * a$ (the commutative property), $(a*b)*c = a*(b*c)$ (the associative property), and $a*(b+c) = (a*b)+(a*c)$ (the distributive property).

23.1.1 Polynomial and Integer Multiplication

Let $A(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_{n-1} x^{n-1}$ and $B(x) = b_0 + b_1 x + b_2 x^2 + \ldots + b_{n-1} x^{n-1}$ be two polynomials of degree $n-1$ with coefficient sequences $a = a_0 \ldots a_{n-1}$ and $b = b_0 \ldots b_{n-1}$. If we compute the product of these two polynomials, we obtain a polynomial of degree $2n-2$ whose coefficient sequence is precisely $a * b$. For this reason, many prefer to think of convolution as nothing more than polynomial multiplication, since the two are completely equivalent. Using the FFT, we therefore obtain a fast algorithm for multiplying polynomials, capable of multiplying two degree-n polynomials in $O(n \log n)$ time.

Large integers are very closely related to polynomials. For example, we can write the base-10 number 7392 as the polynomial $7x^3 + 3x^2 + 9x + 2$ evaluated at $x = 10$. In this manner, we can represent any n-digit number p using the n coefficients of a polynomial $p(x)$ of degree $n-1$. Suppose now that we want to multiply two large

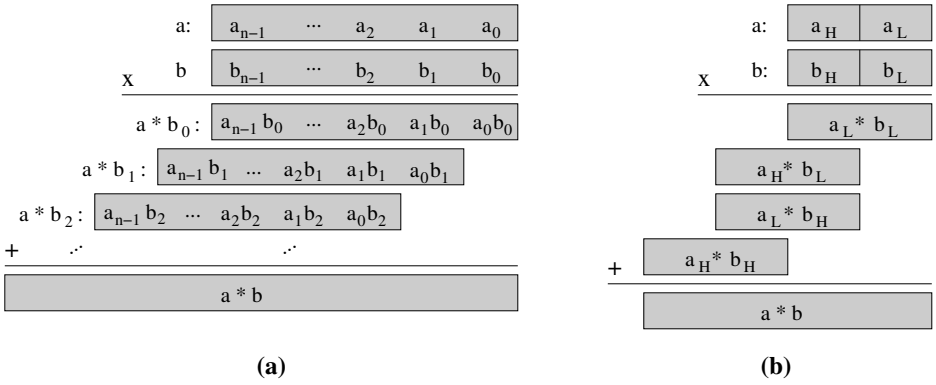

FIGURE 23.1: Convolution of two length-n vectors as performed by (a) "long multiplication", and (b) the divide-and-conquer approach described in problem 463, using blocks length $n/2$.

integers p and q, each containing at most n digits. This is *almost* equivalent to computing the coefficients in the product of $p(x)$ and $q(x)$. To give an example, let $p = 123$ and $q = 456$, so $p(x) = x^2 + 2x + 3$ and $q(x) = 4x^2 + 5x + 6$. The product of these two polynomials is $p(x)q(x) = 4x^4 + 15x^3 + 10x^2 + 8x + 8$, whose coefficient sequence is 4, 15, 10, 8, 8. If we look at the actual product $pq = 56088$, we see that this is essentially the same, after we spend $O(n)$ time making a post-processing scan from right to left to adjust for carries. [Brief details]

As a result of the near equivalence between large integer multiplication and polynomial multiplication, we can use the FFT to multiply two n-digit integers[1] in only $O(n \log n)$ time. This result is important in the area of cryptography, since many of today's prominent encryption and decryption algorithms perform a significant amount of large integer arithmetic. Without the FFT, multiplication of n-digit integers takes $\Theta(n^2)$ time[2] using the standard "elementary school" method for long multiplication shown in Figure 23.1(a).

Problem 462 (Computing the Coefficients of a Polynomial Given its Roots). Using the FFT as a black box that performs convolution in $O(n \log n)$ time, please describe a simple $O(n \log^2 n)$ divide-and-conquer algorithm for computing the coefficients of a degree-n polynomial $A(x)$ given its n roots $r_1 \ldots r_n$. Please assume the leading coefficient of our polynomial is one, so we can write $A(x) = (x - r_1)(x - r_2) \ldots (x - r_n)$. [Solution]

Problem 463 (A Simple Divide-and-Conquer Algorithm for Convolution). Although convolution (respectively polynomial and integer multiplication) takes $\Theta(n^2)$ time if we use the obvious straightforward implementation, we can use a simple divide-and-conquer approach to improve the running time to $\Theta(n^\alpha)$, where $\alpha = \log_2 3 \approx 1.57$. This approach also illustrates the very useful principle of *block multiplication*, which

[1] When we say "n-digit integer", we mean an integer that takes n digits to express in some base $b = O(1)$, such as an n-digit binary or decimal integer.

[2] Recall that we typically assume a word size of $\Theta(\log n)$ with the RAM model of computation, where n somehow represents the size of our input. Under this assumption, $\Theta(\log n)$-digit integers are the largest numbers we can claim to be able to multiply in a single step.

we shall use on many occasions in this chapter and also when we study matrix multiplication later. As shown in Figure 23.1(b), we can convolve two length-n vectors a and b by splitting each vector into two blocks of length $n/2$. We then solve the much simpler problem of convolving two length-2 vectors on these blocks, only using convolution of whole blocks in place of multiplication. Letting a_H and a_L denote the high and low halves of a, and b_H and b_L denote the high and low halves of b, this approach seems to require four recursive convolutions on length-$n/2$ blocks to compute the intermediate products $a_H * b_H$, $a_H * b_L$, $a_L * b_H$, and $a_L * b_L$, which are then added together in $O(n)$ time to obtain $a * b$. If we perform each length-$n/2$ convolution recursively, we can describe the running time of the entire algorithm using the recurrence $T(n) = 4T(n/2) + \Theta(n)$, which unfortunately solves to $T(n) = \Theta(n^2)$. However, with a simple trick, we can reduce the number of recursive length-$n/2$ convolutions at each step from 4 to 3, thereby improving our recurrence to $T(n) = 3T(n/2) + \Theta(n)$, which solves to $T(n) = \Theta(n^\alpha)$. Please show how we can compute the 4 intermediate products $a_H * b_H$, $a_H * b_L$, $a_L * b_H$, and $a_L * b_L$ using only convolutions of length $n/2$ combined with $O(n)$ extra work performing addition and subtraction (i.e., we are trading one expensive convolution for a slightly larger number of cheaper additions and subtractions). [Solution]

Problem 464 (Equivalent Hardness of Squaring). Let $C(n)$ denote the best possible running time we can achieve for convolving two length-n vectors (equivalently, multiplying two degree-n polynomials or two n-digit integers). Using a similar trick as in the preceding problem, please show that it must take $\Omega(C(n))$ time to convolve a length-n vector with itself (equivalently, to square a degree-n polynomial or an n-digit integer). This shows that in terms of asymptotic running time, squaring is no easier than multiplication of two different elements. [Solution]

Problem 465 (Multiplying Polynomials and Integers of Different Length). Suppose we wish to convolve a small length-m vector a with a much larger vector b. Equivalently, suppose we wish to multiply polynomials of degrees $m-1$ and $n-1$, or that we wish to multiply integers having m and n digits. Treating the FFT as a black box that convolves two length-n sequences in $O(n \log n)$ time, please show how to use block convolution to compute $a * b$ in only $O(n \log m)$ time. [Solution]

Problem 466 (Zero-Delay Convolution). We have seen simple applications of block convolution — computing $a*b$ by partitioning a and b into blocks — in several of the preceding problems. If you feel comfortable with the principle of block convolution, this problem will put your skills to a proper test! Suppose we wish to compute the convolution $c = a * b$ of two length-n vectors a and b in an "online" setting, where a is given in advance but b provided one element at a time. If we only know $b_0 \ldots b_i$, we only have enough information to compute $c_0 \ldots c_i$. Suppose we are required to provide the value of c_i before we are allowed to see b_{i+1} (which in turn would allow us to compute c_{i+1}, thereby revealing b_{i+2} and so on). This problem is quite realistic, as it arises in signal-processing applications where we cannot tolerate any delay due to buffering (discussed shortly) and it also arises in solution of certain types of stochastic dynamic programming problems, which we highlight in a few pages in problem 471. Using the FFT as a black box that convolves two length-n sequences in $O(n \log n)$ please show how to perform zero-delay convolution of two length-n sequences in $O(n \log^2 n)$ total time (that is, $O(\log^2 n)$ amortized time for each successive element of b we introduce). As a hint, try partitioning a into successive blocks of sizes 1, 1, 2, 4, 8, 16, etc. [Solution]

23.1.2 Polynomial and Integer Division

Now that we know how to multiply degree-n polynomials and n-digit integers in $O(n \log n)$ time, we can leverage this result to build a division algorithm that also

runs in $O(n \log n)$ time[3]. We focus here on the case of *integer* division — the process of dividing two integers to obtain an integer quotient and an integer remainder. For example, integer division of 18 by 7 yields the quotient 2 and the remainder 4. If we instead want to find the first d digits after the decimal point in the fractional result $18/7 \approx 2.57143$, we can use integer division with d extra zeros appended to the numerator. For example, integer division of 180000 by 7 yields the quotient 25714. This same technique can be used to compute the reciprocal of an n-digit integer x: if we let z be a one followed by $n+d$ zeros, then integer division of z by x gives us an integer quotient containing the first d digits of $1/x$. Note that this discussion is not specific to decimal numbers — it applies to binary numbers, or numbers written in any constant base for that matter.

Polynomial division behaves much like integer division. For example, if $p(x) = x^3 - x^2 - 3x + 5$ and $q(x) = x^2 - 2x + 1$, then division of these two polynomials yields the quotient $x+3$ and the remainder $2x+2$. For simplicity, let us assume the leading coefficient of q is always one (this can be accomplished by dividing p and q by the leading coefficient of q). The quotient of p divided by q is now simply the number of times we can repeatedly subtract q from p until we obtain a polynomial of lower degree than q, which is the remainder. This is exactly the same as integer division: when we divide two integers x and y, the quotient tells us the number of times we can repeatedly subtract y away from x until we obtain a number less than y, which is the remainder. Polynomial division is fairly straightforward to implement in $O(n \log n)$ time using fast polynomial multiplication as a building block [Full details]. Integer division in $O(n \log n)$ time is similar, although its implementation requires attention to a few extra details due to carries. [Full details]

23.1.3 Polynomial Evaluation and Interpolation

Let $A(x) = a_0 + a_1 x + \ldots + a_{n-1}x^{n-1}$ be a polynomial of degree $n-1$. Recall that the n coefficients of A are completely determined if we specify the value of $A(x)$ at n arbitrary points $x_1 \ldots x_n$. For example, two points uniquely determine a line (a polynomial of degree one), three points uniquely determine a quadratic (a polynomial of degree two), and so on. Moreover, this fact holds even when we perform integer arithmetic modulo a prime; for a simple proof of this fact, refer back to problem 37. We refer to $a_0 \ldots a_{n-1}$ as the *coefficient* representation of $A(x)$ and we call the sequence of point-value pairs $(x_1, A(x_1)), \ldots (x_n, A(x_n))$ the *point-value* representation of $A(x)$. Both methods are equally suitable for specifying the polynomial $A(x)$. In fact, this "duality" of representation is a key ingredient in our ability to perform fast convolution with the FFT, as we shall see in a few pages. At the moment, let us consider how efficiently we can perform the operations of *evaluation* and *interpolation*, which convert back and forth between the coefficient and point-value representations of a polynomial.

Given A in coefficient form, suppose we wish to evaluate A at a particular point x. This is easy to do in $O(n)$ time using *Horner's rule*, by writing $A(x) = a_0 + x(a_1 + x(a_2 + x(a_3 \ldots a_{n-1}x)))$. By applying this process n times, we can evaluate $A(x)$ at n arbitrary points $x_1 \ldots x_n$ in $O(n^2)$ time. As it turns out, the FFT gives us an

[3]Remember that under our standard $\Theta(\log n)$-bit word size assumption, we can only perform $\Theta(\log n)$-digit integer division in constant time on a RAM.

even faster method for performing multi-point evaluation. Building upon the result that we can multiply two degree-n polynomials in $O(n \log n)$ time, we can compute $A(x_1) \dots A(x_n)$ in only $O(n \log^2 n)$ time with a clever recursive algorithm. [Further details]

Using the evaluation algorithm above, we can efficiently convert a polynomial from coefficient form to point-value form. The inverse of this process, recovering the n coefficients of a degree-$(n-1)$ polynomial from its value at n points, is known as interpolation. In problem 462, we showed how to interpolate a polynomial of degree $n-1$ given its $n-1$ roots in only $O(n \log^2 n)$ time[4] Interpolating from arbitrary values is slightly trickier, although it turns out we can still achieve the same performance guarantee. If we are told that $A(x_i) = y_i$ for $i = 1 \dots n$, then by expanding out $A(x)$ this gives us a system of n linear equations in n variables:

$$\begin{bmatrix} 1 & x_1 & x_1^2 & \dots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \dots & x_2^{n-1} \\ 1 & x_3 & x_3^2 & \dots & x_3^{n-1} \\ \vdots & & & & \vdots \\ 1 & x_n & x_n^2 & \dots & x_n^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix}.$$

The most obvious approach for performing interpolation is therefore to try and solve this system for $a_0 \dots a_{n-1}$, and we can do this in $O(M(n))$ time, where $M(n)$ denotes the time required to multiply two $n \times n$ matrices (as we will see in Chapter 24, the best known running time for matrix multiplication is currently around $O(n^{2.376})$). However, if we are using real number arithmetic, then this process tends to be rather "ill conditioned" from a numerical standpoint (see also Chapter 24) since it involves a matrix whose entries differ tremendously in magnitude; as a result, we could end up with a substantial amount of round-off error. If our problem involves integer arithmetic modulo a prime number, however, then this is no longer an issue, since we can stay within the RAM model and use exact arithmetic on small integers (this is one of the benefits of using integer arithmetic whenever possible). As it turns out, however, solving a linear system is actually not the best approach for interpolation. We can also perform interpolation in a bottom-up inductive fashion in only $O(n^2)$ time [Details]. Using fast convolution via the FFT as a building block, we can further improve this running time to $O(n \log^2 n)$. [Details].

Interpolation of the n coefficients of a polynomial from n points is actually a special case of a more general process known as *Chinese remaindering*. In Chapter 25, we will study this problem in greater detail, and we will discuss how the fast interpolation algorithms above generalize naturally in the case of Chinese remaindering.

23.1.4 Special Matrices

It typically takes us $O(n^2)$ time to multiply an $n \times n$ matrix by a length-n vector, and $O(M(n))$ time to solve an $n \times n$ linear system of equations (Chapter 24). There are many special classes of matrices, however, for which we can improve on these results by using fast convolution.

[4] Normally we need to know the value of a degree-$(n-1)$ polynomial at n points to interpolate it. However, when we interpolate from roots, we typically assume the leading coefficient of our polynomial is one, so only $n-1$ roots suffice to determine the remaining $n-1$ coefficients.

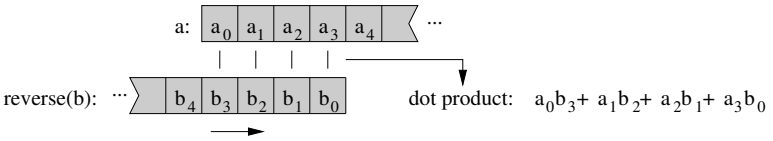

FIGURE 23.2: Visualizing convolution as a sequence of "shifted dot products". Here, we slide b (in reverse) across a, and at each step we compute a dot product to obtain the next term of $a * b$.

Vandermonde Matrices. An $n \times n$ *Vandermonde* matrix $V(x_1, \ldots, x_n)$ is specified by n values $x_1 \ldots x_n$ as follows:

$$V(x_1, \ldots, x_n) = \begin{bmatrix} 1 & x_1 & x_1^2 & \ldots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \ldots & x_2^{n-1} \\ 1 & x_3 & x_3^2 & \ldots & x_3^{n-1} \\ \vdots & & & & \vdots \\ 1 & x_n & x_n^2 & \ldots & x_n^{n-1} \end{bmatrix}.$$

Having just discussed evaluation and interpolation, this matrix should look quite familiar: multiplication of a column vector $a = a_0 \ldots a_{n-1}$ by $V(x_1, \ldots, x_n)$ corresponds to the evaluation of a polynomial with coefficient vector a at the points $x_1 \ldots x_n$, and if we solve the linear system $V(x_1, \ldots, x_n)a = y$ for the vector a, this corresponds to interpolation. Since we just learned how to perform evaluation and interpolation in $O(n \log^2 n)$ time using fast convolution, we can therefore perform matrix-vector multiplication and solve linear systems involving Vandermonde matrices in the same amount of time.

One can show that the determinant of $V(x_1, \ldots, x_n)$ is equal to $\prod_{i<j}(x_i - x_j)$, a quantity known as the *discriminant* of $x_1 \ldots x_n$. Using fast convolution, one can compute the absolute value of this quantity in only $O(n \log^2 n)$ time [Details]. You may recall we briefly mentioned back in Chapter 3 that the element uniqueness problem on $x_1 \ldots x_n$ could be solved by checking if the discriminant is zero.

Toeplitz and Circulant Matrices. A *Toeplitz* matrix is a matrix whose diagonals consist of equal elements. Such a matrix is determined entirely by its topmost row and leftmost column:

$$A = \begin{bmatrix} a_0 & a_1 & a_2 & \ldots & a_{n-1} \\ a_{-1} & a_0 & a_1 & \ldots & a_{n-2} \\ a_{-2} & a_{-1} & a_0 & \ldots & a_{n-3} \\ \vdots & & & & \vdots \\ a_{-(n-1)} & a_{-(n-2)} & a_{-(n-3)} & \ldots & a_0 \end{bmatrix}$$

A special case of the Toeplitz matrix is the *circulant* matrix, where each row is obtained by a cyclic shift of the preceding row (so $a_{-i} = a_{n-i}$). Toeplitz and circulant matrices and their relatives arise in a variety of situations in practice; an example is shown in the next problem. It is quite simple to use convolution to perform matrix-vector multiplication against these special matrices in $O(n \log n)$ time. [Details]

Problem 467 (Equally-Spaced Point Charges). Consider the following physics problem: we are given n equally-spaced point charges $q_1, q_2, \ldots, q_n$ at positions $1, 2, \ldots, n$ on a number line, and we would like to compute the resulting electric potential $\phi_1, \phi_2, \ldots, \phi_n$ at these same points. The electric potential ϕ_j at position j is given by

$$\phi_j = \sum_{i \neq j} \frac{q_i}{|i - j|}.$$

That is, the potential at j due to i depends directly on q_i and inversely on the distance between j and i. It takes $\Theta(n^2)$ time to apply the formula above to compute $\phi_1 \ldots \phi_n$. See if you can show how to compute $\phi_1 \ldots \phi_n$ in only $\Theta(n \log n)$ time. [Solution]

23.1.5 Signal Processing

Streams of digital information (e.g., audio and video signals) are nothing more than long sequences of numbers. By processing these sequences in an appropriate fashion, we can perform all sorts of useful functions, such as data compression, removal of noise, filtering out certain frequencies, and many more. Most devices performing digital signal processing (DSP) do so using convolution. For example, if you tell your stereo to reduce the volume of high frequencies, it typically accomplishes this by convolving an audio signal with an appropriate short sequence that acts as a "low pass" filter. If this sequence has length n, then we can perform this convolution in $O(\log n)$ time per input element by buffering the input signal into length-n blocks, or in $O(\log^2 n)$ time per input element using zero-delay convolution, which avoids the need for buffering [Further details].

When we study the FFT shortly, we will see that one of its common uses (aside from convolution) is to transform between the *time-domain* and *frequency-domain* representations of a signal, thereby decomposing a signal into its constituent sinusoidal frequencies. This transformation is known as the *Discrete Fourier Transform* (DFT), and we will learn more about it once we have described the DFT in greater detail.

23.1.6 Pattern Matching

Consider the fundamental string matching problem from Chapter 9: find all occurrences of a pattern $P[1 \ldots m]$ within a longer text $T[1 \ldots n]$. Let us assume the pattern and text are binary strings; if not, we apply the transformation from problem **??** to reduce our problem to the binary case, at an expense of an extra $O(\log \sigma)$ factor in our ultimate running time, where $\sigma \leq m + n = O(n)$ denotes the number of distinct characters present in our strings.

What does pattern matching have to do with convolution? Quite a bit, if we look at convolution the right way. As shown in Figure 23.2, we can obtain the successive elements in $a * b$ by sliding b (in reverse) across a and taking a dot product at each shift. Since the dot product of two binary vectors tells us the number of locations in which both vectors have a "one" bit in common, convolution is now starting to look quite similar to pattern matching! We leave the rest of this application as an exercise below, where we show how convolution-based pattern matching can handle "wildcard" characters — something our traditional string matching algorithms from

Chapter 9 cannot do well.

Problem 468 (Pattern Matching with Wildcard Characters). Consider the pattern matching problem above, where we wish to find all occurrences of a binary pattern $P[1 \ldots m]$ within a longer binary text $T[1 \ldots n]$.

(a) Please show how to use the results of two convolutions to solve this problem in $O(n \log n)$ time. As before, please assume we can use the FFT to convolve two length-n vectors in $O(n \log n)$ time. [Solution]

(b) Now suppose some of the characters in the pattern and/or the text are "wildcard" characters. That is, each character in P and T comes from the alphabet $\{0, 1, ?\}$, where the '?' character is allowed to match any other character (e.g., the string "1??0?1" would match both "101001" and "100011"). Please show how to modify your algorithm above to find all matches of P in T in $O(n \log n)$ time. [Solution]

(c) Please extend your algorithm from the previous part to handle the two-dimensional case, where we want to find all matches of a rectangular binary matrix P within a larger rectangular binary matrix T (both of which might contain wildcard characters). If T has n total entries, please show how to solve this problem in $O(n \log n)$ time. [Solution]

23.1.7 Counting and Generating Functions

Suppose you would like to assemble a basket containing 7 total pieces of fruit that contains: an odd number of apples, at least 2 bananas, and no more than 3 pears. How many different valid combinations exist? This problem and many like it can be easily solved using convolution. Let $A(x) = x + x^3 + x^5 + x^7$ be a polynomial in which the coefficient of x^i is one if we are allowed to place i apples in the basket, and zero otherwise. Similarly, we define polynomials $B(x) = x^2 + x^3 + x^4 + x^5 + x^6 + x^7$ and $P(x) = 1 + x + x^2 + x^3$ for bananas and pairs. If you think carefully about the formula for convolution, you will see that the coefficient of x^7 in the product polynomial $A(x)B(x)P(x)$ tells us the answer we seek (in this case, 8 different baskets).

As another example, suppose you want to know how many different subsets of size 7 you can choose from a larger set of 10 elements. The answer is of course the binomial coefficient $\binom{10}{7}$ (that is, the coefficient of x^7 in $(1 + x)^{10}$), and we can derive this result using convolution. Just as above, suppose we can select zero or one copies of each of 10 different types of fruit, and we would like to know how many different 7-fruit baskets we can assemble. Each fruit corresponds to the polynomial $(1+x)$, and by multiplying 10 of these together we find our answer in the coefficient of x^7.

Convolution-based counting using polynomial multiplication is perhaps the main idea underlying *generating functions*, if you happen to have studied these in a past discrete mathematics course. The only additional layer of complexity in this case is the fact that we often deal with infinite polynomials, which are represented in a more convenient form by closed-form algebraic functions. For example, the infinite polynomial $P(x) = 1 + x^2 + x^4 + x^6 + \ldots$ can be regarded as a geometric series, which sums to $1/(1 - x^2)$ for $|x| < 1$. We can therefore use the simpler "generating function" $1/(1 - x^2)$ to represent $P(x)$ when multiplying by other polynomials or

generating functions, after which we can recover the coefficients in the product using a bit of arithmetic.

Problem 469 (Sum and Difference Histograms). You are given as input a set $S \subseteq \{1, 2, 3, \ldots, n\}$. Using the FFT as a black box that convolves two length-n sequences in $O(n \log n)$ time, please give an $O(n \log n)$ algorithm that counts the number of different two-element subsets of S whose elements sum to k, for every possible value of k. What about three-element subsets, etc.? Next, modify your algorithm so it computes in $O(n \log n)$ time the number of different two-element subsets of S differing by k, for every possible value of k. [Solution]

Problem 470 (Median Node-to-Node Distance in a Tree.). Consider the problem of computing the median distance among all $\binom{n}{2}$ pairs of nodes in a free tree (we consider the related, easier problem of average distance in problem 323). It is fairly easy to solve this problem in $\Theta(n^2)$ time by listing out all $\binom{n}{2}$ node-to-node distances and applying a linear-time median-finding algorithm. However, one can do much better, as we will discover in this problem.

(a) Using convolution and the fact that trees can be decomposed in a balanced fashion by removal of separator nodes (Section 8.3.4), please show how to solve this problem in $O(n \log^3 n)$ time. A randomized algorithm is fine. As a hint, first show how to quickly count the number of pairs of nodes in a tree whose distance from each-other lies in a particular range $[a, b]$. [Solution]

(b) For an even faster solution (which even works in the case where our tree has non-unit edge lengths), see if you can use the result of problem 58 to achieve $O(n \log^2 n)$ time (a randomized algorithm is fine here as well). It turns out that $O(n \log n)$ is also possible; see the endnotes for references. [Solution]

23.1.8 Probability Distributions of Sums

Convolution plays an important role in the study of probability. Let A be a non-negative integer random variable taking the value k with probability a_k, so we can represent the probability distribution of A as a vector $a = a_0, a_1, a_2, \ldots$ of possibly infinite length. By treating this vector as the sequence of coefficients in a polynomial $A(x) = a_0 + a_1 x + a_2 x^2, \ldots$, we form what is called the *probability generating function* of A, or sometimes the *z-transform* of A (in which case we often use the variable z instead of x). Letting $b = b_0, b_1, b_2, \ldots$ denote the probability distribution of random variable B, what is $\mathbf{Pr}[A + B = 3]$? By summing up the probabilities of all the ways to obtain a sum of 3 (i.e., $A = 0$ and $B = 3$, $A = 1$ and $B = 2$, and so on), we find that $\mathbf{Pr}[A + B = 3] = a_0 b_3 + a_1 b_2 + a_2 b_1 + a_3 b_0$. From this observation, we find that the probability distribution of a random variable $C = A + B$ is exactly $a * b$. In terms of polynomial multiplication, $C(x) = A(x)B(x)$.

One simple example of this principle is the following: recall that a binomial random variable is obtained by summing up a collection of independent Bernoulli (indicator) random variables: $X = X_1 + \ldots X_k$. If each indicator variable X_i takes the value 1 with probability p, we can represent its distribution by the polynomial $(1-p)+px$. To obtain the distribution for X, we simply compute $[(1-p)+px]^n$, in which the coefficient of x^k gives us the familiar result $\mathbf{Pr}[X = k] = p^k(1-p)^{n-k}\binom{n}{k}$.

Problem 471 (Stochastic Dynamic Programming). In this problem we inves-

tigate the use of the zero-delay convolution technique from problem 466 to develop efficient solutions to dynamic programming problems involving probability distributions.

(a) Suppose you are given a length-n vector $x_0 \ldots x_{n-1}$ describing the probability distribution of some discrete random variable X, so $x_i = \mathbf{Pr}[X = i]$. Consider the problem of computing the expected number of samples we can draw from X's distribution before their sum reaches a specified target T. Please show how to formulate this problem as a dynamic program that has an obvious $O(T^2)$ solution. Then, try to use zero-delay convolution to improve the running time to $O(T \log^2 T)$. [Solution]

(b) **Stochastic Replenishment.** Suppose we need to keep a machine operating for T units time, where the machine depends on some critical part (e.g., a light bulb) to run. We have n different models available for this part from which to choose, where the ith model has a cost c_i and a lifetime t_i (an integer-value random variable whose distribution is provided to us in the form of a length-T vector). At time $t = 0$, we must select one of the n models to purchase; we then wait until the part wears out, and based on the amount of time remaining we select a new model to install (possibly the same as before). This process continues until we reach time $t = T$. Our goal is to compute a "policy" for purchasing parts that minimizes our expected total cost. Please show how to set up this problem as a dynamic program with an obvious $O(nT^2)$ solution, then show how to solve it in $O(nT \log^2 T)$ time using zero-delay convolution. [Solution]

(c) **The Stochastic Shortest Path Problem.** Automobile transportation networks (and several other types of networks) typically exhibit uncertainty in the transit times along edges. For example, it is usually fast to take the highway between two nodes, but there may be a small probability that the highway will be jammed and that the travel time along this edge will be much larger. To model this, we let the travel time along each edge in our graph be a positive integer random variable, each of whose distributions you are given as input as a length-T vector. Several different stochastic "shortest path" objectives have been studied in the literature in this context. A common objective is to designate a source node s and destination node d, and to ask for an $s \rightsquigarrow d$ path of minimum expected travel time. However, this problem is easy, since we can solve it using a traditional shortest path algorithm by simply replacing each edge distribution with its expectation. A much more interesting objective is the following: given an upper limit on our travel time T, we would like to maximize our probability of arriving at d within T units of time. We do not need to select the entire $s \rightsquigarrow d$ path in advance, however. Just as in the previous problem, we can choose the path "adaptively" one edge at a time, based on our current location and the amount of time remaining before the deadline. Please give an $O(mT^2)$ dynamic programming algorithm that solves this problem, and then show how to improve its running time to $O(mT \log^2 T)$ using zero-delay convolution. What if we place costs on edges and ask for a path of minimum expected cost, provided that the cost of arriving after T units of time is penalized by a large pre-specified amount M? [Solution]

23.2 Fast Convolution with the FFT

Consider two polynomials $A(x) = a_0 + a_1 x + \ldots + a_{n-1}x^{n-1}$ and $B(x) = b_0 + b_1 x + \ldots + b_{n-1}x^{n-1}$ with coefficient sequences $a = a_0 \ldots a_{n-1}$ and $b = b_0 \ldots b_{n-1}$. Since convolution and polynomial multiplication are the same thing, $c = a * b$ is given by the sequence of coefficients for the product polynomial $C(x) = A(x)B(x)$. Therefore, let us think in terms of polynomial multiplication from now on.

Recall that a polynomial of degree $n - 1$ can be represented equally well in two

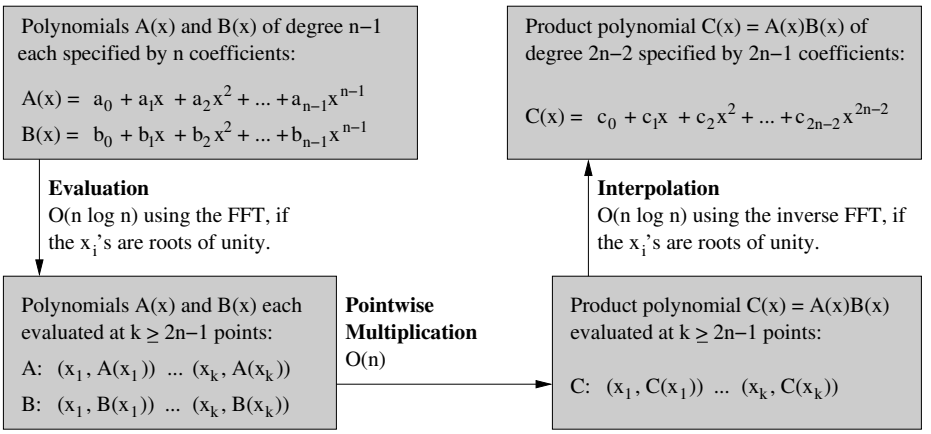

FIGURE 23.3: The process of multiplying two polynomials using the FFT.

different ways: as a sequence of n coefficients or as its value at n different points. Moreover, polynomial multiplication is a trivial $O(n)$ process if we use the second representation! The product $C(x) = A(x)B(x)$ is a polynomial of degree $2n-2$, so it is uniquely specified if we know its value at any set of $k \geq 2n-1$ points $x_1 \ldots x_k$. If we know $A(x_1) \ldots A(x_k)$ and $B(x_1) \ldots B(x_k)$, we can simply multiply these corresponding values together pairwise to obtain the point-value representation of $C(x)$ (i.e., $C(x_i) = A(x_i)B(x_i)$).

Since convolution is much simpler in the "point-value" domain than in the "coefficient" domain, this suggests that perhaps we should perform convolution using the pathway described in Figure 23.3: we first evaluate A and B at $k \geq 2n-1$ points $x_1 \ldots x_k$ to convert to point-value representation (we will actually set k to the smallest power of two that is at least as large as $2n-1$), perform pointwise multiplication in $O(n)$ time, and then convert the result back to coefficient form using interpolation to obtain the coefficient form of the product polynomial C. We learned earlier how to perform evaluation and interpolation in $O(n \log^2 n)$ time, but these methods are not applicable here because they are based on fast polynomial multiplication — the problem we are currently trying to solve! Fortunately, we still have one trick up our sleeves: we get to choose the k points at which we evaluate/interpolate our polynomials. As we will now see, if we choose these k points to be *kth roots of unity*, then we will be able to perform evaluation and interpolation in only $O(n \log n)$ time, giving us a total running time of $O(n \log n)$ for the entire convolution process.

23.2.1 The Discrete Fourier Transform

The *Discrete Fourier Transform* (DFT) is the process of evaluating a polynomial of degree $n-1$ (represented by a n coefficients) at n special points known as the nth roots of unity. We first define what we mean by an nth root of unity, then we show how the FFT algorithm can execute a DFT and its inverse (the interpolation of a polynomial at the nth roots of unity) both in only $O(n \log n)$ time.

An *nth root of unity* is a number ω such that $\omega^n = 1$. For example, $+1$, -1, $+i$, and $-i$ are all 4th roots of unity if we are performing arithmetic over complex numbers (here i denotes $\sqrt{-1}$), and the integers in the set $\{1, 4, 8, 16\}$ are all 4th roots of unity if we are performing integer arithmetic modulo the prime 17. Since nth roots of unity are roots of the polynomial $\omega^n - 1 = 0$, there can be at most n such values, owing to the fact that a polynomial of degree n has at most n roots (recall that this is true in any algebraic field, including the complex numbers or arithmetic modulo a prime). If we are performing arithmetic over the complex numbers, the nth roots of unity are given by $e^{2\pi ik/n} = \cos\frac{2\pi k}{n} + i\sin\frac{2\pi k}{n}$ for $k = 0 \ldots n-1$. If we are performing integer arithmetic modulo a prime p, then we can also find n distinct nth roots of unity as long as p is sufficiently large.

A *primitive* nth root of unity is a nth root of unity that is not an kth root of unity for any $k < n$. Stated differently, if ω is a primitive nth root of unity, then n is the smallest power to which we can raise ω to obtain 1. For example, the complex numbers $+1$ and -1 are not primitive 4th roots of unity, since they are also square roots of unity, whereas $+i$ and $-i$ are both primitive 4th roots of unity. Likewise, in integer arithmetic modulo 17, both 4 and 8 are primitive 4th roots of unity, but 1 and 16 are not since they are also square roots of unity (note that 16 plays the role of "-1" here). Primitive nth roots of unity are useful since they generate all of the nth roots of unity. Letting ω_n denote a primitive nth root of unity, the set $\{\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}\}$ is the set of all nth roots of unity. In terms of complex numbers, $\omega_n = e^{2\pi i/n} = cos\frac{2\pi}{n} + i\sin\frac{2\pi}{n}$ is a primitive nth root of unity. In terms of arithmetic modulo a suitably large[5] prime p, we can find a primitive nth root of unity in an efficient manner using a bit of number theory. [Further details]

Let $A(x)$ be a polynomial of degree $n-1$ with coefficient vector $a = a_0 \ldots a_{n-1}$. The DFT of a is the length-n vector we obtain by evaluating $A(x)$ at the n roots of unity $x = \omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$. Recall that we can describe evaluation of a polynomial in the context of linear algebra using Vandermonde matrices. This gives us another simple way to describe the DFT: it is the process of multiplying a vector a by special Vandermonde matrix called the *Fourier* matrix,

$$F_n = V(\omega_n^0, \ldots, \omega_n^{n-1}) = \begin{bmatrix} \omega_n^{0\cdot 0} & \omega_n^{0\cdot 1} & \omega_n^{0\cdot 2} & \ldots & \omega_n^{0\cdot(n-1)} \\ \omega_n^{1\cdot 0} & \omega_n^{1\cdot 1} & \omega_n^{1\cdot 2} & \ldots & \omega_n^{1\cdot(n-1)} \\ \omega_n^{2\cdot 0} & \omega_n^{2\cdot 1} & \omega_n^{2\cdot 2} & \ldots & \omega_n^{2\cdot(n-1)} \\ \vdots & & & & \vdots \\ \omega_n^{(n-1)\cdot 0} & \omega_n^{(n-1)\cdot 1} & \omega_n^{(n-1)\cdot 2} & \ldots & \omega_n^{(n-1)\cdot(n-1)} \end{bmatrix},$$

whose (i, j) entry is ω_n^{ij} for $0 \le i, j \le n-1$. Since the DFT is therefore nothing more than a linear transformation, we see that the DFT of the sum of two vectors is equal to the sum of the DFTs of the vectors, the DFT of k times a vector is

[5] It is important that a prime described by $O(\log n)$ bits will suffice, since this is small enough to fit into a single word in the RAM model, allowing for fundamental integer arithmetic operations to run in $O(1)$ time. In fact, there is another subtlety we should address when using integer arithmetic modulo a prime while performing convolution: note that the elements in the output vector of a convolution can be quite a bit larger that the elements in its input — if our input vectors contain numbers of magnitude at most B, then the output vector will contain numbers of magnitude at most nB^2. Hence, we need to ensure that our prime is larger than nB^2 to prevent unwanted overflow and "wrap-around". Let us therefore assume that the elements of our output, as well as the prime we use, are all sufficiently small so as to fit into an $O(\log n)$-bit word.

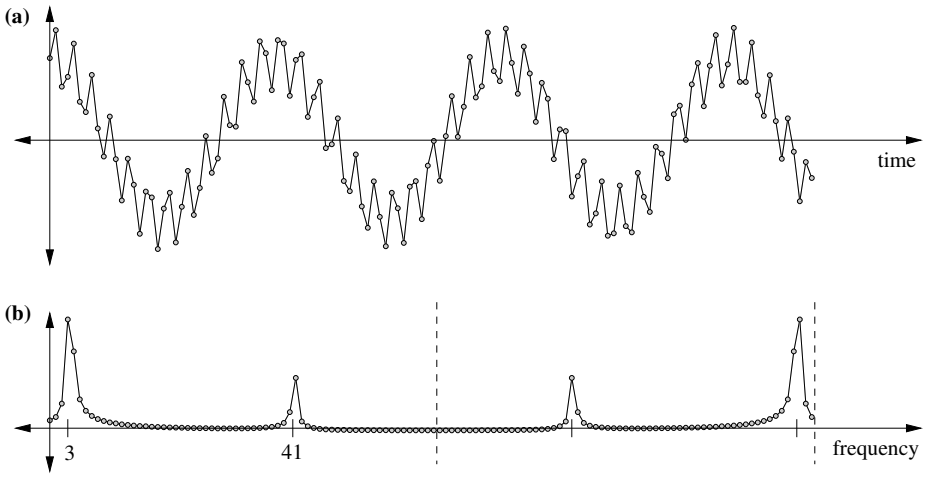

FIGURE 23.4: The graph in (a) shows an example signal $x = x_0 \ldots x_{127}$ constructed by adding a high-frequency sine wave to a low-frequency cosine wave. The graph in (b) shows $y = y_0 \ldots y_{127}$, the DFT of x. The elements of y are complex numbers, so we have plotted their magnitudes $|y_0| \ldots |y_{127}|$, as these describe the total contribution of the frequencies present in x. The first peak between 3 and 4 corresponds to the fact that roughly 3.4 periods of the low-frequency cosine wave appear in our signal, and the second smaller peak around 41 corresponds to the fact that there are 40.7 periods of the smaller sine wave. Furthermore, by looking at the phase of the complex numbers y_i at these peaks, we can deduce that the first component is close to a cosine while the second is closer to a sine. Note finally the symmetry in (b), which always occurs when we take the DFT of a real-valued signal. You can think of the second symmetric half of (b) as actually representing "negative" frequencies; in fact, we could have drawn this half to the left of the first half to emphasize this point.

equal to k times the DFT of the vector, and so on. Linear algebra enthusiasts will also be happy to hear to that the rows in F_n are orthogonal, so multiplication of a vector by F_n can also be interpreted as the projection of that vector into a new basis (this also ties into our discussion below on signal processing, since if we use complex arithmetic, this new basis can be characterized in terms of sinusoidal waves of different frequencies).

The inverse DFT (IDFT) is the process of interpolating the coefficients of A given the values $A(\omega_n^0) \ldots A(\omega_n^{n-1})$. Remarkably, the IDFT is essentially the same process as the DFT. If b is the DFT of a length-n vector a, then we can recover a by taking the DFT of b and dividing the resulting vector by n, where we use ω_n^{-1} in place of ω_n in our DFT. Note that ω_n^{-1} is also a primitive nth root of unity (if we are using integer arithmetic modulo a prime p, then ω_n^{-1} denotes the unique multiplicative inverse of ω_n modulo p, which we can compute with Euclid's algorithm in Chapter 25). In terms of linear algebra, the IDFT is equivalent to multiplication by the inverse Fourier matrix $F_n^{-1} = \frac{1}{n} V(\omega_n^0, \omega_n^{-1}, \omega_n^{-2}, \ldots, \omega_n^{-(n-1)})$. [Mathematical justification]

Signal Processing Revisited. The DFT and IDFT are particularly prominent

operations in the realm of signal processing. Recall that the operations of evaluation and interpolation allow us to convert between the coefficient and point-value representations of a polynomial. For the special case where we are evaluating and interpolating at roots of unity, the DFT and IDFT are commonly viewed as a means of converting between the "time domain" and "frequency domain" representations of a signal, where the frequency domain representation gives us a decomposition of our signal into a weighted sum of sinusoidal waves of different frequencies. Let $A(x)$ be a polynomial with coefficient vector a of length n, where a is the time-series representation of a signal of interest. By evaluating $A(x)$ at $x = e^{2\pi ik/n}$, we obtain a complex number that describes the contribution of frequency k when we decompose the signal a into a sum of sinusoidal waves: its magnitude is the amplitude of the contribution at frequency k, and its angle describes the phase of this contribution[6] Since the DFT evaluates $A(x)$ at many different frequencies, it gives us a vector of complex numbers that describes the frequency composition, or "spectrum" of a. An example is shown in Figure 23.4. [Further mathematical insight].

Just as convolution of two vectors simplifies to pointwise multiplication when we convert to point-value representation, the convolution of two signals corresponds to pointwise multiplication in their frequency-domain representations. For example, if we were to convolve a signal against the signal shown in Figure 23.4(a), this would end up canceling out all of the frequencies in our signal except for the two present in Figure 23.4(b). This fact helps us design digital filters of different sorts when performing signal processing. See the endnotes for further references to applications of the DFT.

23.2.2 The Fast Fourier Transform Algorithm

We now discuss the details of the Fast Fourier Transform (FFT) algorithm. The FFT is a simple divide and conquer algorithm for performing a DFT (and also the IDFT, as discussed above) in $O(n \log n)$ time. We can implement the FFT using either complex numbers or integers modulo a prime p. In the case of complex numbers, our model of computation is the real RAM, and in the integer case our model is the standard RAM. If we use complex numbers, we must be slightly careful in practice with numerical round-off issues, since we are manipulating fractional values (cosines and sines). This could result in our trying to convolve two integer-valued vectors and obtaining a vector with entries like 3.99999 instead of 4. If we are using the FFT as part of the process of convolving two integer vectors (for example, if we are multiplying two large integers), then it is probably best to use arithmetic modulo a prime, so we can stay within the RAM model and use only exact integer arithmetic. For the remainder of this section, let us assume n is a power of two, which we can easily achieve by zero-padding the vector whose DFT we are computing.

One of the key insights behind the FFT is the following: if we take the n members of the set $\{\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}\}$ of nth roots and square them all, we obtain the $n/2$ elements in the set $\{\omega_{n/2}^0, \ldots, \omega_{n/2}^{n/2-1}\}$ of all $(n/2)$th roots of unity. For example, if

[6] The magnitude of a complex number $x = a + bi$ is $|x| = (a^2 + b^2)^{1/2}$, and its angle (phase) is $\theta = \tan^{-1} b/a$ (these two numbers give the polar coordinates of the point (a, b) in the complex plane).

we square the 4th roots of unity $\{+1, -1, +i, -i\}$, we get precisely the set $\{+1, -1\}$ of square roots of unity. This follows easily from the fact that each $(n/2)$th root of unity has two square roots in the set of nth roots of unity.

Recall now that the purpose of the FFT is to evaluate a polynomial $A(x) = a_0 + a_1x + a_2x^2 + \ldots a_{n-1}x^{n-1}$ at all the nth roots of unity. To do this, we decompose the coefficient vector $a_0, a_1, a_2, \ldots, a_{n-1}$ into its even entries $a_0, a_2, a_4, \ldots, a_{n-2}$, and its odd entries $a_1, a_3, a_5, \ldots, a_{n-1}$. By constructing half-sized polynomials with these new coefficient vectors, we can cleverly decompose $A(x)$ as follows:

$$A(x) = A_{even}(x^2) + xA_{odd}(x^2),$$

where

$$\begin{aligned} A_{even}(x) &= a_0 + a_2x + a_4x^2 + a_6x^3 + \ldots + a_{n-2}x^{n/2-1}, \\ A_{odd}(x) &= a_1 + a_3x + a_5x^2 + a_7x^3 + \ldots + a_{n-1}x^{n/2-1}. \end{aligned}$$

Now, it is easy to see that in order to evaluate $A(x)$ at all the nth roots of unity, this is equivalent to evaluating the two half-sized polynomials $A_{even}(x)$ and $A_{odd}(x)$ at all of the squares of the nth roots of unity, and this is nothing more than the set of all $(n/2)$th roots of unity. We have therefore decomposed the problem of evaluating a polynomial with n coefficients at the n different nth roots of unity into two subproblems where we evaluate a polynomial having $n/2$ coefficients at the $n/2$ different $(n/2)$th roots of unity (with a linear amount of extra work involved in the decomposition). This allows us to describe the running time of our algorithm with the recurrence $T(n) = 2T(n/2) + \Theta(n)$, which solves to $T(n) = \Theta(n \log n)$. [Small example of the FFT in action].

