Calvin: *Here's another math problem I can't figure out. What's 9+4?*

Hobbes: *Ooh, that's a tricky one. You have to use calculus and imaginary numbers for this.*

Calvin: *IMAGINARY NUMBERS?!*

Hobbes: *You know, eleventeen, thirty-twelve, and all those. It's a little confusing at first.*

Calvin: *How did YOU learn all this? You've never even gone to school!*

Hobbes: *Instinct. Tigers are born with it.*

— "Calvin and Hobbes" (January 6, 1998)

It needs evaluation
So let the games begin
A heinous crime, a show of force
A murder would be nice, of course

— "Bad Horse's Letter", *Dr. Horrible's Sing-Along Blog* (2008)

★A Fast Fourier Transforms

A.1 Polynomials

In this lecture we'll talk about algorithms for manipulating *polynomials*: functions of one variable built from additions subtractions, and multiplications (but no divisions). The most common representation for a polynomial $p(x)$ is as a sum of weighted powers of a variable x:

$$p(x) = \sum_{j=0}^{n} a_j x^j.$$

The numbers a_j are called *coefficients*. The *degree* of the polynomial is the largest power of x; in the example above, the degree is n. Any polynomial of degree n can be specified by a sequence of $n+1$ coefficients. Some of these coefficients may be zero, but not the nth coefficient, because otherwise the degree would be less than n.

Here are three of the most common operations that are performed with polynomials:

- **Evaluate:** Give a polynomial p and a number x, compute the number $p(x)$.
- **Add:** Give two polynomials p and q, compute a polynomial $r = p + q$, so that $r(x) = p(x) + q(x)$ for all x. If p and q both have degree n, then their sum $p + q$ also has degree n.
- **Multiply:** Give two polynomials p and q, compute a polynomial $r = p \cdot q$, so that $r(x) = p(x) \cdot q(x)$ for all x. If p and q both have degree n, then their product $p \cdot q$ has degree $2n$.

Suppose we represent a polynomial of degree n as an array of $n+1$ coefficients $P[0..n]$, where $P[j]$ is the coefficient of the x^j term. We learned simple algorithms for all three of these operations in high-school algebra:

```
EVALUATE($P[0..n], x$):
    $X \leftarrow 1$    $\langle\langle X = x^j \rangle\rangle$
    $y \leftarrow 0$
    for $j \leftarrow 0$ to $n$
        $y \leftarrow y + P[j] \cdot X$
        $X \leftarrow X \cdot x$
    return $y$
```

```
ADD($P[0..n], Q[0..n]$):
    for $j \leftarrow 0$ to $n$
        $R[j] \leftarrow P[j] + Q[j]$
    return $R[0..n]$
```

```
MULTIPLY($P[0..n], Q[0..m]$):
    for $j \leftarrow 0$ to $n + m$
        $R[j] \leftarrow 0$
    for $j \leftarrow 0$ to $n$
        for $k \leftarrow 0$ to $m$
            $R[j+k] \leftarrow R[j+k] + P[j] \cdot Q[k]$
    return $R[0..n+m]$
```

EVALUATE uses $O(n)$ arithmetic operations.[1] This is the best we can hope for, but we can cut the number of multiplications in half using *Horner's rule*:

$$p(x) = a_0 + x(a_1 + x(a_2 + \ldots + xa_n)).$$

```
HORNER($P[0..n], x$):
    $y \leftarrow P[n]$
    for $i \leftarrow n-1$ downto 0
        $y \leftarrow x \cdot y + P[i]$
    return $y$
```

The addition algorithm also runs in $O(n)$ time, and this is clearly the best we can do.

The multiplication algorithm, however, runs in $O(n^2)$ time. In the previous lecture, we saw a divide and conquer algorithm (due to Karatsuba) for multiplying two n-bit integers in only $O(n^{\lg 3})$ steps; precisely the same algorithm can be applied here. Even cleverer divide-and-conquer strategies lead to multiplication algorithms whose running times are arbitrarily close to linear—$O(n^{1+\varepsilon})$ for your favorite value $e > 0$—but with great cleverness comes great confusion. These algorithms are difficult to understand, even more difficult to implement correctly, and not worth the trouble in practice thanks to large constant factors.

A.2 Alternate Representations

Part of what makes multiplication so much harder than the other two operations is our input representation. Coefficients vectors are the most common representation for polynomials, but there are at least two other useful representations.

A.2.1 Roots

The Fundamental Theorem of Algebra states that every polynomial p of degree n has exactly n *roots* $r_1, r_2, \ldots r_n$ such that $p(r_j) = 0$ for all j. Some of these roots may be irrational; some of these roots may by complex; and some of these roots may be repeated. Despite these complications, this theorem implies a unique representation of any polynomial of the form

$$p(x) = s \prod_{j=1}^{n} (x - r_j)$$

where the r_j's are the roots and s is a scale factor. Once again, to represent a polynomial of degree n, we need a list of $n+1$ numbers: one scale factor and n roots.

Given a polynomial in this root representation, we can clearly evaluate it in $O(n)$ time. Given two polynomials in root representation, we can easily multiply them in $O(n)$ time by multiplying their scale factors and just concatenating the two root sequences.

Unfortunately, if we want to add two polynomials in root representation, we're pretty much out of luck. There's essentially *no* correlation between the roots of p, the roots of q, and the roots of $p+q$. We could convert the polynomials to the more familiar coefficient representation first—this takes $O(n^2)$

[1] I'm going to assume in this lecture that each arithmetic operation takes $O(1)$ time. This may not be true in practice; in fact, one of the most powerful applications of FFTs is fast *integer* multiplication. One of the fastest integer multiplication algorithms, due to Schönhage and Strassen, multiplies two n-bit binary numbers using $O(n \log n \log\log n \log\log\log n \log\log\log\log n \cdots)$ bit operations. The algorithm uses an n-element Fast Fourier Transform, which requires several $O(\log n)$-nit integer multiplications. These smaller multiplications are carried out recursively (of course!), which leads to the cascade of logs in the running time. Needless to say, this is a can of worms.

time using the high-school algorithms—but there's no easy way to convert the answer back. In fact, for most polynomials of degree 5 or more in coefficient form, it's *impossible* to compute roots exactly.[2]

A.2.2 Samples

Our third representation for polynomials comes from a different consequence of the Fundamental Theorem of Algebra. Given a list of $n+1$ pairs $\{(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)\}$, there is *exactly one* polynomial p of degree n such that $p(x_j) = y_j$ for all j. This is just a generalization of the fact that any two points determine a unique line, since a line is (the graph of) a polynomial of degree 1. We say that the polynomial p *interpolates* the points (x_j, y_j). As long as we agree on the sample locations x_j in advance, we once again need exactly $n+1$ numbers to represent a polynomial of degree n.

Adding or multiplying two polynomials in this sample representation is easy, as long as they use the same sample locations x_j. To add the polynomials, just add their sample values. To multiply two polynomials, just multiply their sample values; however, if we're multiplying two polynomials of degree n, we need to *start* with $2n+1$ sample values for each polynomial, since that's how many we need to uniquely represent the product polynomial. Both algorithms run in $O(n)$ time.

Unfortunately, evaluating a polynomial in this representation is no longer trivial. The following formula, due to Lagrange, allows us to compute the value of any polynomial of degree n at any point, given a set of $n+1$ samples.

$$p(x) = \sum_{j=0}^{n-1} \left(y_j \frac{\prod_{k \neq j}(x - x_k)}{\prod_{k \neq j}(x_j - x_k)} \right) = \sum_{j=0}^{n-1} \left(\frac{y_j}{\prod_{k \neq j}(x_j - x_k)} \prod_{k \neq j}(x - x_k) \right)$$

Hopefully it's clear that formula actually describes a polynomial, since each term in the rightmost sum is written as a scaled product of monomials. It's also not hard to check that $p(x_j) = y_j$ for all j. As I mentioned earlier, the fact that this is *the only* polynomial that interpolates the points $\{(x_j, y_j)\}$ is an easy consequence of the Fundamental Theorem of Algebra. We can easily transform Lagrange's formula into an $O(n^2)$-time algorithm.

A.2.3 Summary

We find ourselves in the following frustrating situation. We have three representations for polynomials and three basic operations. Each representation allows us to almost trivially perform a different pair of operations in linear time, but the third takes at least quadratic time, if it can be done at all!

	evaluate	add	multiply
coefficients	$O(n)$	$O(n)$	$\boldsymbol{O(n^2)}$
roots + scale	$O(n)$	∞	$O(n)$
samples	$\boldsymbol{O(n^2)}$	$O(n)$	$O(n)$

A.3 Converting Between Representations

What we need are fast algorithms to convert quickly from one representation to another. That way, when we need to perform an operation that's hard for our default representation, we can switch to a different representation that makes the operation easy, perform that operation, and then switch back.

[2]This is where numerical analysis comes from.

This strategy immediately rules out the root representation, since (as I mentioned earlier) finding roots of polynomials is impossible in general, at least if we're interested in exact results.

So how do we convert from coefficients to samples and back? Clearly, once we choose our sample positions x_j, we can compute each sample value $y_j = p(x_j)$ in $O(n)$ time from the coefficients using Horner's rule. So we can convert a polynomial of degree n from coefficients to samples in $O(n^2)$ time. The Lagrange formula gives us an explicit conversion algorithm from the sample representation back to the more familiar coefficient representation. If we use the naïve algorithms for adding and multiplying polynomials (in coefficient form), this conversion takes $O(n^3)$ time.

We can improve the cubic running time by observing that *both* conversion problems boil down to computing the product of a matrix and a vector. The explanation will be slightly simpler if we assume the polynomial has degree $n-1$, so that n is the number of coefficients or samples. Fix a sequence $x_0, x_1, \ldots, x_{n-1}$ of sample *positions*, and let V be the $n \times n$ matrix where $v_{ij} = x_i^j$ (indexing rows and columns from 0 to $n-1$):

$$V = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{bmatrix}.$$

The matrix V is called a *Vandermonde* matrix. The vector of coefficients $\vec{a} = (a_0, a_1, \ldots, a_{n-1})$ and the vector of sample *values* $\vec{y} = (y_0, y_1, \ldots, y_{n-1})$ are related by the matrix equation

$$V\vec{a} = \vec{y},$$

or in more detail,

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{bmatrix}.$$

Given this formulation, we can clearly transform any coefficient vector $\vec{a}$ into the corresponding sample vector $\vec{y}$ in $O(n^2)$ time.

Conversely, if we know the sample values $\vec{y}$, we can recover the coefficients by solving a system of n linear equations in n unknowns; this takes $O(n^3)$ time if we use Gaussian elimination. But we can speed this up by implicitly hard-coding the sample positions into the algorithm, To convert from samples to coefficients, we can simply multiply the sample vector by the inverse of V, again in $O(n^2)$ time.

$$\vec{a} = V^{-1}\vec{y}$$

Computing V^{-1} would take $O(n^3)$ time if we had to do it from scratch using Gaussian elimination, but because we fixed the set of sample positions in advance, the matrix V^{-1} can be written directly into the algorithm.[3]

So we can convert from coefficients to sample value and back in $O(n^2)$ time, which is pointless, because we can add, multiply, or evaluate directly in either representation in $O(n^2)$ time. But wait!

[3]Actually, it is possible to invert an $n \times n$ matrix in $o(n^3)$ time, using fast matrix multiplication algorithms that closely resemble Karatsuba's sub-quadratic divide-and-conquer algorithm for integer/polynomial multiplication.

There's a degree of freedom we haven't exploited—***We get to choose the sample positions!*** Our conversion algorithm may be slow only because we're trying to be too general. Perhaps, if we choose a set of sample points with just the right kind of recursive structure, we can do the conversion more quickly. In fact, there is a set of sample points that's perfect for the job.

A.4 The Discrete Fourier Transform

Given a polynomial of degree $n-1$, we'd like to find n sample points that are somehow as symmetric as possible. The most natural choice for those n points are the *nth roots of unity*; these are the roots of the polynomial $x^n - 1 = 0$. These n roots are spaced exactly evenly around the unit circle in the complex plane.[4] Every nth root of unity is a power of the *primitive* root

$$\boxed{\omega_n = e^{2\pi i/n} = \cos\frac{2\pi}{n} + i\sin\frac{2\pi}{n}.}$$

A typical nth root of unity has the form

$$\omega_n^j = e^{(2\pi i/n)j} = \cos\left(\frac{2\pi}{n}j\right) + i\sin\left(\frac{2\pi}{n}j\right).$$

These complex numbers have several useful properties for any integers n and k:

- There are only n different nth roots of unity: $\omega_n^k = \omega_n^{k \bmod n}$.
- If n is even, then $\omega_n^{k+n/2} = -\omega_n^k$; in particular, $\omega_n^{n/2} = -\omega_n^0 = -1$.
- $1/\omega_n^k = \omega_n^{-k} = \overline{\omega_n^k} = (\overline{\omega_n})^k$, where the bar represents complex conjugation: $\overline{a+bi} = a - bi$
- $\omega_n = \omega_{kn}^k$. Thus, every nth root of unity is also a (kn)th root of unity.

If we sample a polynomial of degree $n-1$ at the nth roots of unity, the resulting list of sample values is called the *discrete Fourier transform* of the polynomial (or more formally, of the coefficient vector). Thus, given an array $P[0\,..\,n-1]$ of coefficients, the discrete Fourier transform computes a new vector $P^*[0\,..\,n-1]$ where

$$\boxed{\boxed{P^*[j] = p(\omega_n^j) = \sum_{k=0}^{n-1} P[k]\cdot\omega_n^{jk}}}$$

We can obviously compute P^* in $O(n^2)$ time, but the structure of the nth roots of unity lets us do better. But before we describe that faster algorithm, let's think about how we might invert this transformation.

Recall that transforming coefficients into sample values is a *linear* transformation; the sample vector is the product of a Vandermonde matrix V and the coefficient vector. For the discrete Fourier transform, each entry in V is an nth root of unity; specifically,

$$\boxed{v_{jk} = \omega_n^{jk}}$$

[4] In this lecture, i always represents the square root of -1. Computer scientists are used to thinking of i as an integer index into a sequence, an array, or a for-loop, but we obviously can't do that here. The physicist's habit of using $j = \sqrt{-1}$ just delays the problem (How do physicists write quaternions?), and typographical tricks like I or $\mathbf{i}$ or Mathematica's ⅈ are just stupid.

for all integers j and k. Thus,

$$V = \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \cdots & \omega_n^{2(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \cdots & \omega_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \cdots & \omega_n^{(n-1)^2} \end{bmatrix}$$

To invert the discrete Fourier transform, converting sample values back to coefficients, we just have to multiply P^* by the inverse matrix V^{-1}. But the following amazing fact implies that this is almost the same as multiplying by V itself:

Claim: $V^{-1} = \overline{V}/n$

Proof: Let $W = \overline{V}/n$. We just have to show that $M = VW$ is the identity matrix. We can compute a single entry in M as follows:

$$m_{jk} = \sum_{l=0}^{n-1} v_{jl} \cdot w_{lk} = \sum_{l=0}^{n-1} \omega_n^{jl} \cdot \overline{\omega_n}^{lk}/n = \frac{1}{n}\sum_{l=0}^{n-1} \omega_n^{jl-lk} = \frac{1}{n}\sum_{l=0}^{n-1} (\omega_n^{j-k})^l$$

If $j = k$, then $\omega_n^{j-k} = \omega_n^0 = 1$, so

$$m_{jk} = \frac{1}{n}\sum_{l=0}^{n-1} 1 = \frac{n}{n} = 1,$$

and if $j \neq k$, we have a geometric series

$$m_{jk} = \sum_{l=0}^{n-1} (\omega_n^{j-k})^l = \frac{(\omega_n^{j-k})^n - 1}{\omega_n^{j-k} - 1} = \frac{(\omega_n^n)^{j-k} - 1}{\omega_n^{j-k} - 1} = \frac{1^{j-k} - 1}{\omega_n^{j-k} - 1} = 0.$$

That's it! □

In other words, if $W = V^{-1}$ then $w_{jk} = \overline{v_{jk}}/n = \overline{\omega_n^{jk}}/n = \omega_n^{-jk}/n$. What this means for us computer scientists is that any algorithm for computing the discrete Fourier transform can be easily modified to compute the inverse transform as well.

A.5 Divide and Conquer!

The symmetry in the roots of unity also allow us to compute the discrete Fourier transform efficiently using a divide and conquer strategy. The basic structure of the algorithm is almost the same as MergeSort, and the $O(n \log n)$ running time will ultimately follow from the same recurrence. The *Fast Fourier Transform* algorithm, popularized by Cooley and Tukey in 1965[5], assumes that n is a power of two; if necessary, we can just pad the coefficient vector with zeros.

[5]Actually, the FFT algorithm was previously published by Runge and König in 1924, and again by Yates in 1932, and again by Stumpf in 1937, and again by Danielson and Lanczos in 1942. So of course it's often called the Coley-Tukey algorithm. But the algorithm was first *used* by Gauss in the 1800s for calculating the paths of asteroids from a finite number of equally-spaced observations. By hand. Fourier himself always did it the hard way.

Cooley and Tukey apparently developed their algorithm to help detect Soviet nuclear tests without actually visiting Soviet nuclear facilities, by interpolating off-shore seismic readings. Without their rediscovery of the FFT algorithm, the nuclear test ban treaty would never have been ratified, and we'd all be speaking Russian, or more likely, whatever language radioactive glass speaks.

Let $p(x)$ be a polynomial of degree $n-1$, represented by an array $P[0\,..\,n-1]$ of coefficients. The FFT algorithm begins by splitting p into two smaller polynomials u and v, each with degree $n/2-1$. The coefficients of u are precisely the the even-degree coefficients of p; the coefficients of v are the odd-degree coefficients of p. For example, if $p(x) = 3x^3 - 4x^2 + 7x + 5$, then $u(x) = -4x + 5$ and $v(x) = 3x + 7$. These three polynomials are related by the equation

$$\boxed{p(x) = u(x^2) + x \cdot v(x^2).}$$

In particular, if x is an nth root of unity, we have

$$p(\omega_n^k) \;=\; u(\omega_n^{2k}) + \omega_n^k \cdot v(\omega_n^{2k}).$$

Now we can exploit those roots of unity again. Since n is a power of two, n must be even, so we have $\omega_n^{2k} = \omega_{n/2}^k = \omega_{n/2}^{k \bmod n/2}$. In other words, the values of p at the nth roots of unity depend on the values of u and v at $(n/2)$th roots of unity.

$$p(\omega_n^k) \;=\; u(\omega_{n/2}^{k \bmod n/2}) + \omega_n^k \cdot v(\omega_{n/2}^{k \bmod n/2}).$$

But those are just coefficients in the DFTs of u and v! We conclude that the DFT coefficients of P are defined by the following recurrence:

$$\boxed{P^*[k] = U^*[k \bmod n/2] + \omega_n^k \cdot V^*[k \bmod n/2]}$$

Once the Recursion Fairy give us U^* and V^*, we can compute P^* in linear time. The base case for the recurrence is $n = 1$: if $p(x)$ has degree 0, then $P^*[0] = P[0]$.

Here's the complete FFT algorithm, along with its inverse.

```
FFT($P[0..n-1]$):
    if $n = 1$
        return $P$

    for $j \leftarrow 0$ to $n/2 - 1$
        $U[j] \leftarrow P[2j]$
        $V[j] \leftarrow P[2j+1]$

    $U^* \leftarrow$ FFT($U[0..n/2-1]$)
    $V^* \leftarrow$ FFT($V[0..n/2-1]$)

    $\omega_n \leftarrow \cos(\frac{2\pi}{n}) + i\sin(\frac{2\pi}{n})$
    $\omega \leftarrow 1$

    for $j \leftarrow 0$ to $n/2 - 1$
        $P^*[j]$       $\leftarrow U^*[j] + \omega \cdot V^*[j]$
        $P^*[j+n/2]$ $\leftarrow U^*[j] - \omega \cdot V^*[j]$
        $\omega \leftarrow \omega \cdot \omega_n$

    return $P^*[0..n-1]$
```

```
InverseFFT($P^*[0..n-1]$):
    if $n = 1$
        return $P$

    for $j \leftarrow 0$ to $n/2 - 1$
        $U^*[j] \leftarrow P^*[2j]$
        $V^*[j] \leftarrow P^*[2j+1]$

    $U \leftarrow$ InverseFFT($U[0..n/2-1]$)
    $V \leftarrow$ InverseFFT($V[0..n/2-1]$)

    $\overline{\omega_n} \leftarrow \cos(\frac{2\pi}{n}) - i\sin(\frac{2\pi}{n})$
    $\omega \leftarrow 1$

    for $j \leftarrow 0$ to $n/2 - 1$
        $P[j]$       $\leftarrow 2(U[j] + \omega \cdot V[j])$
        $P[j+n/2]$ $\leftarrow 2(U[j] - \omega \cdot V[j])$
        $\omega \leftarrow \omega \cdot \overline{\omega_n}$

    return $P[0..n-1]$
```

The overall running time of this algorithm satisfies the recurrence $T(n) = \Theta(n) + 2T(n/2)$, which as we all know solves to $T(n) = \Theta(n \log n)$.

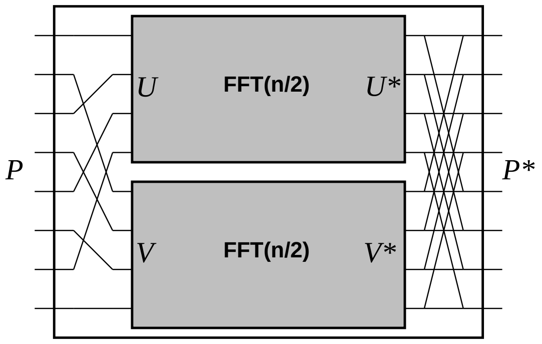
A.6 Fast Multiplication

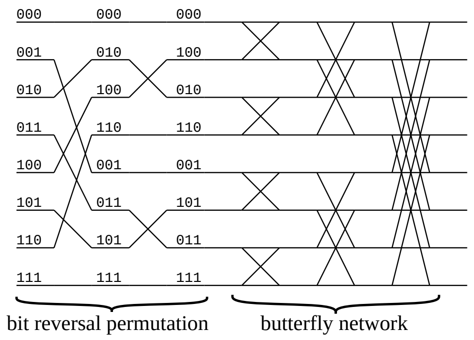
Given two polynomials p and q, each represented by an array of coefficients, we can multiply them in $\Theta(n \log n)$ arithmetic operations as follows. First, pad the coefficient vectors and with zeros until the size is a power of two greater than or equal to the sum of the degrees. Then compute the DFTs of each coefficient vector, multiply the sample values one by one, and compute the inverse DFT of the resulting sample vector.

```
FFTMultiply($P[0..n-1], Q[0..m-1]$):
    $\ell \leftarrow \lceil \lg(n+m) \rceil$
    for $j \leftarrow n$ to $2^\ell - 1$
        $P[j] \leftarrow 0$
    for $j \leftarrow m$ to $2^\ell - 1$
        $Q[j] \leftarrow 0$
    $P^* \leftarrow FFT(P)$
    $Q^* \leftarrow FFT(Q)$
    for $j \leftarrow 0$ to $2^\ell - 1$
        $R^*[j] \leftarrow P^*[j] \cdot Q^*[j]$
    return InverseFFT($R^*$)
```

A.7 Inside the FFT

FFTs are often implemented in hardware as circuits. To see the recursive structure of the circuit, let's connect the top-level inputs and outputs to the inputs and outputs of the recursive calls. On the left we split the input P into two recursive inputs U and V. On the right, we combine the outputs U^* and V^* to obtain the final output P^*.





The recursive structure of the FFT algorithm.

If we expand this recursive structure completely, we see that the circuit splits naturally into two parts. The left half computes the *bit-reversal permutation* of the input. To find the position of $P[k]$ in this permutation, write k in binary, and then read the bits backward. For example, in an 8-element bit-reversal permutation, $P[3] = P[011_2]$ ends up in position $6 = 110_2$. The right half of the FFT circuit is a *butterfly network*. Butterfly networks are often used to route between processors in massively-parallel computers, since they allow any processor to communicate with any other in only $O(\log n)$ steps.

Caveat Lector! This presentation is appropriate for graduate students or undergrads with strong math backgrounds, but it leaves most undergrads confused. You may find it less confusing to approach the material in the opposite order, as follows:

First, any polynomial can be split into even-degree and odd-degree parts:

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

We can evaluate $p(x)$ by recursively evaluating $p_{\text{even}}(x^2)$ and $p_{\text{odd}}(x^2)$ and doing $O(1)$ arithmetic operations.

Now suppose our task is to evaluate the degree-n polynomial $p(x)$ at n different points x, as quickly as possible. To exploit the even/odd recursive structure, we must choose the n evaluation points carefully. Call a set X of n values *delicious* if one of the following conditions holds:

- X has one element.
- The set $X^2 = \{x^2 \mid x \in X\}$ has only $n/2$ elements and is delicious.

Clearly the size of any delicious set is a power of 2. If someone magically handed us a delicious set X, we could compute $\{p(x) \mid x \in X\}$ in $O(n \log n)$ time using the even/odd recursive structure. Bit reversal permutation, blah blah blah, butterfly network, yadda yadda yadda.

If n is a power of two, then the set of integers $\{0, 1, \ldots, n-1\}$ is delicious, **provided we perform all arithmetic modulo n**. But that only tells us $p(x) \bmod n$, and we want the actual value of $p(x)$. Of course, we can use larger moduli: $\{0, c, 2c, \ldots, (n-1)c\}$ is delicious mod cn. We can avoid modular arithmetic entirely by using complex roots of unity—the set $\{e^{2\pi i(k/n)} \mid k = 0, 1, \ldots, n-1\}$ is delicious! The sequence of values $p(e^{2\pi i(k/n)})$ is called the *discrete Fourier transform* of p.

Finally, to invert this transformation from coefficients to values, we repeat exactly the same procedure, using the same delicious set *but in the opposite order*. Blardy blardy, linear algebra, hi dee hi dee hi dee ho.

Exercises

1. For any two sets X and Y of integers, the Minkowski sum $X + Y$ is the set of all pairwise sums $\{x + y \mid x \in X, y \in Y\}$.

   (a) Describe an analyze and algorithm to compute the number of elements in $X+Y$ in $O(n^2 \log n)$ time. *[Hint: The answer is* ***not*** *always n^2.]*

   (b) Describe and analyze an algorithm to compute the number of elements in $X+Y$ in $O(M \log M)$ time, where M is the largest absolute value of any element of $X \cup Y$. *[Hint: What's this lecture about?]*

2. (a) Describe an algorithm that determines whether a given set of n integers contains two elements whose sum is zero, in $O(n \log n)$ time.

   (b) Describe an algorithm that determines whether a given set of n integers contains *three* elements whose sum is zero, in $O(n^2)$ time.

   (c) Now suppose the input set X contains only integers between $-10000n$ and $10000n$. Describe an algorithm that determines whether X contains three elements whose sum is zero, in $O(n \log n)$ time. *[Hint: Hint.]*